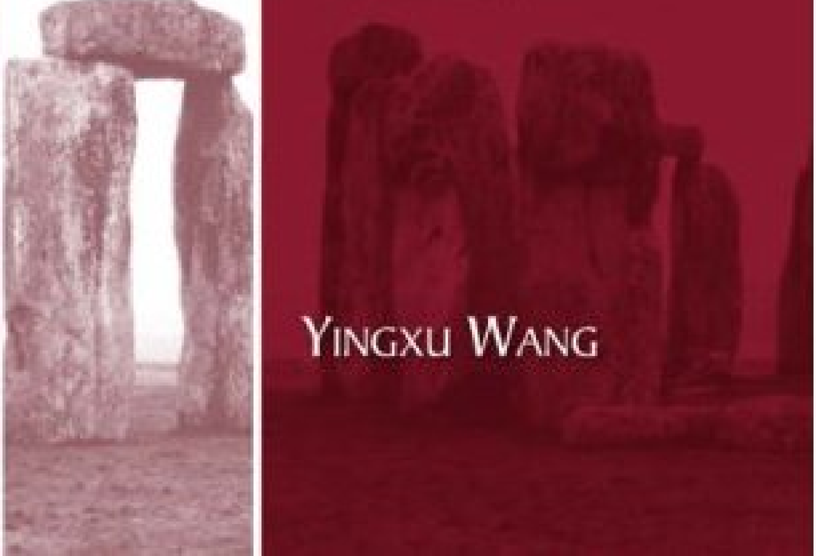
Auerbach Publications
Taylor & Francis Group

Software Engineering Foundations

A Software Science Perspective



Yingxu Wang

Software Engineering Foundations

A Software Science Perspective

Other Auerbach Publications in Software Development, Software Engineering, and Project Management

Accelerating Process Improvement Using Agile Techniques
Deb Jacobs
ISBN: 0-8493-3796-8

Antipatterns: Identification, Refactoring, and Management
Phillip A. Laplante and Colin J. Neill
IISBN: 0-8493-2994-9

Applied Software Risk Management: A Guide for Software Project Managers
C. Ravindranath Pandian
ISBN: 0-8493-0524-1

The Art of Software Modeling
Benjamin A. Lieberman
ISBN: 1-4200-4462-1

Database and Applications Security: Integrating Information Security and Data Management
Bhavani Thuraisingham
ISBN: 0-8493-2224-3

The Debugger's Handbook
J.F. DiMarzio
ISBN: 0-8493-8034-0

Defining and Deploying Software Processes
F. Alan Goodman
ISBN: 0-8493-9845-2

Effective Software Maintenance and Evolution: A Reuse-Based Approach
Stanislaw Jarzabek
ISBN: 0-8493-3592-2

Embedded Linux System Design and Development
P. Raghavan, Amol Lad, and Sriram Neelakandan
ISBN: 0-8493-4058-6

Flexible Software Design: Systems Development for Changing Requirements
Bruce Johnson, Walter W. Woolfolk, Robert Miller and Cindy Johnson
ISBN: 0-8493-2650-8

Global Software Development Handbook
Raghvinder Sangwan, Matthew Bass, Neel Mullick, Daniel J. Paulish, and Juergen Kazmeier
ISBN: 0-8493-9384-1

The Handbook of Mobile Middleware
Paolo Bellavista and Antonio Corradi
ISBN: 0-8493-3833-6

Manage Software Testing
Peter Farrell-Vinay
ISBN: 0-8493-9383-3

Maximizing ROI on Software Development
Vijay Sikka
ISBN: 0-8493-2312-6

Process-Based Software Project Management
F. Alan Goodman
ISBN: 0-8493-7304-2

Reducing Risk with Software Process Improvement
Louis Poulin
ISBN: 0-8493-3828-X

The ROI from Software Quality
Khaled El Emam
ISBN: 0-8493-3298-2

Service Oriented Enterprises
Setrag Khoshafian
ISBN: 0-8493-5360-2

Six Sigma Software Development, Second Edition
Christine B. Tayntor
ISBN: 1-4200-4426-5

Software Engineering Quality Practices
Ronald Kirk Kandt
ISBN: 0-8493-4633-9

Software Requirements: Encapsulation, Quality, and Reuse
Rick Lutowski
ISBN: 0-8493-2848-9

Software Sizing, Estimation, and Risk Management
Daniel D. Galorath and Michael W. Evans
ISBN: 0-8493-3593-0

Software Specification and Design: An Engineering Approach
John C. Munson
ISBN: 0-8493-1992-7

Software Testing and Continuous Quality Improvement, Second Edition
William E. Lewis
ISBN: 0-8493-2524-2

Strategic Software Engineering: An Interdisciplinary Approach
Fadi P. Deek, James A.M. McHugh, and Osama M. Eljabiri
ISBN: 0-8493-3939-1

Successful Packaged Software Implementation
Christine B. Tayntor
ISBN: 0-8493-3410-1

Testing Code Security
Maura A. van der Linden
ISBN: 0-8493-9251-9

UML for Developing Knowledge Management Systems
Anthony J. Rhem
ISBN: 0-8493-2723-7

X Internet: The Executable and Extendable Internet
Jessica Keyes
ISBN: 0-8493-0418-0

AUERBACH PUBLICATIONS
www.auerbach-publications.com
To Order Call: 1-800-272-7737 • Fax: 1-800-374-3401
E-mail: orders@crcpress.com

SOFTWARE ENGINEERING FOUNDATIONS

A SOFTWARE SCIENCE PERSPECTIVE

YINGXU WANG

Auerbach Publications
Taylor & Francis Group
Boca Raton New York

Auerbach Publications is an imprint of the
Taylor & Francis Group, an **informa** business

Auerbach Publications
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-0-8493-1931-0 (Hardcover)

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Library of Congress Cataloging-in-Publication Data

Wang, Yingxu.
Software engineering foundations: a software science perspective / Yingxu Wang.
p. cm. -- (The crc series in software engineering ; v. 2)
Includes bibliographical references and index.
ISBN 978-0-8493-1931-0 (alk. paper)
1. Software engineering. I. Title.

QA76.758.W375 2007
005.1--dc22 2007015773

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the Auerbach Web site at
http://www.auerbach-publications.com

To my parents, wife, and daughters

Great knowledge sees all in one. Small knowledge
breaks down into the many.

Chuang Tzu (399 – 295 BC)

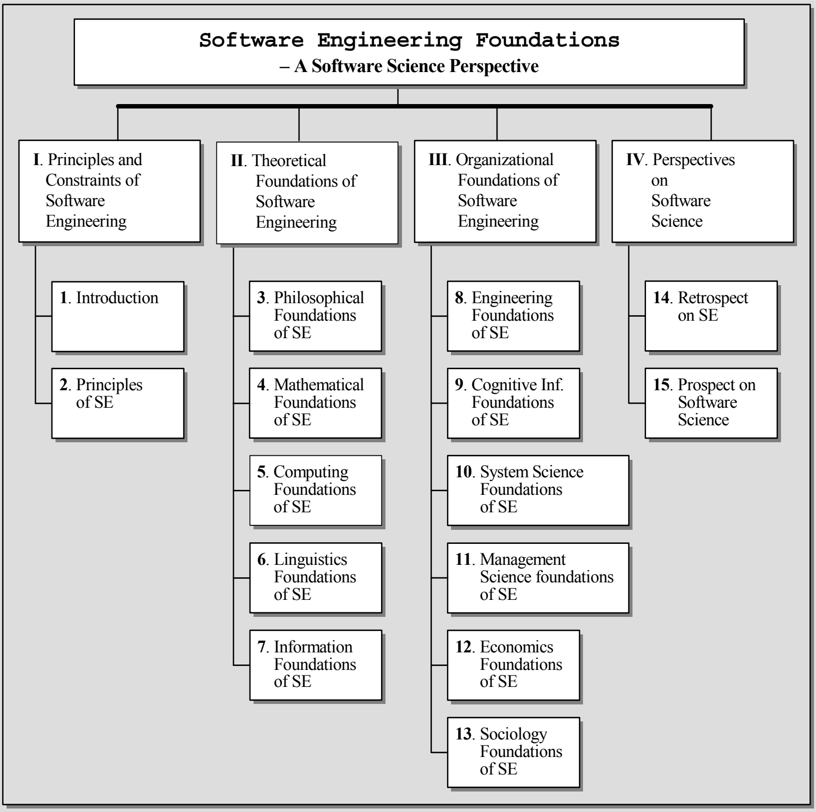
Problems that are created by our current level of thinking
cannot be solved by that same level of thinking.

Albert Einstein (1879 – 1955)

The more science becomes divided into specialized disciplines, the more
important it becomes to find unifying principles.

Herman Haken (1977)

Summary of Contents



Summary of Contents

Software Engineering Foundations

A Software Science Perspective

Part I. Principles and Constraints of Software Engineering

Part II. Theoretical Foundations of Software Engineering

Part III. Organizational Foundations of Software Engineering

Part IV. Perspectives on Software Science

Table of Contents

Preface

Software engineering is a discipline of engineering science that studies the nature of software, approaches and methodologies of large-scale software development, and theories and laws behind software behaviors and software engineering practices.

Software engineering appears still to be a young and immature science and engineering discipline characterized by a wide variety of segmented knowledge, a lack of a theoretical framework, and a bountiful inefficient industrial practice. To deal with the difficulties inherent in large-scale software development, rigorous and transdisciplinary foundations of software engineering are yet to be explored. A particular gap in the current software engineering curriculum is the missing of a fundamental framework that would provide students and practitioners for overarching, durable, and transdisciplinary theories, in order to explain a great many complicated phenomena and problems of software engineering in terms of a core set of theoretical and organizational foundations.

This book attempts to set forth a comprehensive, coherent, and rigorous framework of theoretical and empirical foundations of software engineering. It covers a wide range of necessary foundations for software engineering, such as those of philosophy, mathematics, computing, linguistics, informatics, engineering science, cognitive informatics, systems science, management, economics, and sociology.

It is recognized that two important reasons make software engineering an ideal testbed for existing theories and methodologies in the forementioned disciplines from mathematics to cognitive informatics, and from management science to sociology. The reasons are:

a) Software engineering is the latest and the most complicated engineering branch that mankind has ever experienced.

b) Software engineering is inherently a transdisciplinary field in both its theoretical foundations and empirical applications.

Constrained by the cognitive, organizational, and resources limitations and their complicated interrelations, most problems in software engineering are innately complicated. Many of them has been observed in the very beginning of software engineering for 40 years, some of them may even be traced back to more than a century ago in management science and system philosophy.

Therefore, a rigorous book is expected to explore and address a set of coherent and unified principles, foundations, theories, laws, models, frameworks, and empirical methodologies of software engineering. These have motivated the basic research into software engineering foundations as presented in this book with a software science perspective.

The Objectives of this Book

The objectives of this book on foundations of software engineering are as follows:

- To explore the whole picture of software engineering, particularly the theoretical and empirical knowledge accumulated so far in this discipline, and the fundamental problems that the discipline faces.
- To identify the fundamental cognitive, organizational, and resource constraints to software engineering.
- To reveal that all the fundamental problems in software engineering are necessarily complicated theoretical problems rather than only empirical ones.
- To recognize the need for multi-facet and transdisciplinary theories and empirical knowledge for software engineering.
- To highlight that a rigorous and formal approach is needed to seek the fundamental principles, laws, and their transdisciplinary foundations required by the nature of the problems in software engineering.
- To recognize the need for a scientific and rational coordinative work organization theory for software engineering.
- To realize the inherent limitation of the historical programming-language-centered approach to software engineering.
- To recognize the need for mathematical modeling of both software system architectures and static/dynamic behaviors, supplemented with the support of automatic code generation systems, to software engineering.

- To understand that the ultimate goal of software engineering is automated code generation, rather than intensive programmer-centered practice. Therefore, any mathematical, theoretical, and empirical means contributing towards this goal should give significant attention.
- To reveal that software engineering encompasses not only a wider domain of empirical applications, but also a richer set of theoretical essences that are closer to the root of human knowledge in terms of mathematics, philosophy, cognitive informatics, computation, sociology, and system science.
- To predict the emergence of software science on the basis of the transdisciplinary and theoretical studies on software engineering, as well as the observation on the generic pattern of science and engineering discipline maturity.

This book adopts a rigorous approach to explore the theoretical and empirical foundations of software engineering. It is a great curiosity to investigate into the transdisciplinary foundations of software engineering and the laws and theories behind them. It is also a great joyance to see a wide variety of complicated phenomena and empirical practices in software engineering can perfectly fit in the proposed theoretical framework of software science and engineering.

The Features of this Book

This book is characterized both as a comprehensive reference text for practitioners and as a *vade mecum* for students. This book is self-contained and only basic programming experience and software engineering concepts are required. This book is designed and expected to appeal to students, software engineers, scholars, and managers who are curious in exploring the theoretical and empirical foundations and laws underpinning the fast development of software engineering techniques and practice.

This book, as the first textbook on rigorous and transdisciplinary theoretical foundations of software engineering, provides the following features:

- A holistic exploration of theoretical foundations of software engineering

- A coherent framework of software engineering theories and methodologies
- A clear knowledge structure and a coherently organized body of knowledge for software engineering
- In-depth comments on alternative methodologies and approaches
- Plentiful references
- Real-world problems and heuristic questions
- Detailed guide and case studies for practitioners in the industry
- An integration of latest research findings, new methodologies, and their applications in the discipline of software engineering

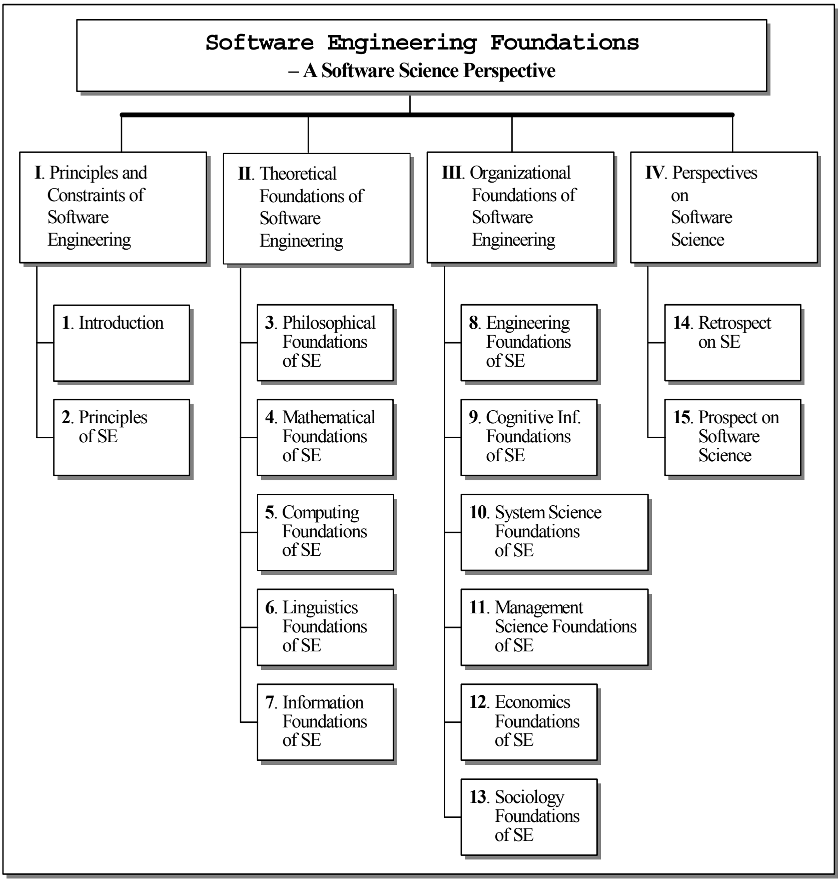
This book is needed for the following reasons:

- Software engineering is an immature and fast growing discipline. Although a number of books are available on various technologies in software engineering, a few of them has been rigorously covering the theoretical and organizational foundations of it. Now, it is the time to address this vital problem and to build a solid foundation for software engineering.
- It is recognized that software engineering depends on multidisciplinary foundations such as philosophy, computation, mathematics, informatics, system engineering, management, cognitive informatics, linguistics, and engineering economics. There was a lack of effort that attempted to put all these together and to explore the impact of the interdisciplinary approach to software engineering.
- Current software engineering is based on empirical practices, while theoretical research and investigation into foundations of software engineering have long been left behind. This book attempts to synergize theories, principles, and best practices of software engineering into a coherent framework.

This book is developed based on the author's 30-year experience in research, teaching, and industrial collaborations. This book is designed as an essential text for software engineers, students, and managers. This book provides a comprehensive and rigorous text addressing unified and integrated principles, foundations, theories, laws, frameworks, methodologies, best practices, alternative solutions, open issues for further research, and plentiful resources of software engineering. The manuscript of this book in the form of lecture notes has been successfully taught in several graduate/undergraduate courses in the software engineering program of University of Calgary for more than seven years.

The Architecture of this Book

The theoretical and empirical foundations of software engineering presented in this book encompass four parts and 15 chapters as shown in the following architecture. The four parts of this book cover principles/constraints, theoretical foundations, organizational foundations of software engineering, as well as perspectives on software science, respectively.


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
1. Introduction
2. Principles of SE
II. Theoretical Foundations of Software Engineering
3. Philosophical Foundations of SE
4. Mathematical Foundations of SE
5. Computing Foundations of SE
6. Linguistics Foundations of SE
7. Information Foundations of SE
III. Organizational Foundations of Software Engineering
8. Engineering Foundations of SE
9. Cognitive Inf. Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE
IV. Perspectives on Software Science
14. Retrospect on SE
15. Prospect on Software Science

Part I. Principles and Constraints of Software Engineering

It is recognized that software engineering requires both *theoretical* and *empirical* research. The former focuses on foundations and basic theories of software engineering, whilst the latter concentrates on heuristic principles, methodologies, tools/environments, and best practices. Although software engineering has accumulated a rich set of empirical principles, a few of them have been refined and formalized in order to form coherent theories for software engineering.

The knowledge structure of Part I on *Principles and Constraints of Software Engineering* encompasses the following chapters:

- Chapter 1. Introduction
- Chapter 2. Principles of Software Engineering

Part I addresses the nature of software and software engineering. The basic constraints to software engineering and the fundamental principles for software engineering are systematically sought. A set of 14 basic constraints is identified in Chapter 1 in three categories known as the cognitive constraints, organizational constraints, and resource constraints. Then, a set of 31 fundamental software engineering principles is elicited in Chapter 2. The usages of the fundamental principles of software engineering are perceived to be counter measures to tackle the basic constraints. Via mapping the fundamental principles into the basic constraints of software engineering, a unified framework of software engineering principles is established.

Part I will provide a whole picture of software engineering, particularly the theoretical and empirical knowledge cumulated so far in this discipline, and the fundamental problems the discipline faces. It establishes a solid basis enabling readers to investigate into the theoretical and organizational foundations of software engineering with formal, rigorous, and transdisciplinary approaches in the remainder of this book.

Part II. Theoretical Foundations of Software Engineering

Theoretical software engineering studies the nature of software, mathematical models of software architectures, mechanisms of software behaviors, methodologies of large-scale software development, and laws behind software behaviors and software engineering practices. Part II attempts to present readers the philosophical, mathematical, computing, linguistic, and informatics metaphors of software and software engineering.

It is recognized that all the fundamental problems in software engineering are complicated theoretical problems rather than only empirical ones. A rigorous and formal approach is needed to seek the fundamental principles and laws of software engineering, and their transdisciplinary foundations required by the nature of the problems in software engineering.

The knowledge structure of Part II on *Theoretical Foundations of Software Engineering* encompasses the following chapters:

- Chapter 3. Philosophical Foundations of Software Engineering
- Chapter 4. Mathematical Foundations of Software Engineering
- Chapter 5. Computing Foundations of Software Engineering
- Chapter 6. Linguistics Foundations of Software Engineering
- Chapter 7. Information Science Foundations of Software Engineering

This part focuses on fundamental theories of software engineering with the cross-fertilization among engineering philosophy, denotational mathematics, computing theories, formal linguistics, and informatics. It is noteworthy that, historically, language-centered programming had been the dominant methodology in computing and software engineering. However, it should not be taken for granted as the only approach to software engineering, because the expressive power of programming languages is inadequate to deal with complicated software systems. Further, the rigorousness and level of abstraction of programming languages are too low in modeling the dynamic architectures and behaviors of software systems. This is why a bridge in mechanical engineering or a building in civil engineering was not modeled or described by natural or artificial languages. This observation leads to the recognition of the need for mathematical modeling of both software system architectures and static/dynamic behaviors, supplemented with the support of automatic code generation systems.

Part II will establish a coherent theoretical framework of software engineering with a comprehensive set of formal principles and laws for software engineering. New structures of denotational mathematics will be developed to deal with the innate complexity of software systems. The philosophical, informatics, and linguistic theories and laws that constrain software and software engineering practice will be systematically derived. On the basis of this part, the empirical framework of software engineering, in terms of its organizational, system engineering, and cognitive informatics foundations, will be presented in the next part.

Part III. Organizational Foundations of Software Engineering

Organizational foundations of software engineering incorporate multi-facet and transdisciplinary theories and empirical knowledge for software engineering. Part III presents the organizational and system metaphors toward software engineering. Three main threads are adopted in this part known as the *system science*, *cognitive informatics*, and *organizational theories* at different levels in the domains of engineering, management science, economics, and sociology. It is recognized in this part that the hidden reasons caused so many failures of large-scale software engineering projects are not only pure technical issues, but also organizational issues of non-optimal labor allocation and the incorrect sequences of interlocked labor-duration-cost determination.

The knowledge structure of Part III on *Organizational Foundations of Software Engineering* encompasses the following chapters:

- Chapter 8. Engineering Foundations of Software Engineering
- Chapter 9. Cognitive Informatics Foundations of Software Engineering
- Chapter 10. System Science Foundations of Software Engineering
- Chapter 11. Management Science Foundations of Software Engineering
- Chapter 12. Economics Foundations of Software Engineering
- Chapter 13. Sociology Foundations of Software Engineering

This part addresses the organizational and cognitive theories and methodologies of software engineering with a transdisciplinary approach. With system science theories as an overarching framework, the organizational theories for software engineering form a hierarchical structure covering classic and contemporary thought of engineering science, management science, economics, and sociology, from the bottom up. Cognitive informatics is intensively studied in this part in order to address the cognitive constraints of software engineering.

Part III will establish the organizational foundations of software engineering with engineering science, cognitive informatics, and system science. Supplemented to Part II, this part will reveal that the particularly important aspects of software engineering theories are the organizational and cognitive theories. It will demonstrate that the profound causes that result in all the failures in software engineering history are not only pure technical issues, but also organizational issues due to the limitations of human cognitive capability.

Part IV. Perspectives on Software Science

Software engineering is immature because it lacks a theoretical framework with underpinning foundations. A vast volume of empirical knowledge has been documented in software engineering without efficiently and intensively theoretical processing and refinement. Therefore, a formal documentation of software engineering theories and fundamental body of knowledge is the key towards the maturing of software engineering. This book is devoted as a rational attempt to establish the formal and coherent theoretical framework of software engineering for its maturity.

The knowledge structure of Part IV on *Perspectives on Software Science* encompasses the following chapters:

- Chapter 14. Retrospect on Software Engineering
- Chapter 15. Prospect on Software Science

This part attempts to reveal that almost all the fundamental problems that could not be solved in the last four decades in software engineering stemmed from the lack of coherent theories in the form of software science. The objective of this part is to demonstrate how software science may be established on the basis of the theoretical foundations about it, the empirical observations on it, and the transdisciplinary knowledge gained from other much matured disciplines.

Part IV will wrap up this book by a retrospect on the coherent framework of software engineering theories, and a prospect on the structure of the emerging discipline of software science. This part reveals that software engineering encompasses not only a wider domain of empirical applications, but also a richer set of theoretical essences that are closer to the root of human knowledge. In this discipline, denotational mathematics, intelligent code generation techniques, and coordinative work organization methodologies will play significant roles in the theoretical framework of software science and engineering.

The Readership of this Book

The readership of this book is intended to include graduate, senior-level undergraduate students, and instructors in software engineering and computer science; researchers and practitioners in software engineering; and software engineers and managers in the software industry.

This book provides a comprehensive and rigorous text addressing unified and integrated principles, foundations, theories, models, frameworks, methodologies, empirical approaches, open issues for further research, and comprehensive resources with bibliography and indexes.

One of the graduate students and an experienced full-time software engineer commented on a course based on the manuscript of this book as follows:

> "This course sought to identify and explore the varied knowledge and disciplines that form the foundations of software engineering. While it is recognized that software engineering is a discipline which branches from the work of computer science, it should have, at its core, a broader and multidisciplinary base of knowledge. There are two aspects that stand out when I reflect on this course. The first is the exposure to the historical work of software engineering, especially through the classic papers that I had not previously encountered. The second is the concept of the multidisciplinary foundations of software engineering, the first time I have seen them gathered together and made explicit. I was pleasantly surprised on the nature of the course, and am happy I have had the chance to take this class. I had not considered the implication that the foundations and roots of software engineering were not established, defined, or understood."

This book is self-contained and only basic programming experience and software engineering concepts are required. This book is designed and expected to appeal to students, developers, scholars, and managers because software engineering theories and methodologies are leading the agenda in the light of the information era.

This book may be used for a one-semester or two-semester course on *Theoretical Foundations of Software Engineering* at undergraduate or graduate level. In the case of a one-semester course, Parts I, II, IV, and Chapter 8 are recommended. For graduate courses, this book may be tailored flexibly. The chapters in Part II may be composed for a graduate course on *Theoretical Software Engineering* in computer science and/or software engineering programs. The chapters in Part III may be selected for a course on *Organizational Foundations of Software Engineering* for graduate students majoring in software engineering. Some chapters of this book may also be selected for graduate seminars on *abstract systems, formal theories of management science, computational psychology,* and/or *theoretical foundations of engineering economics*.

Acknowledgments

This work is carried out in collaboration with IEEE Technical Council on Software Engineering (TCSE), IEEE Software Engineering Standard Committee (SESC), the Steering Committee of the IEEE International Conference on Cognitive Informatics (ICCI), the International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), Canadian Advisory Committee (CAC) to ISO JTC1/SC7 on Software and System Engineering, Natural Science and Engineering Research Council (NSERC), Canadian Innovation Foundation (CFI), IEEE Canadian Committee for Conferences on Computer and Software Engineering Education, Alberta Informatics Circle (iCore), IBM, NORTEL, BC-Telus, the IEEE Southern Alberta Section, and the Calgary Software Quality Interest Group. The author would like to acknowledge their support to this fundamental research program.

The author would like to acknowledge the inspiration from the work of Friedrich L. Bauer, Edsger W. Dijkstra, Tony C.A.R. Hoare, David L. Parnas, Ole-Johan Dahl, Frederick P. Brooks, Jeff Kramer, Barry Boehm, Geoff Dromey, Victor Basili, Nancy G. Leveson, Ian Sommerville, John McDermid, and Dines Bjoner.

I would like to thank my colleagues Witold Kinsner, Witold Pedrycz, Dilip Patel, Shushma Patel, Graham King, Jeffrey Tsai, Philip Sheu, Du Zhang, Yiyu Yao, Vijay Sood, Christine Chan, Colette Rolland, Tom Mainbaum, Franck Barbier, Mou Hu, Jichao Ban, Keith Chan, Motoei Azuma, Chaochen Zhou, Fuqing Yang, Bo Zhang, Junliang Chen, Jifeng He, Yixin Zhong, Ronald H. Johnston, Chan Wirasinghe, Guenther Ruhe, Len T. Bruton, Om P. Malik, Raj M. Rangayyan, Jim W. Haslett, Abu Sesay, Henry Leung, Mike R. Smith, Laurence E. Turner, and Franck Maurer, for many enjoyable discussions and fruitful research collaborations.

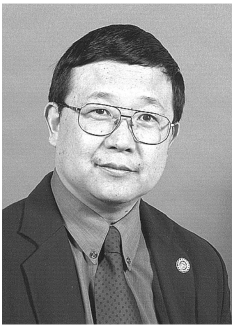
I would like to acknowledge all students who attended courses SENG 609.19, SENG 523, SENG 609.26, and SENG 609.27 at undergraduate and graduate levels at the University of Calgary in the last seven years and witnessed the development of the theories and foundations of software engineering as presented in this book. I would like to thank two of the PhD candidates in my Theoretical and Empirical Software Engineering Research Center (TESERC), Peter Tan and Cyprian Ngolah, for their teaching

assistantship for the above courses and work on the development of supporting tools for Real-Time Process Algebra (RTPA).

I would like to thank my publisher Rich O'Hanley, editors John Wyzalek and Karen Simon, for their professional advice and support. I would like to acknowledge the practical assistance and valuable help of Dr. Jingzhou Li, Angela Rundle, Ella Gee, Lisa Fleece, Huiling Yang, Janelle McConnell, Jihua Ma, and Siyuan Wang.

Yingxu Wang

About the Author



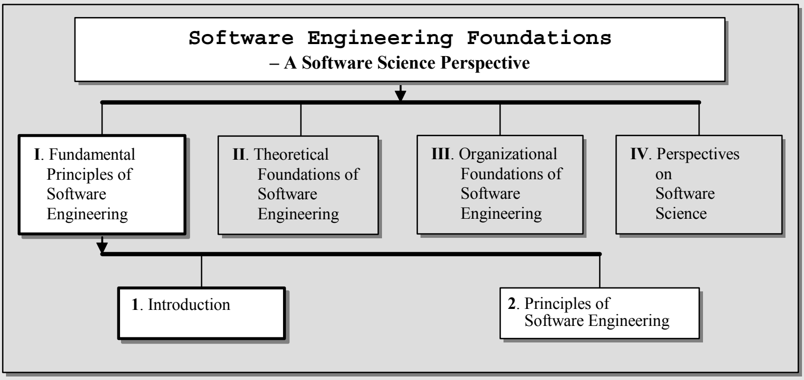
Yingxu Wang is Professor of Software Engineering and Cognitive Informatics, Director of Theoretical and Empirical Software Engineering Research Center (TESERC), and Director of International Center for Cognitive Informatics (ICfCI) at the University of Calgary. He received a PhD in Software Engineering from the Nottingham Trent University, UK, in 1997, and a BSc in Electrical Engineering from Shanghai Tiedao University in 1983. He was a Visiting Professor in the Computing Laboratory at Oxford University during 1995, and has been a full professor since 1994.

Prof. Wang is a Fellow of WIF, a P. Eng of Canada, a Senior Member of IEEE, and a member of ACM, ISO/IEC JTC1, the Canadian Advisory Committee (CAC) for ISO, the Advisory Committee of IEEE Canadian Conferences on Electrical and Computer Engineering (CCECE), and the National Committee of Canadian Conferences on Computer and Software Engineering Education (C3SEE). He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is the founding Editor in Chief of the *International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), Editor of *CRC book series in Software Engineering*, and Editor in Chief of *IGI book series on Advances in Cognitive Informatics and Natural Intelligence*. He has accomplished a number of European Union, Canadian, and industry-funded research projects as principal investigator and/or coordinator, and has published over 300 papers, including more than 50 journal papers, and 12 books in software engineering and cognitive informatics. He has served on numerous editorial boards and program committees, and as guest editors for a number of academic journals. He has won dozens of research achievement, best paper, and teaching awards in the last 30 years, particularly the IBC 21st Century Award for Achievement "*in recognition of outstanding contribution in the field of Cognitive Informatics and Software Science*," and the National Zhan Tianyou Young Scientist Prize (one of the first ten winners) in China in 1994.

The author can be reached at *yingxu@ucalgary.ca* or *yingxu.wang@ieee.org*. For further details, see: http://www.enel.ucalgary.ca/People/wangyx/Books/SEF.

PART I

PRINCIPLES AND CONSTRAINTS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Fundamental Principles of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
1. Introduction
2. Principles of Software Engineering

S*oftware engineering* is an engineering discipline that studies the nature of software, approaches and methodologies of large-scale software development, and the theories and laws behind software behaviors and software engineering practices. The nature of software engineering and its theories and methodologies are determined by the nature of the objects under study, *software*, and the needs for adequate and denotational mathematical, theoretical, and methodological means for this discipline. Although software engineering has accumulated a rich set of empirical and heuristic principles, a few of them have been refined and formalized in order to form coherent theories for software engineering.

It is recognized that software engineering requires both *theoretical* and *empirical* research. The former focuses on foundations and basic theories of software engineering, whilst the latter concentrates on fundamental principles, tools/environments, and best practices.

The knowledge structure of Part I on *Principles and Constraints of Software Engineering* is as follows:

- Chapter 1. Introduction
- Chapter 2. Principles of Software Engineering

This part addresses the nature of software and software engineering. The basic constraints to software engineering and the fundamental principles for software engineering are systematically studied. A set of 14 basic constraints is identified in Chapter 1 in three categories known as the *cognitive* constraints, *organizational* constraints, and *resource* constraints. Then, a set of 31 fundamental software engineering principles are elicited in Chapter 2. The usages of the fundamental principles of software engineering are perceived to be counter measures to tackle the basic constraints. Via mapping the fundamental principles into the basic constraints of software engineering, a unified framework of software engineering principles is established.

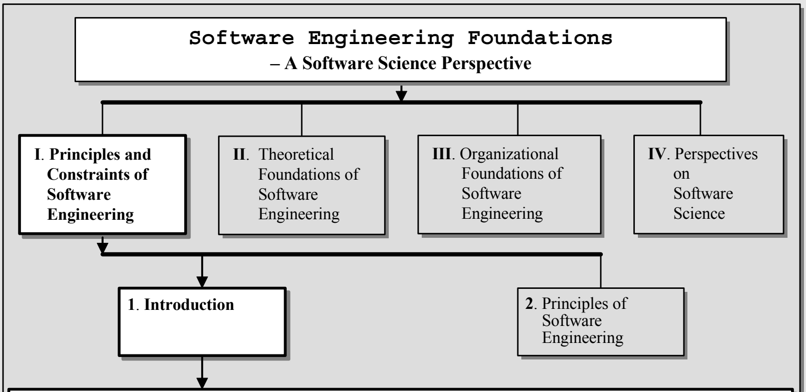
Chapter 1, *Introduction*, presents fundamental concepts, structures, and constraints of software engineering, and explores the problem domain of software engineering. The essences of software as instructive and behavioral information, the fundamental problems, and basic constraints of software engineering are identified. The approaches to software engineering are explored in the context of how the basic problems of software engineering are coped with. Then, the construction of theoretical and transdisciplinary foundations of software engineering is presented as a strategic approach towards software engineering. The architecture of this book, as well as interrelationships and dependency between the four parts and 15 chapters of this book, is systematically overviewed.

Chapter 2, *Principles of Software Engineering*, surveys the vast literature of software engineering in order to elicit and summarize the pioneer pursuits of software engineering principles in the last four decades. A comprehensive set of fundamental principles for software engineering is identified, which provides a whole picture for understanding the theories and foundations of software engineering. Based on the survey and comparative study, a unified framework of 31 software engineering principles is developed, which may be adopted as powerful measures for tackling the basic constraints to software engineering.

Part I will provide an overarching framework of software engineering by reviewing the theoretical and empirical knowledge accumulated in this discipline. It will also identify the fundamental problems the discipline faces. This part will establish a solid basis enabling readers to investigate into the theoretical and organizational foundations of software engineering with formal, rigorous, and transdisciplinary approaches in the remainder of this book.

Chapter 1

INTRODUCTION


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
1. Introduction
2. Principles of Software Engineering

1. Introduction

Knowledge Structure

- What is Software Engineering?
  - What is Software?
  - What is Software Engineering?
  - What is the Status of Software Engineering as an Engineering Discipline?
- Characteristics of Software Engineering
  - The Nature of Software
  - Perceptions on Software Engineering
  - Software Engineering as an Engineering Discipline
  - Hierarchy of Abstraction and Descriptivity in Software Engineering
- Basic Constraints of Software Engineering
  - The Software Engineering Constraint Model
  - Cognitive Constraints of Software Engineering
  - Organizational Constraints of Software Engineering
  - Resources Constraints of Software Engineering
- Approaches to Software Engineering
  - Programming Methodologies
  - Automated Software Engineering
  - Software Engineering Processes
  - Software Development Models
  - Formal Methods
  - Theoretical Foundations of Software Engineering
- Transdisciplinary Foundations of Software Engineering
  - Philosophical Foundations
  - Computing Foundations
  - Information Science Foundations
  - Cognitive Informatics Foundations
  - Management Science Foundations
  - Sociology Foundations
  - Mathematical Foundations
  - Linguistics Foundations
  - Engineering Foundations
  - Systems Science Foundations
  - Economics Foundations
- The Architecture of this Book

Learning Objectives

- To view software engineering as a scientific and engineering discipline.
- To be aware of the nature of software and the characteristics of software engineering.
- To understand the fundamental constraints of software engineering.
- To be aware of the approaches to software engineering.
- To appreciate the needs for seeking the transdisciplinary foundations of software engineering.
- To understand the architecture of this book.

"There is nothing more practical than a good theory."

Immanuel Kant (1724-1804)

"If our problems in building and interacting with complex systems are really rooted in intellectual manageability and human limits in managing complexity, then we will need to stretch these limits to build ever more complex systems."

N.G. Leveson (1997)

"Software engineering education can, and must, focus on fundamentals."

David L. Parnas (1998)

1.1 Overview

S*oftware engineering* is an increasingly important discipline that studies the nature of software, approaches and methodologies for large-scale software development, and the theories and laws behind software behaviors and software engineering practices.

Software, the object under study in software engineering, is a unique abstract structure, which will be rigorously treated by the mathematics, product, information, system, cognitive informatics, and intelligent behavioral metaphors throughout this book. As a result, the theoretical and transdisciplinary foundations of software engineering will be established in line with what Kant (1724 – 1804) asserted: "There is nothing more practical than a good theory."

Fundamental problems yet to be explored in software engineering are identified, *inter alia,* as follows:

- What is the nature of software?
- What are the basic constraints of software engineering?
- What are the mathematical means required in software engineering?
- What are the engineering approaches to software engineering?
- Why have more than half of software engineering projects failed in the history? Is this a theoretical, organizational, or operational problem?
- What are the attributes of software quality and whether can they be quantitatively measured?

- Is time and labor interchangeable in software engineering? If so, what are the constraints for the interchangeability between them?
- How is a project team optimally organized in large-scale software engineering projects?
- How may the software industry be systematically organized?

None of the fundamental issues shown above would be pursued solely by empirical means or simply following common senses in practice. A rigorous and theoretical approach is needed to seek the fundamental principles and laws of software engineering and their transdisciplinary foundations. This is required by the nature of problems of software engineering, which may be classified into two categories known as the *theoretical* and *empirical* problems as shown in Table 1.1. For instance, a generic program model and an abstract work organization methodology are theoretical problems, while a specific application and a given data structure are empirical ones.

Table 1.1
Theoretical vs. Empirical Problems in Software Engineering

Category of problems	**Typical means**	**Typical methodology**
Theoretical	Abstract, mathematics-based	Inductive, formal inferences
Empirical	Concrete, data-based	Deductive, experimental validation

Table 1.1 contrasts the basic characteristics of these two categories of problems in terms of their typical means and methodologies. According to Table 1.1, the criteria to distinguish *theoretical and empirical* problems under study in software engineering can be derived in the following theorem.

The 1st Law of Software Engineering

Theorem 1.1 *Software engineering problems* must be treated by both *theoretical* and *empirical* methodologies. The former is characterized by abstract, inductive, mathematics-based, and formal-inference-centered studies; while the latter is characterized by concrete, deductive, data-based, and experimental-validation-centered studies.

The theme of this book is set forth on the enquiry of both theoretical and empirical problems in software engineering, particularly the former, where most of the key problems remain.

The objectives of this book on foundations of software engineering are as follows:

- To explore the whole picture of software engineering, particularly the theoretical and empirical knowledge accumulated so far in this discipline, and the fundamental problems that the discipline faces.
- To identify the fundamental cognitive, organizational, and resource constraints to software engineering.
- To reveal that all the fundamental problems in software engineering are complicated theoretical problems rather than only empirical ones.
- To recognize the need for multi-facets and transdisciplinary theories and empirical knowledge for software engineering.
- To highlight that a rigorous and formal approach is needed to seek the fundamental principles, laws, and their transdisciplinary foundations required by the nature of the problems in software engineering.
- To recognize the need for a scientific and rational work organization theory for software engineering.
- To realize the inherent limitation of the historical programming-language-centered approach to software engineering.
- To recognize the need for mathematical modeling of both software system architectures and static/dynamic behaviors, supplemented with the support of automatic code generation systems, to software engineering.
- To understand that the ultimate goal of software engineering is automated code generation, rather than intensive programmer-centered practice. Therefore, any mathematical, theoretical, and empirical means helping towards this goal should give significant attention.
- To predict the emergence of software science on the basis of the transdisciplinary theoretical studies on software engineering, as well as the observation on the generic patterns of engineering discipline maturity.

This book will reveal that software engineering encompasses not only a wider domain of empirical applications, but also a richer set of theoretical essences that are closer to the root of human knowledge in terms of mathematics, philosophy, cognition, informatics, computation, sociology, and system science.

1.1.1 SOFTWARE ENGINEERING: STATUS AND PROBLEMS

Any matured science and engineering discipline has a stable pyramid structure among its foundations, education, and practices/applications. So will software engineering, with the ideal logical structure as shown in Fig. 1.1.

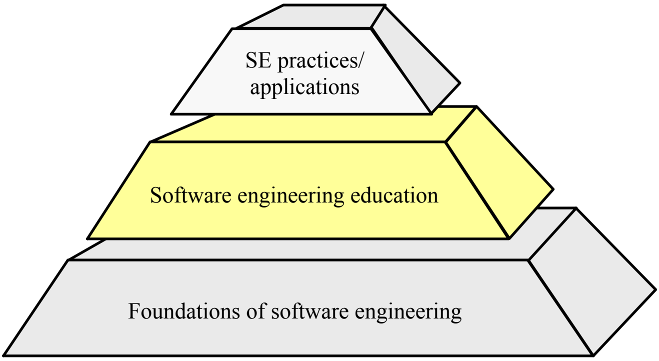

SE practices/ applications
Software engineering education
Foundations of software engineering

Figure 1.1 Relationship between software engineering foundations, education, and practices/applications

However, the current status of software engineering as a discipline demonstrates an upside down pyramid, where the whole field is driven by industrial practice and technical innovations. Theories and fundamental research in software engineering, particularly the laws that constrain software behaviors and software engineering practice, have been left behind or overlooked, if not been ignored or perceived inexist. As a consequence, software engineering educators had no solid and durable theoretical framework to base in teaching. Instead, they were busy explaining an extremely wide variety of practices, techniques, and tools as in a fashion industry.

It is observed by McDonnell in 1999 that the average *half-life* of most software engineering techniques is only about two to three years [McConnell, 1999]. That is, after every year, 15% to 25% techniques one has acquired in software engineering practice will be obsolete. What a young field where techniques had never been durable!

The special phenomenon in software engineering, in which we are still facing the same problems as those identified in the very beginning of software engineering four decades ago [Bauer, 1972/1976; Naur and Randell, 1969, Ashenhurst and Graham, 1987], indicates that the current software engineering theories, foundations, and mathematical means are *inadequate*, and the current empirical approach toward programming is *insufficient*. It also indicates that the problems and the objects of our study in software engineering are fundamentally complicated and unique, which require new forms of mathematics and new theories different from those that deal with entities in the physical world or in the conventional engineering disciplines [McDermid, 1991; Pressman, 1992; Sommerville, 1996; Pfleeger, 1998; Peters and Pedrycz, 2000; Vliet, 2000; Wang and King, 2000a; Wang and Patel, 2000; Broy and Denert, 2002; Wang and Bryant, 2002; Wang, 2000/05a/05d/05f/05g/05h/05i/05j/05k/05*l*/06a/06c/06f/06g/06h/06i].

The profound uniqueness of the discipline of software science and engineering lies on the fact that its objects under study are located in a dual world as described below and formalized in Theorem 1.2 [Wang, 2002d/2003a/2006a].

Definition 1.1 The *general worldview*, as shown in Fig. 1.2, reveals that the *natural world* (NW) is a dual world encompassing both the *physical* (concrete) *would* (PW) and the *abstract* (perceived) *would* (AW).

The 2nd Law of Software Engineering

Theorem 1.2 The *Information-Matter-Energy* (IME) model states that the natural world (*NW*) which forms the context of human intelligence and software science is a dual: one aspect of it is the *physical* world (*PW*), and the other is the *abstract* world (*AW*), where *matter* (*M*) and *energy* (*E*) are used to model the former, and *information* (*I*) to the latter, i.e.:

$$\begin{aligned} NW &\hat{=} PW \;||\; AW \\ &= \boldsymbol{p}(M,E) \;||\; \boldsymbol{a}(I) \\ &= \boldsymbol{n}(I,M,E) \end{aligned} \tag{1.1}$$

where || denotes a parallel relation, and $\boldsymbol{p}$, $\boldsymbol{a}$, and $\boldsymbol{n}$ are functions that determine a certain *PW*, *AW*, or *NW*, respectively.

According to the IME model [Wang, 2004b/06b/2007a], information plays a vital role in connecting the physical world with the abstract world. Models of the natural world have been well studied in physics and other natural sciences. However, the modeling of the abstract world is still a fundamental issue yet to be explored in cognitive informatics, computing, software science, knowledge engineering, brain sciences, and cognitive informatics. Especially, the relationships between I-M-E and their transformations are deemed as one of the fundamental questions in science and engineering.

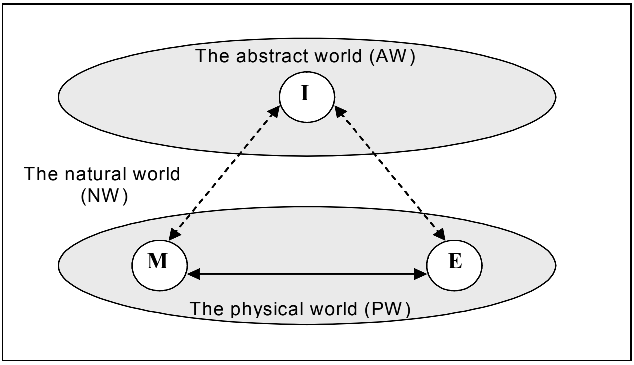


Figure 1.2 The IME model of the general worldview

Corollary 1.1 The *natural world NW*(*I, M, E*), particularly the part of the *abstract world AW*(*I*), is deemed diversely by individuals because of the differences of perceptions and mental contexts among them.

Corollary 1.1 indicates that although the physical world *PW*(*M, E*) is the same to everybody, the natural world *NW*(*I, M, E*) is unique to different individuals, because the abstract world *AW*(*I*), as a part of it, is subjective depending on the information that individual obtains and perceives.

Software is a special type of behavioral information of computing and a means of interaction between the abstract world and the physical world. The nature of software makes software engineering a unique discipline, which is innately the most complicated engineering branch that human ever experienced, and inherently the most overarching transdisciplinary field in both theories and applications. These are also the reasons that set forth software engineering as an ideal testbed for existing theories and methodologies of a wide range of science and engineering disciplines from mathematics to cognitive informatics, and from management science to sociology.

1.1.2 MYTHS ON SOFTWARE ENGINEERING

A variety of myths exist on perceiving software engineering from both academics and practitioners. Some of the common ones can be described as follows:

- Software engineering has no theoretical foundation since mathematics had not played a central role in programming.
- Software engineering is not an engineering discipline rather than a branch of art, because there were few scientific laws to follow.
- Everybody can do programming; Kids even do it better sometimes.
- When one can get a program run in a programming language that displays the classical greeting "Hello world," one is then confident to claim that he/she is a programmer.
- Do we need software engineering? Whether software engineering is a faculty of empirical best practices or a system of essential theories?

It is observed that quite a few people who major in software engineering have never experienced the impact of sizes of problems in software development, where the *complexity threshold* for understanding the true problems in software engineering is considered 5,000 Lines Of Code (LOC) to 10,000 LOC [Wang and King, 2000a]. Note that LOC here is treated as a *unit* rather than the measure, which is identified as the *symbolic size* of software. When the symbolic sizes of problems are below a few hundreds LOC, programmers or students may doubt whether software engineering theories and methodologies should be applied in the development processes. However, if the sizes of problems are above the threshold, practitioners will always complain there is a lack of theories and methodologies that may provide strategic and specific help for both programmers and managers.

Software engineering theories and methodologies are developed for dealing with complexity and intellectual challenges in large-scale software development. According to cognitive psychology and empirical statistics [Wang, 2001d/03f/05j/06c; Wang and King, 2000a], the size threshold of complexity in software development is given below.

Definition 1.2 A *general complexity threshold* of software engineering is empirically set according to the symbolic size of software systems, S_0, as follows:

$$S_0 \geq 5{,}000 \quad [\text{LOC}] \tag{1.2}$$

where LOC is the *unit* of the symbolic size of software.

The complexity threshold is equivalent to 100-page source code in a typical high-level programming language and related documentation, which is above one person-year workload according to average software productivity

[Boehm, 1987; Dale and Zee, 1992; Jones, 1981/1986; Livermore, 2005]. Taking this threshold as a benchmark of system size or cognitive complexity in software system development, one may realize that only a few students who graduated from computer science or software engineering have already been trained on problems in the order of complexity beyond this threshold.

Therefore, the primary aim of software engineering theories and technologies is to facilitate students, software engineers, and managers to deal with the fundamental problems generated by the inherent complexity of software development, such as:

- How to design and implement a software system that one is not able to do by only oneself?
- How to cope with the development of a software system in which one does not completely know or understand the whole system and parts produced by other team members?

Strategic answers to the above questions will be sought in the remainder of this chapter and throughout this book.

This chapter presents fundamental concepts, structures, and constraints of software engineering. In the remainder of this chapter, Section 1.2 presents a set of basic concepts and characteristics of software engineering. Section 1.3 identifies fundamental problems and basic constraints of software engineering. The approaches to software engineering are explored in Section 1.4 in the context of how the basic problems in software engineering are coped with. Based on the preceding introductory sections, the construction of theoretical and transdisciplinary foundations of software engineering is presented in Section 1.5 as a strategic approach towards software engineering. Section 1.6 describes the architecture of this book, as well as interrelationships and dependency between the four parts and 15 chapters of this book.

1.2 Characteristics of Software Engineering

As introduced in Section 1.1, there are a variety of perceptions, even myths, toward the nature of software and software engineering from both academics and practitioners. In order to understand the fundamental characteristics of software and software engineering, this section explores the metaphors of software, and the intensions and extensions of the term software engineering.

1.2.1 PERCEPTIONS ON SOFTWARE

Software, in daily life, is simply meant as anything flexible and without a physical dimension. Software or a program system is a frequently referred concept and a widely applied human creative artifact in both the software industry and the information society. The engineering discipline for software development, software engineering, has emerged for four decades since Friedrich Bauer proposed the term in 1968 [Bauer, 1972/1976; Naur and Randell, 1969]. However, from the point of view of the development lifecycle of science and engineering disciplines, which may typically evolve hundreds of years, software engineering is still an infant. In software engineering, we are still searching a suitable mathematical means that may be used for modeling the nature of software, and seeking an understanding on what laws constrain software behaviors and software engineering processes.

A computer *program* means the code in a programming language physically or the algorithms plus typed data logically. *Software* is an integration of the program code with design and supporting documentations and intermediate work products, such as original requirements, system specification, system design and decision rationales, code configuration, test cases and results, maintenance mechanisms, and user manuals.

Observing the above list of software work products, a definition of software can be given as follows.

Definition 1.3 *Software* is an intellectual artefact that provides a *solution* for a *repeatable* computing application, which enables existing tasks to be done easier, faster, and smarter, or which provides innovative applications for the industries and everyday life.

The nature of software has been perceived quite differently in research and practice of computing and software engineering. The following perceptions on the nature of software can be found in the literature:

- Software is a mathematical entity
- Software is a concrete product
- Software is a set of behavioral information

The following subsections describe the three-type perceptions on software known as the *mathematical*, *product,* and *informatics* metaphors.

1.2.1.1 The Mathematical Metaphor of Software

The *mathematical metaphor* of software is adopted by many computer scientists who perceive software as a stored programmed logic on computing hardware [von Neumann, 1963; Dijkstra, 1976; Cries, 1981; Lewis and Papadimitriou, 1988; Hartmanis, 1994] or simply as a mathematical entity [Hoare, 1969/1986; Scott and Strachey, 1971; Hoare et al., 1987; Wang, 2005a].

A general taxonomy of the usages of computational mathematics can be derived on the basis of their relations with natural languages. It is recognized that languages are the means of thinking. Although they can be rich, complex, and powerfully descriptive, natural languages share the common and basic mechanisms, such as *to be*, *to have*, and *to do* [Wang, 2003b/06d/06e/06f/06h/06j/07a]. All mathematical means and forms, in general, are an abstract description of these three categories of human or system behaviors and their common rules. Taking this view, mathematical logic may be perceived as the abstract means for describing 'to be,' set theory for describing 'to have,' and algebras, particularly process algebra, for describing 'to do.'

This is a fundamental view toward the formal description and modeling of human and system behaviors in general, and software behaviors in particular, because a software system can be perceived as a virtual agent of human beings created to do something repeatable, to extend human capability, reachability, and/or memory capacity. The author found that both human and software behaviors can be described by a three-dimensional representative model comprising *action*, *time*, and *space*. For software system behaviors, the three dimensions are known as *mathematical operations*, *event/process timing*, and *memory manipulation* [Wang, 2002a/02b/02c].

A powerful concept introduced by C.A.R. Hoare in formal methods is the *process* [Hoare, 1986]. With this concept, the behaviors of a software system can be perceived as a set of processes composed with given rules defined in a particular process algebra. It is found that a process can be formally modeled by a set of cumulative relational statements [Wang, 2006f/06h]; further, a program can be formally modeled by a set of cumulative relational processes. The cumulative relational models of processes and programs will be described in Sections 4.6.1 and 5.5.1, respectively.

1.2.1.2 The Product Metaphor of Software

Software is conventionally deemed as a concrete product in software engineering and the IT industry [Baker, 1972; ISO 9001, 1989/94; ISO 9126, 1991; Taguchi, 1986; Jones, 1986; SQPL, 1990; Dromey, 1995]. With the *product metaphor*, a number of manufacturing technologies and quality assurance principles were introduced into software engineering. However, the phenomenon, in which we are facing almost the same fundamental problems in software engineering as we dealt with four decades ago, indicates a failure of the manufacture-based and mass-production-oriented metaphor and related technologies in software development. Therefore, we have to examine the nature of software and how we produce software in software engineering.

The primary technical deficiencies in the software industry are the inadequate of abstractive and precise description means for software architectures and behaviors, and the perplexity of labor organization in large groups and large-scale projects. The former refers to the current practice that uses less expressive and inaccurate means to describe more abstract and complicated software systems. The latter refers to the extremely high rate of interpersonal coordination and coupling that dramatically changes the nature of large-scale project organization. Both of them are rooted deeply in the theoretical deficiency of software engineering and the overlooking of the unique need of software engineering that are so fundamentally different from conventional engineering disciplines and manufacturing industries.

The descriptive power and abstraction means needed in software engineering will be discussed in Section 1.2.4, while theories and solutions for the organizational issues in software engineering will be developed in Chapters 8, 10, 11, and 13, respectively, on the engineering, system science, management science, and sociology foundations of software engineering.

1.2.1.3 The Informatics Metaphor of Software

As shown in the IME model given in Theorem 1.2 and Fig. 1.2, information is the third essence in modeling the natural world supplementing to matter and energy. According to cognitive informatics theory [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06], information is any property or attribute of entities in the natural world that can be abstracted, digitally represented, and mentally processed.

Software is both behavioral information to designers and instructive information to computers. With the informatics metaphor, the definition of software may be revised as follows.

Definition 1.4 *Software* is a specific solution for computing in order to implement a certain architecture and obtain a set of expected behaviors on a universal computer platform for a required application.

For software engineering to become a matured engineering discipline like others, it must establish its own laws and theories, which are perceived to be mainly relied on by denotational mathematics [Wang, 2002a/05a/06d/06e/06f/06j/07a] and cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06].

The informatics metaphor provides a new approach to study the nature and basic properties of software in software engineering, which forms an important part of the cognitive informatics foundations of software engineering that will be described in Chapters 7 and 9. In conventional engineering disciplines, the common approach moves from abstract to concrete, and the final product is the physical realization of an abstract design. In software engineering, however, the approach is reversed. The final software product is the virtualization and abstraction, by binary streams, of a set of original real-world requirements. The only tangible part of a software implementation is its storage media or its run-time behaviors. This is probably the most unique and interesting feature of software engineering.

In software design we need to describe the abstract architecture of the system and its components by logical and algebraic modeling techniques, and their static and dynamic behaviors in terms of actions and processes. In software system behaviors description and specification, the architectures of software refer to frameworks and patterns. Software *static behaviors* are those that can be determined at design-and-compile time, and *dynamic behaviors* are those that are indeterministic until run-time.

In recognizing that software is a special abstract system of behavioral and instructive information, the processes and techniques widely used in the publishing industry and the journalism industry are worthy to be intensively studied and adopted in software engineering.

Further discussions on the nature of software and software engineering on the *cognitive, intelligent behavioral,* and *system* properties of software will be presented in Section 3.4 on the philosophical views of software.

1.2.2 PERCEPTIONS ON SOFTWARE ENGINEERING

Before the identification of what knowledge is required for software engineering, an understanding of what is meant by 'software engineering' is needed in the first place. Readers need to know why the engineering approach seems to fit well with the goals of software development [McDermid, 1991; Pressman, 1992; Sommerville, 1996; Pfleeger, 1998; Peters and Pedrycz, 2000; Vliet, 2000; Wang and King, 2000a; Wang and Patel, 2000; Broy and Denert, 2002; Wang and Bryant, 2002; Wang, 2000/05a/05d/05f/05g/05h/05i/05j/05k/05l/06a/06c/06f/06g/06h/06i].

The term *software engineering* was initially proposed by Friedrich L. Bauer at the 1968 NATO conference on Software Engineering [Bauer, 1976;

Naur and Randell, 1969]. In his paper, Bauer defined software engineering as:

> "The establishment and use of sound engineering principles in order to obtain economical software that is reliable and works efficiently on real machines."

Bauer introduced software engineering as a solution to the so called *software crisis*. However, he did not explain what the sound engineering principles are and which of them are applicable to software engineering. That is why, after 38 years, professionals are still arguing what software engineering is and whether it makes sense to speak the engineering of software development.

Later, in 1990, IEEE Standard 610.12 defines software engineering as follows [IEEE 610.12, 1990]:

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
>
> (2) The study of approaches as in (1).

The nature of problems in software engineering has been addressed by D.L. Parnas (1972/78), F.P. Brooks (1975/87), C.A.R. Hoare (69/75/85), B. Boehm (1976/81/83), J.A. McDermid (1991), Y. Wang (2004b/04c/05a/06c/06g/06h/06i), Y. Wang and G. King (2000a), Y. Wang and D. Patel (2000), and Y. Wang and A. Bryant (2002). A summary of fundamental characteristics of software engineering is listed below:

- Inherited complexity and diversity
- Difficulty of establishing and stabilizing requirements
- Changeability or malleability of software
- Abstraction and intangibility of software
- Requirement of various problem-domain knowledge
- Nondeterministic and polysolvability in design
- Polyglotics and polymorphism in implementation
- Dependability of interactions between software, hardware, and humans

The following humour known as the '*cat theory*' had been presented at NASA's web site [NASA, 2000]:

- *Mechanical engineering* is like looking for a black cat in a lighted room.
- *Chemical engineering* is like looking for a black cat in a dark room.
- *Software engineering* is like looking for a black cat in a dark room in which there is no cat.
- *System engineering* is like looking for a black cat in dark room in which there is no cat and one yells, "I got it!"

The humour cat was posted in the 1990s and represented the statuses of relative maturities of different engineering disciplines. Why is the cat in the mechanical or chemical rooms? Because these disciplines have already developed rigorous theoretical frameworks and suitable mathematical means, but the other disciplines did not. Therefore, lack of theoretical foundations is the same situation that challenges software engineering and system engineering characterized as immature science and engineering disciplines.

Along with the research and practices of software engineering, and the speedy growing of the software industry, the definition of software engineering has further evolved. In 1991, J.A. McDermid provided an extended definition of software engineering as follows [McDermid, 1991]:

> "Software engineering is the science and art of specifying, designing, implementing and evolving – with economy, timeliness and elegance – programs, documentation and operating procedures whereby computers can be made useful to man."

This is a representative of the second-generation definitions of software engineering. Comparing the first- and second-generation definitions of software engineering, it can be seen that the former perceived software engineering as a *method* for software development while the latter implied that software engineering is both *science* and *art* for programming. Bearing in mind that the intention is to better represent trends and to recognize software engineering as an engineering discipline while deemphasizing the uncontrollable and unrepeatable aspects of programming as an art, the third-generation definition of software engineering can be represented by the following [Wang and King, 2000a].

Definition 1.5 *Software engineering* is a discipline that adopts engineering approaches, such as established methodologies, processes, measurement, tools, standards, organisation methods, management methods, quality assurance systems and the like, in the development of large-scale software seeking to result in high productivity, low cost, controllable quality, and measurable development schedule.

In order to analyze the differences between the three generations of definitions, a comparison of the intensions and extensions of these perceptions on software engineering is listed in Table 1.2. The table shows how the understanding of software engineering can be greatly improved by contrasting the perceived nature of software engineering as well as its means, aims, and attributes.

Table 1.2
Contrast of Representative Definitions of Software Engineering

Generation	Nature	Means	Object under study	Attributes of aims
1 (Bauer, 1968)	A method	Generic engineering principles	Program	• Economy • Reliability • Efficiency
2 (McDermid, 1991)	A science and art	Life cycle methods: • specification • design • implementation • evolving	Program and document	• Economy • Timeliness • Elegance
3 (Wang and King, 2000)	An engineering discipline	Engineering approaches: • methodologies • processes • measurements • tools • standards • organizational methods • management methods • quality assurance systems	Large scale software	• Productivity • Quality • Cost • Time

It is noteworthy that the perceived nature, means, and aims together with the attributes of their definitions of software engineering have been evolved over time. The first-generation definition proposed software engineering as a method or approach to software development; the second-generation definition focused on scientific methods and art for programming. The third-generation definition portrays software engineering as an engineering discipline for large-scale software development in an industrialized context.

1.2.3 SOFTWARE ENGINEERING AS AN ENGINEERING DISCIPLINE

To many professionals engineering means systematic planning, teamwork, rigorous process, repeatability, and efficiency. Software professionals have been arguing the term “software + engineering” and its implication for four decades since Friedrich Bauer first proposed it in 1968 [Bauer, 1976]. Yet, still some fundamental questions remain, such as:

a. Is software development an engineering discipline?

b. Are software developers engineers or craftsmen?

There were completely different assertions and opinions on the above key issues of "to be or not to be" that is still confusing the academics, practitioners, and students in software engineering and in the software industry.

In investigating these fundamental problems, the author finds that the myths were caused by a confusion of time in perceiving software development as, or as not, an engineering discipline. A rational answer to the question whether software development is an engineering discipline is 'not' at present and in the past, and it is going to be and should be 'yes' in the future. Currently, software development is evolving from the laboratory-oriented and all-round-programmer-based practice to an industry-oriented and process-based platform, and software developers are experiencing changes of roles from craftsmen to regulated professionals – the software engineers. The practices of the former are based on personal talents, tastes, and art, while those of the latter are based on theoretical foundations, disciplined processes, and repeatable professional activities.

With an evolution point of view, software engineering is actually a young discipline, which is located in the art age and is in a transition to the engineering age, though there is still some way to go for software engineering to be a matured engineering discipline.

IEEE/ACM identified the following characteristics of software engineering in the Computing Curricula – Software Engineering (CCSE) [IEEE/ACM 2003; Wang, 2005h]:

- "Engineers carry out a task by making a series of decisions, carefully evaluating options, and choosing an approach at each decision-point that is appropriate for the current task in the current context. Appropriateness can be judged by tradeoff analysis, which balances costs against benefits.
- "Engineers measure things, and when appropriate, work quantitatively; they calibrate and validate their measurements; and they use approximations based on experience and empirical data.
- "Engineers emphasize the use of a disciplined process when creating a design.
- "Engineers can have multiple roles: research, development, design, production, testing, construction, operations, management, and others such as sales, consulting, and teaching.

- "Engineers use tools to apply process systematically. Therefore, the choice and use of appropriate tools are key to engineering.
- "Engineering disciplines advance by the development and validation of principles, standards, and best practices.
- "Engineers reuse designs and design artifacts."

Many scientists perceived that software engineering is not only a subset of computer science [Parnas, 1995/96/97/98; Hartmanis, 1994], because it is a much broader field than computer science, which encompasses denotational mathematics, engineering foundations, cognitive informatics foundations, system science foundations, and organizational foundations. Further, the objects under study in software engineering are not only computers and computational data objects, but also behavioral information, knowledge representation, and machine and natural intelligence, which are more fundamental at the root of human knowledge than computing [Wang, 2004b/04c/05a/06c/06g/06h/06i]. Further details will be discussed throughout this book, particularly in Chapter 5.

Therefore, more rigorously, the definition of software engineering can be revised as follows.

Definition 1.6 *Software engineering* is an engineering discipline that studies the nature of software, approaches and methodologies of large-scale software development, and the theories and laws behind software behaviors and software engineering practices.

The perception as presented in Definition 1.6 on software engineering may be treated as a guideline in the search of software engineering theories and foundations towards a matured engineering discipline as adopted in this book.

1.2.4 HIERARCHY OF ABSTRACTION AND DESCRIPTIVITY IN SOFTWARE ENGINEERING

It is recognized that software engineering seems using low-tech means to deal with high-tech problems [Wang, 2005g]. Consider that the accuracy of micrometer technologies is at the level of 10^{-6}m, while that of nanometer technologies is at the level of 10^{-9}m. If one asks whether a microtechnique may be used to denote, measure, or process a nanotechnical product, mechanical or electronic engineers will tell that is impossible! However, software engineers are still attempting to do so by using inadequate means and tools to deal with more complicated software systems. Various graphical blocks and arrows are proposed to denote more intricate system architectures and dynamic behaviors of software systems. At the mean time, academics and practitioners in software engineering seem to be use to the practice. It is believed that both the inadequate techniques and the undoubted attitude form the profound problems of software engineering and are the fundamental reasons of almost all difficulties and failures in industrial practice in software engineering.

This observation leads to the following question: What kind of descriptive means does software engineering need? The answer is abstraction according to Theorem 1.2. *Abstraction* is a powerful and fundamental mental function of human beings that most of the higher cognitive processes of the brain are relied on [Wang et al., 2006; Wang, 2007h]. Abstraction is also the most fundamental principle of mathematics [Lipschutz, 1964]. Abstract objects exist only in the brain as a concept or idea but not exist in the real world as a physical or concrete entity. However, the sources of abstract objects are reflections of real world entities, phenomena, and their relations. Abstraction is a powerful key to reduce complexity in creative work such as software engineering. It is a software engineering principle for eliciting essential properties of a set of objects while omitting inessential details of them.

1.2.4.1 The Hierarchical Abstraction Model of System Descriptivity (HAMSD)

According to the IME model as stated in Theorem 1.2, there are two categories of objects under studies in science and engineering known as the *concrete* entities in the real world and the *abstract* objects in the information world. In the latter, an important part of the abstract objects are human and/or system behaviors, which are planned or executed actions onto the real-world entities and abstract objects.

In Section 1.2.1 software is described as both the abstraction of real-world objects and their relations in its architectural aspect, and the abstraction of the executable behaviors that the system is expected. A hierarchical abstraction model of system descriptivity of human knowledge can be defined below and illustrated in Fig. 1.3.

Definition 1.7 The *Hierarchical Abstraction Model of System Descriptivity* (HAMSD) states that the abstract levels of cognitive information of both the objects and their behaviors can be divided into five levels such as those of *analogue objects, diagrams, natural languages, professional notations,* and *mathematics.*

Because software is the abstraction of both real-world objects and their relations, and the expected and executed behaviors of a system that, in turn, are real or expected human behaviors (see Theorem 3.4), the means and nature of software obeys the same HAMSD model as shown in Fig. 1.3.

The abstract levels of objects or knowledge modeled in the HAMSD model can be explained in Table 1.3. According to Fig. 1.3 and Table 1.3, there are five abstract levels and related descriptive means. The higher the abstraction level of an object, the more complex the description means. In software engineering the objects under study are system and interactive behaviors in the abstract world, rather than concrete entities in the real world, which is located at abstraction levels L3 to L5. This is a fundamental difference between software engineering and other engineering disciplines. This can be formally described in the 3rd law of software engineering.

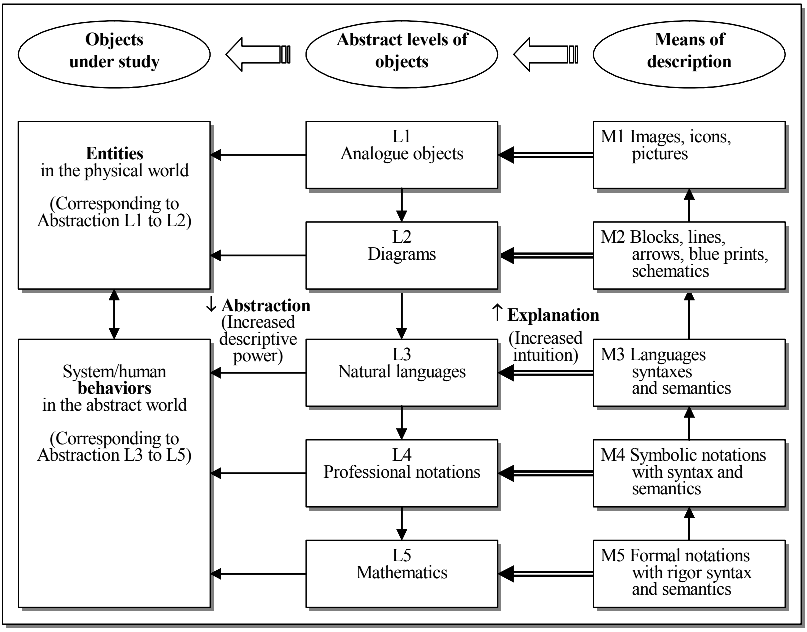

Objects under study
Abstract levels of objects
Means of description
Entities in the physical world
(Corresponding to Abstraction L1 to L2)
System/human behaviors in the abstract world
(Corresponding to Abstraction L3 to L5)
L1 Analogue objects
L2 Diagrams
L3 Natural languages
L4 Professional notations
L5 Mathematics
↓ Abstraction (Increased descriptive power)
↑ Explanation (Increased intuition)
M1 Images, icons, pictures
M2 Blocks, lines, arrows, blue prints, schematics
M3 Languages syntaxes and semantics
M4 Symbolic notations with syntax and semantics
M5 Formal notations with rigor syntax and semantics

Figure 1.3 The hierarchical abstraction model of system descriptivity (HAMSD) for software engineering

Table 1.3
Abstract Levels of Knowledge and Cognitive Information

Level	Category	Description
L1	Analogue objects	Real-world entities, empirical artifacts
L2	Diagrams	Geometric shapes, abstract entities, relations
L3	Natural languages	Empirical methods, heuristic rules
L4	Professional notations	Special notations, rigorous languages, formal methods
L5	Mathematics (philosophy)	High-level abstraction of objects, attributes, and their relations and rules, particularly those that are time and space independent

The 3rd Law of Software Engineering

Theorem 1.3 The *abstract objects under study* state that the *nature of software* stems from intangibility of the abstract objects under study, intricate inner connections of software systems, adaptive interactions to external events and environments, and the cognitive complexity to explicitly describe them.

There are two approaches for system description as shown in Fig. 1.3 known as *abstraction* and *explanation*. The former enables system analysts to increase the descriptive power in terms of expressiveness, preciseness, and rigorous; while the latter helps users to increase the intuitiveness of understanding and comprehension because the means of description is much closer to the real world images and analogue objects directly acquired by the sensors of the brain. Detailed discussion of the cognitive foundations of notation systems may be referred to Chapter 9.

1.2.4.2 Software Engineering Practice: Can Microtech be Used to Denote Nanotech?

Contrary to physical, mechanical, and geometrical designs in the concrete world, software design is carried out in the abstract world where only things that can be embodied are blocks, lines, or arrows, which show connections or relations between architectural and/or behavioral components at different levels. In other words, software behaviors, particularly the dynamic aspect, are *inexpressible* by conventional means of diagrams.

For instance, geometry is the ultimate form of graphical reasoning and is more rigorous than any software block diagrams and class diagrams. However, even for that, algebraic geometry had to be developed to deal with more complicated structures in dynamics in order to enable more powerful inferences in geometrical analyses.

Corollary 1.2 The *expressive power* of icons and diagrams are inadequate in software engineering because they make software design and specifications vague.

To prove Corollary 1.2, one may try to read a cartoon after erasing all dialogues and explanations. It is obvious that one cannot obtain too much accurate information from it. It may be worse that different persons may perceive different information and meanings from the same picture. This is totally unacceptable in engineering design.

Why may visual means be used adequately in system designs in mechanical engineering and civil engineering, but not adequately in software engineering? It is because the objects under study of the former are physically and geographically located at Level 1 according to the HAMSD model, while those of the latter are abstract objects located at Level 3 or above in HAMSD.

It is recognized that architectures of software are complex interrelated objects with functional variables and constraints. Behaviors of software are embedded relational processes [Wang, 2006h]. These types of abstract and complicated entities may only be expressed without implication by professional notation systems at Level 4 or mathematical means at Level 5 according to the HAMSD model, because only more abstract and precise means are powerful enough to express an object at a lower level of abstraction. This leads to the 4th law of software engineering.

The 4th Law of Software Engineering

Theorem 1.4 The *explicit descriptivity* states that only a *higher-level abstract, precise, and rigorous means* is adequate to express an object at a given level of abstraction, where denotational mathematics is the top-level abstraction means.

According to Theorem 1.4, software architectures and behaviors cannot be explicitly expressed by diagrams, because the abstraction level of the latter is lower than the former. However, diagrams can be used to express and denote physical architectures and designs, because the abstraction levels of the physical objects are lower than the means. This explains that, although visualization is a powerful means to form a mental image and conceptual representation of a design by means of diagrams and pictures in other engineering disciplines, it is inadequate in software engineering.

Corollary 1.3 *Symbolic notations and mathematics* are the key means for expressing and embodying software behaviors, because they are at higher level abstraction and therefore with more powerful descriptivity.

The descriptive power of formal notation systems in software engineering lies in its adequacy for describing abstract objects, their relations, behaviors, and for enabling rigorous inferences based on formally defined composing rules in terms of formal syntaxes and semantics.

Although diagrams are widely used to represent abstract physical designs, they are not suitable for expressing and embodying more abstract and intricately interconnected entities like software systems, because using graphical means in system specification and refinement is just like using conventional microtechniques to denote and measure nanotechnologies.

Therefore, in software engineering, graphical means may be used to describe rough conceptual models of software systems for assisting human comprehension, but not for the precise specifications of system architectures and behaviors in system modeling and execution. Lessons learnt from all the failures in the last four decades indicate, no matter how convenient it might appear to be, intuitive comprehension should not be an excuse to stick at any graphical and visual means for rigorously describing complicated software architectures and behaviors. This is what Theorems 1.3 and 1.4 reveal and suggest. The fade away of many historical graphical means such as flow charts and state diagrams is evidence to support the HAMSD theory. Further explanation of the need of formal inference means in software engineering will be described in Section 3.3 on formal inference methodologies.

1.3 Basic Constraints of Software Engineering

Software engineering is a unique and probably the most complicated engineering discipline that has ever been faced by mankind. The constraints of software engineering are inherent due to its intangibility, complexity, diversity, and human dependency. The study of the fundamental constraints of software engineering is helpful to: a) Understand the fundamental problems in software engineering, b) Guide the development of software engineering theories and methodologies, and c) Evaluate newly proposed software engineering theories, principles, and techniques.

1.3.1 THE SOFTWARE ENGINEERING CONSTRAINT MODEL

One of the discoveries on the nature of software engineering reveals that software engineering is characterized as an organizational challenge supplemented to the cognitive and resources challenges [Dijkstra, 1965; Wang, 2004b/06a]. A comprehensive set of 14 basic constraints of software engineering are identified, which can be classified into three categories known as follows:

- The *cognitive* constraints: such as intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures.
- The *organizational* constraints: such as time dependency, conservative productivity, and labor-time interlock.
- The *resources* constraints: such as costs, human dependency, and hardware dependency.

The 5th Law of Software Engineering

Theorem 1.5 The *basic constraints of software engineering* state that software engineering faces the *cognitive, organizational,* and *resources constraints*.

The relationships between the three categories of basic constraints of software engineering can be described by the Software Engineering Constraint Model (SECM) as shown in Fig. 1.4. In SECM, the first eight constraints can be perceived as cognitive and technical constraints in software engineering, whilst the last six are considered as business constraints on organizational limitations or resources scarcity in software engineering.

In the following subsections, we examine each of the basic constraints of software engineering according to the classifications of the SECM model.

1.3.2 COGNITIVE CONSTRAINTS OF SOFTWARE ENGINEERING

Definition 1.8 The *cognitive constraints* of software engineering are a set of innate cognitive attributes of software and the nature of the problems in software engineering that create the intricate relations of software objects and make software engineering inheritably difficult.

All cognitive constraints of software engineering stem from intangibility, intricate inner connections, and the cognitive difficulty of software and their systems. The following subsections describe each of the eight cognitive constraints of software engineering.

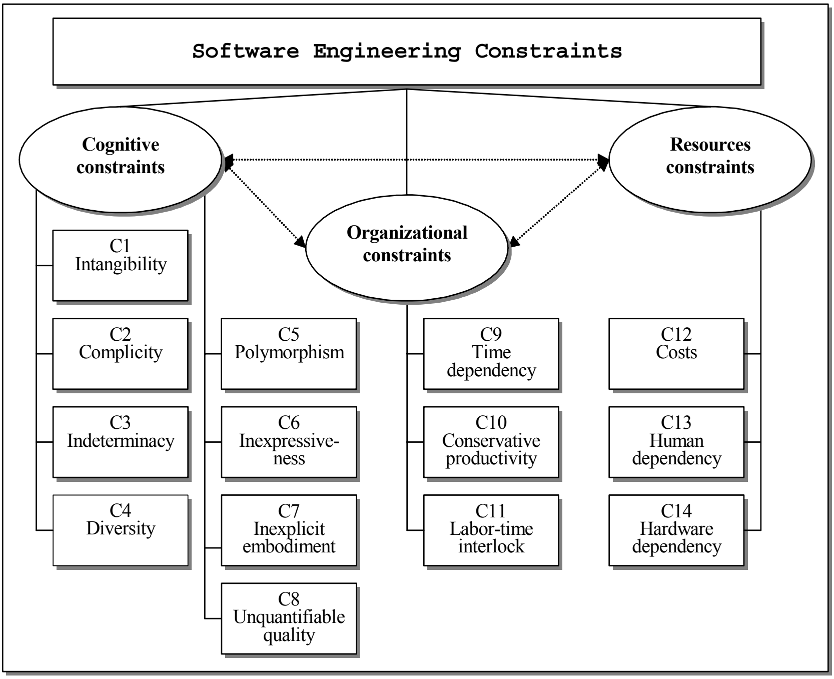


Figure 1.4 The Software Engineering Constraint Model (SECM)

1.3.2.1 Intangibility

Definition 1.9 *Intangibility* (C1) is a basic constraint of software engineering that states software is an abstract artifact which is not constituted by physical objects or presence, and is difficult to be defined or expressed.

The intangibility of software refers to all aspects of software and its development. That is, none of the software engineering processes, such as problem representation, requirements description, design, and implementation, is tangible.

1.3.2.2 Complexity

Definition 1.10 *Complexity* (C2) is a basic constraint of software engineering that states software is innately complex and its intricate internal connections and external couplings make it extremely difficult to be expressed or cognized.

The complexity of software refers to the complexities of its architectures, behaviors, and environments. The *architectural complexity* is the innate complexity of a software system with its data objects and their external/internal representations. The *behavioral complexity* is the complexity of a software system with its processes and their inputs/outputs. The *environmental complexity* is the complexity of a software system with its platform, related interacting processes, and users.

The most unique feature of software complexity is its intricate interconnection among components, functions, operations, and data objects. A small change in one place may result in multiple and unpredictable consequences in other places. This type of problem propagation due to intricate interconnection and coupling is a major challenge for system architects, programmers, and managers.

The integration of a large-scale software system may easily result in a situation where no single person in the team may understand the system. The project leader and system architect may lack the knowledge of sufficient details about the implementation of the system, whilst the programmers may lack the knowledge of global view that treats the system as a whole with the interfaces to other subsystems and components. This is a great challenging and critical phase to human comprehension capability that often results in major failures after almost all resources have been exhausted in large-scale software projects.

In Chapters 9 and 10 we will introduce the concept of cognitive complexity of software, which provides a quantitative measure that reveals the cognitive functional complexity of software systems.

1.3.2.3 Indeterminacy

Definition 1.11 *Indeterminacy* (C3) is a basic constraint of software engineering that states the events, behaviors, or their sequence of occurrence in a software system are not fully determinable on the basis of a given algorithm during design time. Instead, some of them may only be determinable until run-time.

The indeterminacy constraint indicates that, in general, a large portion of software behaviors and/or their sequence of occurrence are unpredictable at design time or compile time. Although the behavior space and all possible events are determinable during design time, the order of events and the behaviors triggered by the chain of events will be greatly varying at run-time. Therefore, indeterminacy makes software design, implementation, and testing extremely difficult, because it results in an extremely large behavior space for the given software system and its complete verification and through testing are impossible sometimes.

Dijkstra discussed the special case of indeterminacy in automata where a given even to a finite state machine in a context may trigger no action because the machine cannot decide explicitly which action should be executed based on the given input and current state of the machine [Dijkstra, 1968b/1975]. This kind of phenomena occurs in operating system, agent systems, complier design, and real-time systems, where additional information or an arbitrary decision needs to be adopted in the machine.

1.3.2.4 Diversity

Definition 1.12 *Diversity* (C4) is a basic constraint of software engineering that states the great variety of software in types, styles, architectures, behaviors, platforms, application domains, implementation techniques, usability, reliability, and quality.

The characteristic of software domain dependency dominates the cognitive complexity of software engineering and the knowledge requirement for architects and programmers who design and implement a software system.

The diversity of software also refers to its types. A wide variety of software systems can be classified into the following categories: system software, tools (compilers, code generators, communication/networking software, database management systems, and test software), and application software. The latter can be further categorized into transaction processing software, distributed software, real-time software, databases, and web-based software.

1.3.2.5 Polymorphism

Definition 1.13 *Polymorphism* (C5) is a basic constraint of software engineering that states the approaches and styles of both software design and implementation are multifaceted and polyglottic.

Definition 1.13 indicates that the possible solution space of software engineering can be very large because both design and implementation have a great many options as shown in Fig. 1.5. According to the problem solving theory in cognitive informatics, software design and development is an open-end problem, which is similar to a creation process, where both possible solutions and paths that lead to one of the solutions are unknown and highly optional.

As that of the polysolvability for design, the polymorphism of software implementation refers to the cognitive phenomenon that approaches to implement a given design are not necessarily single. Many factors influence the solution space such as programming languages, target machine languages, coding styles, data models, and memory allocations. Any change among these factors may result in a different implementation of a software system.

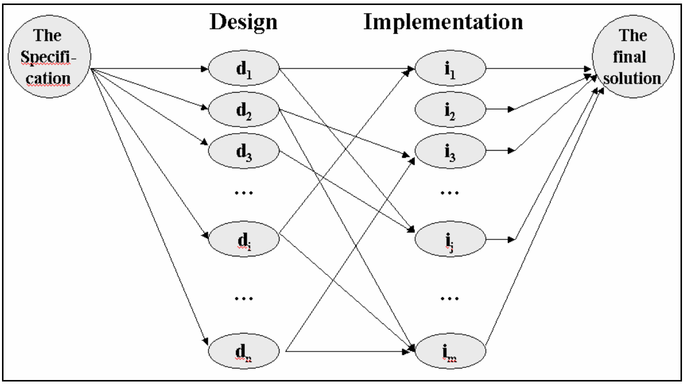


Figure 1.5 Polymorphism of the software solution space for a given problem

The 1st Principle of Software Engineering

Theorem 1.6 *Polymorphous solutions* state that the solution space *SS* of software engineering for a given problem is a product of the number of possible design options N_d and the number of possible implementation options N_i, i.e.:

$$SS = N_d \bullet N_i \tag{1.3}$$

On the basis of Theorem 1.6, a basic engineering principle can be derived as follows.

Corollary 1.4 It is hard to prove technically and/or economically a certain software system is the optimal solution rather than a sound one constrained by the size of the solution space. This is known as the *no number one principle* in engineering.

The polymorphic characteristic of the solution space of software engineering contributes greatly to the complexity of both theories and practices of software engineering.

1.3.2.6 Inexpressiveness

Software system requirements and specifications need to be essentially expressed in three aspects known as the *architecture*, *static behaviors*, and *dynamic behaviors* of the system.

Definition 1.14 *Inexpressiveness* (C6) is a basic constraint of software engineering that states software architectures and behaviors are inherently difficult to be expressed, modeled, represented, and quantified both formally and rigorously.

As discussed in Section 1.2, software represents a set of instructive behavioral information. Unless the behaviors and the underpinning architecture can be expressed rigorously and explicitly, no developer and machine may understand the requirement correctly and completely. Therefore, a new type of denotational mathematics is needed for system specification and refinement, which will be introduced in Chapter 4.

In addition, a specification of a software system is innately a moving target. No practical methodology may suggest customers to fix and freeze their requirements in order to get the system implemented.

1.3.2.7 Inexplicit Embodiment

Because software is intangible, the only way to make it embodied is to adopt expressive means such as formal notations, programming languages, and special diagrams.

Definition 1.15 *Inexplicit embodiment* (C7) is a basic constraint of software engineering that states architectures and behaviors of software systems should be explicitly described by coherent symbolic notations in order to be processed and executed by computers.

Any notation or diagram that cannot explicitly describe the architecture and behaviors of software systems, or that highly depends on human interpretation or imagination for implied instructions, is inadequate. According to the explicit criterion, diagram-based techniques may be useful for describing conceptual models of software systems particularly for nonprofessionals, but it is unlikely to be an expressive and rigorous basis for future automatic code generation systems, because too much design and implementation information are implied rather than explicitly expressed. Machines will be capable to carry out translations or compilations between explicit specifications and code in order to improve software productivity. However, no machine may help to extend inadequate system specifications or to comprehend inexplicit system designs implied in the software architectural and behavioral information.

A denotational mathematical means for describing software engineering work products in the entire lifecycle will be introduced in Chapter 4.

1.3.2.8 Unquantifiable Quality Measures

Determined by the complexity, diversity, and polymorphism constraints discussed earlier, the quality of software is a multifaceted entity and some facets of it are application specific.

Definition 1.16 *Unquantifiable quality measures* (C8) are a basic constraint of software engineering that states the model of software quality has intricate facets and is difficult to be quantitatively modeled and measured.

Software quality can be perceived from a relative point of view as the conformity of a software system to its specifications (design models). Therefore, software quality is inversely proportional to the differences between the behaviours and performance of a software system and those required in the specifications. However, many quality attributes of software, such as design quality, usability, implementation efficiency, and reliability, cannot be quantified, thus immeasurable.

A basic quality principle is "no measurement, no quality control." The factor that it is impossible to measure all quality attributes of a large-scale software system indicates that we are not completely in control of the development of such systems. Some qualitative or informal validation and evaluation techniques, such as review and prototyping [Boehm et al., 1984; Arnowitz et al., 2006], are adopted in software engineering. Practitioners and users seem to be used to this situation. Therefore, measurement theories and methodologies for software systems had never been a central focus in software engineering, particularly because of its inherent difficulty in this area [Wang, 2003f].

1.3.3 ORGANIZATIONAL CONSTRAINTS OF SOFTWARE ENGINEERING

Definition 1.17 The *organizational constraints* of software engineering are a set of coordinative and managerial requirements for software engineering that enables cooperative work to be efficiently carried out among a group of software engineers with different roles.

There are various organizational constraints for software engineering. The fundamental ones identified in software engineering are such as time dependency, conservative productivity, and labor-time interlock, which will be described in the following subsections.

1.3.3.1 Time Dependency

Definition 1.18 *Time dependency* (C9) is a basic constraint of software engineering that states almost all organizational issues in software engineering, such as software development scheduling, business goal of time to market, and labor allocation, are dependent on time.

Although the development time of a certain software system is incompressible in software engineering, it is interchangeable with labor under given conditions. The interchangeability between time and labor is constrained by the coordinative work organization laws of software engineering (Theorems 8.4 through 8.11), which will be presented in Section 8.5.

1.3.3.2 Conservative Productivity

According to the cognitive model of internal information representation in the brain (Section 9.4), all human intelligent and creative work are internally grown by means of synaptic neural connections rather than externally composed. Therefore, there are natural constraints for programming productivity and development time dependent on the complexity of problems. Also, almost all human cognitive processes, such as abstraction, creation, problem solving, learning, and comprehension, are dependent on individuals' cognition capability.

Definition 1.19 *Conservative productivity* (C10) is a basic constraint of software engineering that states abstract artifacts and their relations in system designs need to be represented physiologically in the brain via growing synaptic connections, which is constrained by natural laws and its speed is not consciously controllable.

The fact that before any program is composed, an internal abstract model must be created inside the brain [Wang, 2007g; Wang and Wang, 2006] reveals the most fundamental constraint of software engineering as stated below.

The 6th Law of Software Engineering

Theorem 1.7 *Conservative productivity* states that software productivity is physiologically constrained by the growing speed of synaptic connections inside the brain, because before any creative artifact is generated externally, it must be created and represented physiologically inside the brain by the synaptic connections.

Theorem 1.7 is also supported by *the 24-hour law of memory establishment* as presented in Theorem 9.11. According to the statistics of several sources [Boehm, 1987; Dale and Zee, 1992; Jones, 1981/1986; Livermore, 2005], the average productivity of software development was about 1,300 LOC/person-year in the 1970s, 2,500 LOC/person-year in the 1980s, and 3,000 LOC/person-year in the 1990s where management, quality assurance, and supporting activities are considered. It is obvious that the productivity in software engineering has not been increased remarkably in the last three decades, independent of the fast advances of hardware and programming languages! In other words, no matter what kinds of programming languages are used, as long as they are for human programming, there is no difference in principle.

Corollary 1.5 It is very hard to dramatically improve programmers' productivity of software development, unless automatic tools are adopted for code generation.

This assertion is equivalent to the answers for the following questions: Did you ever know a writer who is productive because he/she writes in a specific language? Would typing speed predominantly determine a writer's productivity?

Productivity of software development is the key among all the cognitive, organizational, and resources constraints in software engineering. The other constraints may be overcome as a result of the improvement of software engineering productivity. Therefore, the major approach to improve software development productivity is to explicitly express software architectures and behaviors, in order to allow automatic tools to seamlessly generate code based on the explicit specifications. Key approaches and theories supporting them will be discussed throughout this book.

1.3.3.3 Labor-Time Interlock

Definition 1.20 *Labor-time interlock* (C11) is a basic constraint of software engineering that states the nature of software project organization is dominated by the extremely high interpersonal coordination rate, which prevents the workload (effort) from free decomposition into a sum of products of arbitrary amount of labor and periods of time.

The empirical observation that labor and time may be interchangeable in conventional engineering disciplines and everyday life is not freely applicable in software engineering, because the much higher rate of requirement for human interaction and coordination in software engineering, up to 70% overhead, dramatically changes the nature of software project organization [McCue, 1978; Wang, 2007d].

The theory and skill of labor-time allocation will determine the outcome of a software engineering project. Inappropriate allocation of labor and time will result in a dramatic increase of a project's workload or lead to a failure of the project. Detailed explanation will be provided in the coordinative work organization theory in Chapters 8, 11, and 13.

1.3.4 RESOURCES CONSTRAINTS OF SOFTWARE ENGINEERING

Definition 1.21 The *resources constraints* of software engineering are referred to the development costs and budgets, human resources, and the supporting and operating platforms of hardware.

The resources constraints of costs, human dependency, and hardware dependency will be described in the following subsections.

1.3.4.1 Costs

Definition 1.22 *Costs* (C12) are a basic constraint of software engineering that states software engineering costs are incurred from both necessary and futility costs, and from both development and maintenance costs.

Software engineering costs are incurred from the contributions of both the necessary development costs and the costs sank into the black hole of inappropriate project organization. According to Theorems 8.4 through 8.11 developed in Section 8.5, inappropriate software engineering organization may easily increase the workload and costs of a certain project as high as ten times as it should normally be. This is the major reason why more than half of software engineering projects had failed in the history. Detailed cost models of software engineering will be developed in Chapter 12 on economics foundations of software engineering.

Software systems have cost too much to be built. It is even more costly to be maintained. There is a trend in software engineering that in a software development organization the costs spent on legacy software system maintenance frequently exceeds those spent on developing new systems at a given point of time. This phenomenon leads to the discovery of *software maintenance crisis* [Wang, 2005d] as described in Sections 12.6.5 and 14.3.3.

1.3.4.2 Human Dependency

Definition 1.23 *Human dependency* (C13) is a basic constraint of software engineering that states all software engineering activities and processes are human-based and constrained by basic human traits, cognitive and creative capabilities, as well as motivations and attitudes.

All processes of software development in software engineering, such as design, implementation, and maintenance, rely on human cognitive and creative power, especially when the work products and objects under study are intangible and complicated. Because software is intangible, almost all processes of software engineering are conducted in the abstract world, where highly capable cognitive power is required for both software engineers and managers.

1.3.4.3 Hardware Dependency

Definition 1.24 *Hardware dependency* (C14) is a basic constraint of software engineering that states software behaviors and functionality can only be embodied via the computing platform and related interactive I/O devices.

The above constraint indicates that software relies closely on hardware and it cannot be functioning without a hardware platform. The platform and media dependency is a common property of information. Software as a special type of behavioral information shares the same property. The fact that both hardware and software can be described by Boolean logic and denotational mathematics shows the equivalence of hardware and software.

Therefore, an excellent software engineer needs not only software engineering knowledge, but also knowledge about computers, networks, and interface devices.

Based on the discussion of this section, it can be concluded that software engineering theories and practices face a set of fundamental cognitive, organizational, and resources constraints. Therefore, the problems and difficulties in software engineering are inherent; behind them there are a set of cognitive, informatics, technical, systematic, managerial, and social reasons, which are the driving force of all the constraints in software engineering. Each of the multidisciplinary reasons that constrains software engineering will be addressed in individual chapters throughout this book. The historical pursuits of effective theories and technologies for dealing with the fundamental constraints in software engineering will be reviewed in the next section.

1.4 Approaches to Software Engineering

Various approaches have been sought in the history of software engineering in order to deal with the fundamental problems and constraints as identified in the previous sections. The methodologies and approaches can be summarized as shown in Table 1.4, which cover programming methodologies, software development models, automated software engineering, formal methods, software engineering processes, and theoretical foundations.

In Table 1.4, H, M, and L represent a high, medium, or low coverage of software engineering problems, respectively, by a given approach. The six software engineering approaches listed in Table 1.4 can be categorized into theoretical and empirical ones. The former are mathematics-based methodologies, such as formal methods and the theoretical foundations; and the latter are principles- and best-practice-based that encompasses the rest of the approaches in the list.

Table 1.4
Domain Coverage of the Approaches to Software Engineering

No.	Approach	Coverage of SE Problems			
		Theoretical	Technical	Organization	Management
1	Programming methodologies	L	H	L	L
2	Software development models	L	H	M	L
3	Automated software engineering environments	M	H	L	L
4	Formal methods	H	H	L	L
5	Software engineering processes	L	H	H	H
6	Theoretical foundations	H	H	H	H

It can be seen that the traditional approaches to software engineering, such as programming methodologies, software development models, and automated software engineering, are mainly technology oriented. They generally lack the capability to address the theoretical, organizational, and managerial problems in software engineering. The relatively newer approaches to software engineering, such as formal methods, software engineering processes, and theoretical foundations, however, encompass a wider and higher portion of software engineering problems, particularly the theoretical foundation approach towards a matured discipline of software engineering that fully covers every facet of the inherent problems.

The following subsections provide a brief description for each of the approaches to software engineering.

1.4.1 PROGRAMMING METHODOLOGIES

Programming methodologies are a set of the earliest technologies for software engineering. Useful principles were proposed such as abstraction, information hiding, functional decomposition, modularization, and reusability.

In tracing the history of programming methodologies, it can be seen that functional decomposition has been adopted in programming since the 1950s [McDermid, 1991]. In the 1970s the most significant progress in programming methodologies was structured programming [Hoare, 1972; Dijkstra, 1965/68/72; Knuth, 1974] and Abstract Data Types (ADTs) [Liskov and Zilles, 1974; Parnas et al., 1976]. These methods are still useful in programming and software system designs. Since the 1980s Object-Oriented Programming (OOP) [Stroustrup, 1982/86; Snyder, 1987] has been broadly adopted. Object-Oriented technologies have inherited the merits of structured programming and ADTs, and have represented them in well-organized mechanisms such as encapsulation, inheritance, reusability, and polymorphism. The most powerful feature of OOP is the supporting of software reuse by inheriting code and structural information at object and system levels. On the basis of OOP, component-based program composition and the availability of Commercial Off-The-Shelf (COTS) software components are the latest developments [Bayana, 2006; Wang et al, 1998a/99e/2000].

1.4.2 SOFTWARE DEVELOPMENT MODELS

Programming methodologies can be perceived as mainly oriented to the conceptual principles of software engineering. A set of more programmatic technologies developed in software engineering is known as the software development models, such as the *waterfall* [Royce, 1970], *prototype* [Boehm et al., 1984; Curtis et al., 1987], *spiral* [Boehm, 1988; Boehm and Bose, 1994], *V* [GMOD, 1992], *evolutionary* [Lehman, 1985; Gilb, 1988;

Gustavsson, 1989], and *incremental* [Parnas, 1979; Mills et al., 1980/87] models.

Supplementary to the above development models, a variety of detailed methods have been proposed for each phase of the development models. For instance, just for the software design phase, a number of design methods have been in existence [McDermid, 1991], typically *flowcharts, data flow diagrams, Nassi-Shneiderman charts, Program Description Languages* (PDLs), *entity-relationship diagrams, Yourdon methods,* and *Jackson system development*. Of course, some of these methods may cover multiple phases in software development.

The software development model approach attempts to provide a set of guidelines for the design and implementation of software at system and module levels. However, this approach has been focused on technical aspects of software development lifecycles. Organizational and managerial methodologies and processes have not been covered. Detailed descriptions and applications of existing software development models may be referred to the classic software engineering books [McDermid, 1991; Pressman, 1992; and Sommerville, 1996].

1.4.3 AUTOMATED SOFTWARE ENGINEERING

In order to support programming methodologies and software development models, an automated software engineering approach has been sought through the adoption of integrated systems and Computer-Aided Software Engineering (CASE) tools. The applications of artificial intelligence, cognitive informatics, and knowledge engineering play important roles in this approach. The standardization of the Unified Modelling Language (UML) [Rumbaugh et al., 1998; OMG, 2005] and related tools such as Rational Rose [Quatrani, 1999], as well as Real-Time Process Algebra (RTPA) based code generators [Tan and Wang, 2006; Tan, Wang, and Ngolah, 2004a/04b/05/06; Ngolah, Wang, and Tan, 2005b/06], are some of the recent progress in automated software engineering. Cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b], agent techniques [Chorafas, 1998], and autonomic computing [IBM, 2001/06; Kephart and Chess, 2003; Murch, 2004; Wang, 2007f] may influence the techniques in this area.

The main technical difficulties in automating software development are requirement acquisition and specification, system architectural behavioral modeling, application domain knowledge representation, and implementation correctness proof. These have led to the development of the approaches of formal methods and theoretical foundations as described in the following subsections.

1.4.4 FORMAL METHODS

Formal methods are a set of mathematics and logic based notations and methodologies for software development [Hoare, 1978/85; Milner, 1980/89]. The logical, algebraic, and functional foundations of programming are studied in formal methods. A number of applications of formal methods in safety-critical system design and program correctness proof have been reported [Hayes, 1987; Schneider, 1989; Wang and Ngolah, 2003; Tan, Wang and Ngolah, 2005].

As structured programming and OOP solved many problems in software development in the 1970s and 1980s, formal methods attempt to dig deeply into the nature of programming and to provide new solutions for rigorous and correction-provable software development. Although knowledge about the nature of programming has been greatly improved by the studies of formal methods, only a few of them, such as Z [Spivey, 1988/92; Bowen et al., 1998; Derrick and Boiten, 2001], SDL [CCITT, 1988], CSP [Hoare, 1978/85], and Real-Time Process Algebra (RTPA) [Wang, 2002a/02b/03c/07h/07i; Wang and Gafurov, 2003; Wang and Ngolah, 2002/03; Wang and Zhang, 2003; Wang and Huang, 2005; Wang and Ruhe, 2007; Tan and Wang, 2003; Adewumi and Wang, 2004; Vu and Wang, 2004; Chiew and Wang, 2004], have been directly applied in real-world software engineering.

1.4.5 SOFTWARE ENGINEERING PROCESSES

In view of domain coverage it is recognized that the conventional approaches, methodologies, and tools that cover individual subdomains of software engineering are inadequate. Thus, it makes sense to think in terms of an overarching set of approaches for a suitable theoretical and practical infrastructure that accommodates both new demands and improvement on existing methodologies. An interesting way forward, which is capable of accommodating the full domain of modern software engineering, is that of the software engineering process.

The *software engineering process* is a set of sequential practices that are functionally coherent, repeatable, and reusable for software engineering organization, development, and management. It is usually referred to as the software process, or simply the process.

The software engineering process approach concerns systematical, organizational, and managerial infrastructures of software engineering. It is necessary to expand the horizons of software engineering in this way because of the rapidly increasing complexity and scale demanded by software products. The need to improve software quality is also a driving force for management in software engineering.

There are a number of process models developed such as the CMM [Humphrey, 1988/89/95; Paulk et al., 1991/93/95], ISO 12207 [ISO/IEC 12207, 1995], ISO 15504 [ISO/IEC 15504, 2000], and the Software Engineering Process Reference Model (SEPRM) [Wang and King, 2000a]. On the basis of the software engineering process technologies, an infrastructural methodology called Process-Based Software Engineering (PBSE) has been established [Wang and Bryant, 2002].

1.4.6 THEORETICAL FOUNDATIONS OF SOFTWARE ENGINEERING

A particular gap in the current curriculum of software engineering is the lack of a fundamental framework that provides students and practitioners for a set of overarching, durable, and multidisciplinary theories and foundations, in order to explain a great many complicated phenomena and problems of software engineering in terms of a core set of fundamental principles. To deal with the difficulties inherent in large-scale software development, the multidisciplinary foundations of software engineering are yet to be explored.

Along with the fast growth of the Internet and the Internet-based distributed programming environment in the 1990s, there has been evidence that the software engineering agenda has been driven by the industry and users. Technical innovations in software engineering have been a major force that drives software engineering trends, methodologies, and practice. However, all unsolved tough problems and the continuous high failure rate of projects in software engineering suggest the necessary need for seeking fundamental theories and systematically structures of software engineering.

One of the fundamental findings in software engineering is that its problems are not solely an empirical one rather than a theoretical one. That is, the same set of fundamental problems that could not have been overcome in the last four decades indicates the existence of fundamental constraints that need theoretical investigations to reveal the laws and principles behind all the problems. Definition 1.6 on software engineering expressed the theoretical view towards an engineering discipline with sound theories and rigorous organizational methodologies. It is also the main objective of this book towards the elicitation and establishment of a rigorous theoretical framework of software engineering.

1.5 Transdisciplinary Foundations of Software Engineering

Although software engineering has emerged as a branch in computer science, it is recognized that software engineering requires much broader and multidisciplinary foundations, particularly those that facilitate rigorous expression of notions and thought in system design, and those that enable optimal organization of creative and cooperative human work.

As discussed in Sections 1.1 and 1.2, the complexity, diversity, and transdisciplinary nature of software engineering sets forth an ideal testbed for existing theories and methodologies of a wide range of science and engineering disciplines, such as philosophy, mathematics, computing, linguistics, information science, cognitive informatics, system science, management science, economics, sociology, and engineering organization. From another angle, software engineering theories can also contribute significantly to those aforementioned disciplines. This forms the theme of this book towards a transdisciplinary and rigorous framework of software engineering foundations.

Albert Einstein believed that 'Problems that are created by our current level of thinking cannot be solved by that same level of thinking.' In other words, the inherent problems in software engineering may be solved by an transdisciplinary approach. This section examines closely related science and engineering disciplines to software engineering that may contribute to the transdisciplinary theories of software engineering. However, the coverage is by no means exhaustive. Therefore, readers are encouraged to seek additional disciplines that would shed light on the development of theories and methodologies of software engineering or that would take software engineering as a testbed to evaluate a specific theory or technique. With the support of multidisciplinary foundations and theories, software engineering education and practice may be carried out on a more solid basis.

1.5.1 PHILOSOPHICAL FOUNDATIONS

Philosophy is the common root of all sciences and the crystallization of fundamental knowledge of mankind in the pursuit of understanding and utilizing nature resources and rules. Philosophy is the top level human knowledge with highly generalized usability and extremely long durability.

Philosophical foundations of software engineering explore the classical thought of science philosophy on epistemology/cognition, holism/reductionism, positivism/empiricism, rationalism/causation, determinism/indeterminism in Chapter 3. Formal inference methodologies as an important branch of philosophy will be described with argumentation techniques, deductive, inductive, abductive, and analogical inferences. The nature of software will be studied on its inherent characteristics and properties of informatics, intelligence, mathematics, cognition, quality, and engineering applications. The nature of software engineering will be investigated on its cognitive properties, engineering characteristics, scope of domains, and the scope of its solution space.

1.5.2 MATHEMATICAL FOUNDATIONS

Mathematics is the basic means to model, describe, and document formal knowledge in any science and engineering disciplines. Mathematics enables rigorous reasoning and inferences be carried out on the basis of simple deductive rules, and the formally documented results are validated without exceptions. Therefore, the entire theory of software science and engineering in Chapter 4 is about mathematical models of software and denotational mathematics for software engineering processes.

Mathematical foundations for software engineering encompass classical mathematics such as set theory, relations, functions, propositional logic, and predicate logic. Complex mathematical entities in software engineering, such as processes, embedded relations, and systems, as well as a comprehensive set of mathematical laws of software, will be newly developed. Software behaviors are modeled by a three-dimensional (3-D) mathematical entity, and RTPA will be introduced as a software engineering notation system to deal with the 3-D behaviors and architectures of software systems. RTPA notations, type system, meta processes, and process relations will be presented, and the RTPA methodology for software system specification and refinement will be systematically described.

1.5.3 COMPUTING FOUNDATIONS

Computing theory is one of the most important and direct foundations of software engineering as presented in Chapter 5. However, classical computing theories may be treated and reinterpreted on focusing the needs for modeling and manipulating data objects, computing behaviors, and resources in software engineering.

The computing foundations for software engineering encompass the basic models and needs in computing, and the fundamental computation models, such as automata, Turing machines, von Neumann machines, and autonomic computing machines. Computing theories will be classified into three categories known as the modeling and manipulation of data objects, computing behaviors, and resources/processes. Based on this, programs and software will be modeled as a coordination of both computational behaviors and data objects in software engineering.

1.5.4 LINGUISTICS FOUNDATIONS

A language is a symbolic system for thought, self-expression, and communication. Although linguistics studies human or natural languages, the theories and foundations of linguistics and formal languages in software engineering are cross fertilized in many areas as presented in Chapter 6.

Linguistic foundations for software engineering encompass the fundamental theories of linguistics, the nature of languages, and their acquisition and applications. Classical linguistics will be extended to the theories of programming languages, which investigate formal language theories, formal semantics, and mathematical models of linguistics.

1.5.5 INFORMATION SCIENCE FOUNDATIONS

According to the IME model as given in Theorem 1.2, information is the third essence for modeling the abstract world and its interactions with the physical world. Information science, or *informatics*, studies the nature of information, its processing, and ways of transformation between information, matter, and energy. In the classical information theory, Shannon defined *information* as a probabilistic predication of message sending from a source. Conventional information theory focuses on information transmission rather than information itself. The contemporary informatics tends to regard information as entities of messages, rather than a probability predication of messages. The new perception is found better to explain the theories and practices in the IT and software industries.

The information science foundations for software engineering encompass classical and contemporary information theories in Chapter 7. In the former, Shannon's information and entropy will be discussed; in the latter, new mathematical model of information will be introduced, and the transition from machine informatics to cognitive informatics that focuses on human perception and processing of information will be described. A comprehensive set of informatics laws of software will be derived that reveal the nature of software. Applications of informatics in software engineering will be discussed on how behavioral information of software is expressed and processed in software engineering.

1.5.6 ENGINEERING FOUNDATIONS

Engineering is a powerful concept and methodology emerged in the industrial revolutions during the 18th and 19th centuries. The engineering foundations of software engineering study generic engineering approaches and basic engineering principles that are commonly shared by all engineering disciplines in Chapter 8.

Engineering principles for software development and organization will be systematically sought. The theory of optimal software engineering organization will be developed based on the investigation on team coordination and the transformability between labor and time. Empirical software engineering foundations will be explored on case studies, experiments, trials, benchmarking, and standardization in software engineering.

1.5.7 COGNITIVE INFORMATICS FOUNDATIONS

Cognitive informatics is a discipline that studies the internal information processing mechanisms of the brain and their engineering applications. Information in cognitive informatics is defined as abstract artifacts and their relations that can be elicited, modeled, represented, stored, and processed by human brains. One of the most interesting findings in cognitive informatics is that so many science and engineering disciplines, such as informatics, computing, software engineering, and cognitive sciences, share a common root problem – how the natural intelligence processes information.

Cognitive informatics foundations for software engineering encompass neurophysiology of cognition and cognitive foundations of the brain and natural intelligence in Chapter 9. A Layered Reference Model of the Brain (LRMB) will be developed. The mechanism of internal information representation will be formally described by the Object-Attribute-Relation (OAR) model. Applications of cognitive informatics in software engineering will be focused on cognitive laws of software engineering, software comprehension, and the measurement of cognitive complexity of software.

1.5.8 SYSTEM SCIENCE FOUNDATIONS

A system is a collection of coherent and interactive entities that has stable functions and clear boundary with its external environment. System science and engineering study the most complicated objects and phenomena in the physical, abstract, and social worlds, namely systems, across all science and engineering disciplines. A system can be treated as an extended mathematical entity constrained by certain mathematical laws. Systems are a powerful concept for describing a closure of a number of components and their relations and behaviors, because any given compound entity can be modeled as a system or a subsystem of another system.

System science foundations for software engineering encompass system philosophies, principles, and properties in Chapter 10. An extension of mathematics to deal with abstract systems, known as system algebra, will be newly developed [Wang, 2006d], which presents an algebraic treatment of system modeling, relations, and operations. System models of software and software engineering processes will be described on the basis of system algebra. A number of system engineering models for software engineering will be discussed.

1.5.9 MANAGEMENT SCIENCE FOUNDATIONS

Management is a coordination process that organizes activities and efforts of a group to achieve goals and results not possible by an individual. Management science is the discipline that studies organizational behaviors, executive decision making, and resource optimization on given internal and external constraints. Historically, software engineering has focused on programming methodologies, programming languages, and software development models. One of the critical areas to software engineering – organizational and management infrastructures – has been largely overlooked.

Management science foundations for software engineering encompass management principles, classical management thought, decision theories, and quality systems in Chapter 11. A set of organizational theorems and laws will be formally derived. A theoretical framework of decision theories will be developed with the mathematical models of decisions, the cognitive process of decision making, the formal decision strategies, the extended game theories, and the newly developed decision grid theory. Quality systems will be presented focusing on quality principles, quality assurance, and quality management systems. The emphases of applications of management science in software engineering will be put on SEPRM and the methodology of process-based software engineering.

1.5.10 ECONOMICS FOUNDATIONS

Economics is the study of how resources are used to produce and distribute commodities and how services are provided in society. Engineering economics is a branch of microeconomics dealing with engineering related economic decisions. Fundamental concepts and principles of economics will be presented in Chapter 12 covering economic models, and analyses. A mathematical model on Adam Smith's equilibrium between demands and supplies will be derived that explains the nature and mechanisms of the invisible hands rigorously.

Software engineering economics will be discussed on elements of software costs, project costs estimation and analysis, and project benefit-cost ratio analysis. The software legacy maintenance model will be developed based on the software engineering economic models, which leads to the discovery of the phenomenon known as software maintenance crisis.

1.5.11 SOCIOLOGY FOUNDATIONS

Sociology is a branch of science that studies the structure, organization, operation, and development of human societies. A society is a dynamic human system that is interacting not only among members of the society, but also between societies and the natural environment. Collective behaviors and how motivation and attitude may influence human productivity and decision making will be studied in Chapter 13. Optimal organization of groups and societies will be focused.

Sociology foundations for software engineering encompass social structures, norms, collective behaviors and social psychology of software engineering. A formal model of social organization will be developed based on a new mathematical model known as the *organization tree*. Models of socialization will be used to explain the historical evolution of human societies, which predict that information society will be the form of human societies following the postindustrial one driven by underpinning economic structures and basic human needs. Applications of sociology in software engineering will be explored in the areas of social organization, social environment, ergonomics, and human factors of software engineering.

1.6 The Architecture of this Book

An improved understanding of the theoretical foundations of software engineering is helpful to design appropriate curricula for software engineering education, and to provide students with a solid and well founded discipline of software engineering knowledge. In a field such as physics, its knowledge structure is well developed based on clear foundations. Physicists know what they can do and what they cannot. This offers a solid basis for judging innovative and emerging technologies. Software engineering requires a similar basis. It is anticipated that investigations into the theoretical foundations of software engineering will provide fundamental capabilities for students and practitioners of software engineering.

The architecture of this book is constructed as shown in Fig. 1.6. The book encompasses four parts on principles and constraints, theoretical foundations, organizational foundations, and perspectives on software science. Part I, *principles and constraints* of software engineering, models the basic constraints in software engineering and explores suitable measures dealing with them by a comprehensive set of software engineering principles. In Part II, *theoretical foundations* of software engineering are created on the basis of philosophy, mathematics, computing, linguistics, and information science. In Part III, *organizational foundations* of software engineering are elicited from those contributing disciplines such as engineering methodologies, cognitive informatics, system science, management science, economics, and sociology. Part IV, *perspectives on software science*, summarizes the theoretical framework of software engineering, and provides a prospect on the development of software science.

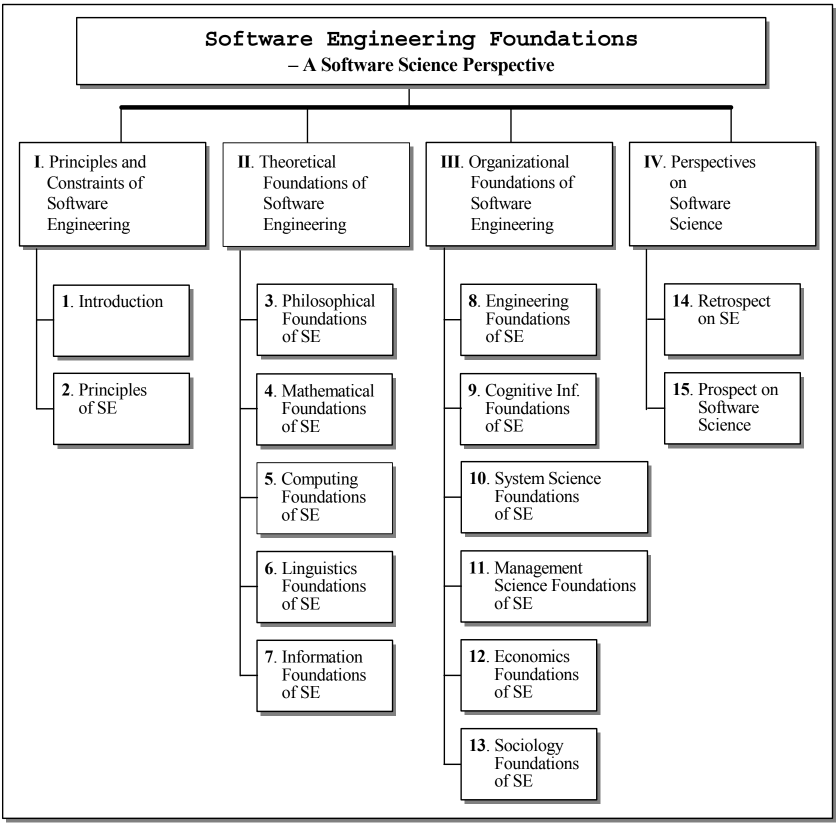

Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
1. Introduction
2. Principles of SE
3. Philosophical Foundations of SE
4. Mathematical Foundations of SE
5. Computing Foundations of SE
6. Linguistics Foundations of SE
7. Information Foundations of SE
8. Engineering Foundations of SE
9. Cognitive Inf. Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE
14. Retrospect on SE
15. Prospect on Software Science

Figure 1.6 Architecture of this book

Corresponding to the architecture of this book, the key subject areas of software engineering foundations are highlighted in Table 1.5. Throughout this book, new theories for both software engineering and related fields are developed, and more formal treatments of existing theories and empirical practice are implemented. Table 1.5 shows the bidirectional impact of this work on transdisciplinary investigation into the theoretical foundations of software engineering.

Table 1.5
Structure of this Book

Part	Chapter	Topic	Key Subject Areas
I. Principles and Constraints of Software Engineering	1	Introduction	• Basic concept of SE • Fundamental constraints of SE • Approaches to SE • Transdisciplinary foundations of SE
	2	Principles of Software Engineering	• Pursuits on principles of SE • A unified framework of SE principles • SE principles as measures for dealing with its constraints
II. Theoretical Foundations of Software Engineering	3	Philosophical Foundations	• Philosophy of science and engineering • Logical reasoning methodologies • The nature of software • The nature of SE • Murphy's laws for SE
	4	Mathematical Foundations	• Set theory • Mathematical logic • Denotational mathematics for SE • Real-Time Process Algebra (RTPA) • The RTPA methodology for software system description • RTPA: notations for SE
	5	Computing Foundations	• Basic computational models • Data object modeling and manipulation • Behavioral modeling and manipulation • Program modeling: coordination of computational behaviors and data objects • Resources and processes modeling and manipulation
	6	Linguistics Foundations	• Fundamentals of linguistics • Formal language theory • Syntax of programming languages • Semantics of programming languages

			• Semantics of RTPA • Linguistics perceptions on SE
	7	Information Science Foundations	• Classic information theory • Contemporary informatics • Informatics laws of software • Applications of informatics in SE
III. Organizational Foundations of Software Engineering	8	Engineering Foundations	• Generic engineering approaches • Basic engineering principles • Engineering principles for SE • Empirical SE • SE standardization
	9	Cognitive Informatics Foundations	• Principles • Cognitive informatics • Cognitive informatics models of the brain • Cognitive models of internal information presentation in the brain • Cognitive informatics for SE • Cognitive Informatics laws of SE • SE psychology • Software comprehension • SE skills and experience • Software agent systems • Cognitive Complexity of SE
	10	Systems Science Foundations	• Principles • System philosophy • Principles of system theories • System modeling • Properties of systems • System engineering for SE • System algebra • The system metaphor of software • System engineering for SE • Software system engineering models
	11	Management Science Foundations	• Principles • Classic management thought • Decision theories • Quality systems • SE management • Decision theories • Formal models of games • Decision grid theory • SE organization • The SE Process Reference Model (SEPRM)

			• Process-Based SE (PBSE)
	12	Economics Foundations	• Principles • Classical economic thought • Economic models • Dynamic values of money and assets • Economic analyses • SE economics • Elements of software costs • SE project costs estimation • Economic analyses for software projects • The software legacy maintenance cost model
	13	Sociology Foundations	• Principles • Principles of sociology • Social psychology • Theory of social organization • Sociology and SE • Organization trees • Social organization of SE • Ergonomics for SE • Human factors in SE
IV. Perspectives on Software Science	14	Retrospect on Software Engineering	• Infrastructure of SE • Software industry organization • Essential knowledge towards excellent software engineers • Impact of the theoretical foundations to SE
	15	Prospect on Software Engineering	• The formal knowledge system for SE • Software industry organization • A discipline of software • Impact of software science on computing • Epilogue

This book adopts a transdisciplinary approach to explore the theoretical foundations of software engineering. This work attempts to put together a number of multidisciplinary foundations for software engineering, such as philosophy, mathematics, computing, linguistics, informatics, engineering science, cognitive informatics, systems science, management science, economics, and sociology. It is a great curiosity to explore such transdisciplinary foundations for software engineering and the laws behind software and software engineering organization. It is also a great comfort to see that a set of extremely complicated phenomena and a wide variety of practices in software engineering can fit in a coherent and integrated framework of software science with overarching and durable theories and foundations.

1.7 Summary

Software is a special type of behavioral information of computing and a means of interaction between the information world and the physical world. The nature of software makes software engineering a unique discipline, which is innately the most complicated engineering branch that humans ever experienced, and inherently the most overarching transdisciplinary field in both theories and applications. These are also the reasons that set forth software engineering as an ideal testbed for existing theories and methodologies of a wide range of science and engineering disciplines from mathematics to cognitive informatics, and from management science to sociology.

Software engineering is an increasingly important discipline that studies the nature of software, approaches and methodologies of large-scale software development, and the laws behind software behaviors and software engineering practices.

The study of **the fundamental constraints** of software engineering is helpful for: a) Understanding fundamental problems in software engineering; b) Guiding the development of software engineering theories and methodologies; and (c) Evaluating software engineering theories, principles, and techniques.

To deal with the difficulties inherent in large-scale software development, **the multidisciplinary foundations** of software engineering are yet to be explored. This book adopts an interdisciplinary approach to explore the foundations of software engineering. This work attempts to put together a number of multidisciplinary foundations for software engineering, such as philosophy, computing, mathematics, informatics, systems science, management science, cognitive science, linguistics, and measurement.

This chapter has introduced the nature, the problem domain, the inherited constraints, and the transdisciplinary solutions towards software engineering. A set of basic concepts of software engineering has been presented. Fundamental problems and constraints of software engineering have been identified. The approaches to software engineering have been explored in the context of how the basic constraints of software engineering may be dealt with. The framework of multidisciplinary foundations of software engineering has been presented as a new approach towards software engineering. This chapter has also described the architecture of this book. As a result, the **problem domain and its nature** of software engineering, as well as the **fundamental approach** to software engineering, have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Introduction to Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 1. Introduction

- ■ Software Engineering as a Discipline
    - • The nature of software
        - *The mathematical metaphor*
        - *The product metaphor*
        - *The informatics metaphor*
    - • The nature of software engineering
    - • Status of software engineering as an engineering discipline
    - • Characteristics of Software Engineering
    - • Hierarchy of abstraction and descriptivity in software engineering

- ■ Fundamental Constraints of Software Engineering
    - • The software engineering constraint model
    - • Cognitive constraints of software engineering
        - *Intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures*
    - • Organizational constraints of software engineering
        - *Time dependency, conservative productivity, and labor-time interlock*
    - • Resources constraints of software engineering
        - *Costs, human dependency, and hardware dependency*

- ■ Approaches to Software Engineering
    - • Programming methodologies
    - • Software development models

  - Automated software engineering
  - Formal methods
  - Software engineering processes
  - Theoretical foundations of software engineering

- Transdisciplinary Foundations of Software Engineering
  - Engineering foundations
  - Philosophical foundations
  - Mathematical foundations
  - Computing foundations
  - Linguistics foundations
  - Information science foundations
  - Cognitive informatics foundations
  - Systems science foundations
  - Management science foundations
  - Economics foundations
  - Sociology foundations

SIGNIFICANT FINDINGS OF THIS CHAPTER

- **Software engineering** is an engineering discipline that studies the nature of software, approaches and methodologies of large-scale software development, and the laws behind software behaviors and software engineering practices. This is a guideline in our search of software engineering theories and foundations towards a matured engineering discipline.

- The *characteristics of theoretical* problems under study in a field are abstract, inductive, mathematics-based, and formal-inference-centered; while those of *empirica*l problems are concrete, deductive, data-based, and experimental-validation-centered (Theorem 1.1).

- The theories and techniques of software engineering are centered by the cognitive, organizational, and resources issues (see the SECM model).

- Software engineering is a unique and **the most complicated engineering discipline** that is ever faced in the human history. The constraints of software engineering are inherited by its intangibility, complexity, and diversity.

- According to **the abstraction and descriptivity model**, the abstract levels of cognitive information of both the objects and their behaviors can be divided into the levels of analogue objects, diagrams, natural languages, special notation systems, and mathematics. Software engineering was using lowtech means (lower level abstraction) to deal with hightech problems (higher level abstraction). This is the root of a whole bunch of problems in software engineering.

- The **problem in software engineering** is not solely an **empirical** one rather than a **theoretical** one. That is, the same set of fundamental problems that could not been overcome in the last four decades indicates the existence of fundamental constraints that need **theoretical investigations** to reveal the laws and principles behind all the problems.

- A particular gap in the current **software engineering curriculum** is the lack of a fundamental framework that provides students and practitioners for overarching, durable, and multidisciplinary theories and foundations, in order to explain a great many complicated phenomena and problems of software engineering in terms of a core set of fundamental principles.

- A **rigorous and theoretical approach** is needed to seek the fundamental principles and laws of software engineering, and their transdisciplinary foundations required by the nature of the problems in software engineering.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

- **The IME model:** The information-matter-energy (IME) model provides a generic world view, which reveals that the concrete and abstract worlds can be modeled as three essences known as matter, energy, and information.

  - According to the IME model, information plays a vital role in connecting the physical world and the abstract world. Software is a special type of behavioral information of computing and a means of interaction between the information world and the physical world. The nature of software makes software engineering a unique discipline, which is innately the most complicated engineering branch that humans experienced, and inherently the most overarching transdisciplinary field in both theories and applications. These are also the reasons that set forth software engineering as an ideal testbed for existing theories and methodologies of a wide range of science and engineering disciplines from mathematics to cognitive informatics, and from management science to sociology.

Software Engineering as a Discipline

• **Software** is an intellectual artefact that provides a *solution* for a *repeatable* computer application, which enables existing tasks to be done easier, faster, and smarter, or which provides innovative applications for the industries and in everyday life. Although the nature of software has been perceived quite differently in research and practice of computing and software engineering, the following perceptions on the nature of software can be found in the literature:

- Software is a mathematical entity
- Software is a concrete product
- Software is a set of behavioral information

• **Software engineering** is a discipline that adopts engineering approaches, such as established methodologies, processes, measurement, tools, standards, organisation methods, management methods, quality assurance systems and the like, in the development of large-scale software seeking to result in high productivity, low cost, controllable quality, and measurable development schedule.

Fundamental Constraints of Software Engineering

• **The abstraction and descriptivity model:** The abstract levels of cognitive information of both the objects and their behaviors can be divided into the levels of analogue objects, diagrams, natural languages, special notation systems, and mathematics.

• **The cognitive constraints of software engineering** (Theorem 1.2) states that fundamental constraints of software engineering stem from intangibility and intricate inner connections of software systems, and the cognitive complexity to explicitly describe them.

• It is noteworthy that in software engineering the objects under study are system and human behaviors in the abstract world rather than concrete entities in the real world. This is a fundamental difference between software engineering and other engineering disciplines.

• The **expressive power** of icons and diagrams are inadequate in software engineering because they make software design and specifications vague.

- It is recognized that architectures of software are complex interrelated objects with functional variables and constraints; and behaviors of software are embedded relational processes. These types of abstract and complicated entities may only be expressed without implication by professional notation systems, because only more abstract and precise means is powerful enough to express an object at a given level of abstraction.

- **The law of explicit descriptivity** (Theorem 1.3) states that only a higher level of more abstract, precise, and rigor means is required to express an object at a given level of abstraction.

- Symbolic notations and mathematics are the key means for expressing and embodying software behaviors, because they are at higher level abstraction and therefore with more adequate descriptive power.

- **The software engineering constraint model (SECM):** SECM models a comprehensive set of 14 basic constraints of software engineering encompassing three categories known as

  - **The cognitive constraints:** such as intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, unquantifiable quality measures.
  - **The organizational constraints:** such as time dependency, conservative productivity, and labor-time interlock.
  - **The resources constraints:** such as costs, human dependency, and hardware dependency.

- The problems and difficulties in software engineering are inherited; behind them there are a set of cognitive, informatics, technical, systematic, managerial, and social reasons, which are the driving force of all the constraints in software engineering.

Approaches to Software Engineering

- Various approaches have been sought in order to deal with the fundamental problems in software engineering as identified in the previous section. The historical development in **software engineering approaches** can be summarized as shown below:

  - Programming methodologies

- Software development models
- Automated software engineering environments
- Formal methods
- Software engineering processes
- Theoretical foundations

Transdisciplinary Foundations of Software Engineering

• **Transdisciplinary foundations of software engineering:** Although software engineering has emerged as a branch of computer science, it is recognized that software engineering requires much broader and multidisciplinary foundations as follows:

- Principles of Software Engineering
- Engineering Foundations
- Philosophical Foundations
- Mathematical Foundations
- Computing Foundations
- Linguistics Foundations
- Information Science Foundations
- Cognitive Informatics Foundations
- Systems Science Foundations
- Management Science Foundations
- Economics Foundations
- Sociology Foundations

• The complicity, broadness, and transdisciplinary nature of software engineering sets forth **an ideal testbed** for existing theories and methodologies of a wide range of science and engineering disciplines, such as philosophy, mathematics, computing, linguistics, information science, cognitive informatics, system science, management science, economics, sociology, and engineering organization.

• From another angle, software engineering theories can contribute significantly to those above mentioned disciplines. This forms the theme of this book towards a transdisciplinary and rigorous framework of software engineering foundations.

Questions and Research Opportunities

1.1 What is the nature of software engineering? Is software engineering unique or special in relation to the other engineering disciplines?

1.2 According to Theorem 1.1, the criteria of *theoretical* problems under study in a field are abstract, inductive, mathematics-based, and formal-inference-centered; while the criteria of *empirica*l problems under study in a field are concrete, deductive, data-based, and experimental-validation-centered. Try to identify three example problems for each of these categories.

1.3 On the basis of Ex. 1.2, discuss why the basic problems of software engineering, such as software system architectural modeling and behavioral description, cannot be pursued solely by empirical means and methodologies, but are needed for theoretical and formal means and methodologies.

1.4 A number of myths have been identified in Section 1.1.2. Try to find an additional myth on perceiving software engineering and explain why it is a myth theoretically or empirically.

1.5 Discuss the meaning of Theorem 1.2 – the *Information-Matter-Energy* (IME) *model* – and its impact on understanding the nature of software and software engineering activities.

1.6 Referring to the general complexity threshold of software engineering as given in Definition 1.2, discuss why sizes of software systems are so important in determining the methodologies and techniques of software engineering.

1.7 The Hierarchical Abstraction Model of System Descriptivity (HAMSD) reveals that the abstract levels of cognitive information and knowledge can be divided into five levels such as those of *analogue objects, diagrams, natural languages, professional notations,* and *mathematics.*

Based on Theorem 1.4 and the HAMSD model, discuss what the abstraction levels of software systems and the UML diagrams are, and whether UML is adequate to model and describe a software system.

1.8 What are the usages for studying the basic constraints of software engineering and their categories?

1.9 Explain what are the *cognitive constraints* of software engineering.

1.10 Explain what are the *organizational constraints* of software engineering.

1.11 Explain what are the *resource constraints* of software engineering.

1.12 Discuss the software dependencies on humans and hardware as modeled in the resource constraints of software engineering.

1.13 Software vs. Hardware: Many traditional hardware architectures and behaviors of systems can be digitalized, therefore implemented by software. Taking an automobile as an example:

a) Construct a conceptual model of the automobile;

b) Try to analyze which parts (as many as possible) of the automobile can be replaced by software;

c) Discuss what your understanding of the relationships between software and hardware on the basis of this example.

1.14 Try to use Theorem 1.6 to prove (or explain) Corollary 1.4 – the *no number one principle* in software engineering.

1.15 Explain Table 1.4 on the domain coverage of the different approaches to software engineering, and analyze their strengths and weaknesses.

1.16 Why is software productivity conservative? Is there any technique that may help to increase programming productivity? (Refer to Theorem 1.7)

1.17 Summarize the multidisciplinary foundations of software engineering that will be covered in this book, and explain why the transdisciplinary approach needs to be taken in understanding the fundamental problems and potential solutions of software engineering.

1.18 Searching for Relevant Theories: On the basis of Ex. 1.17, try to propose one or more potential disciplines or subdisciplines that may contribute to the maturity of software engineering theories and methodologies.

1.19 There is an argument that programming has no scientific foundations because both professionals and amateurs can write programs. Do you agree with this observation? Why?

1.20 Why have more than half of software engineering projects failed in the history? Is this a theoretical, organizational, cognitive, or operational problem?

1.21 What are the attributes of software quality and can they be quantitatively measured, therefore be controlled?

1.22 Is time and labor interchangeable in software engineering? If so, what would be the constraints for the interchangeability between them?

1.23 How is a project team optimally organized in large-scale software engineering projects?

1.24 What would be the mathematical means for dealing with the cognitive complexity of software engineering?

1.25 Software engineering methodologies have been evolved from programming methods, software development models, CASE tools, formal methods, and the software engineering process, to transdisciplinary theoretical foundations. Referring to Section 1.4, analyze the advantages and disadvantages of each approach to software engineering.

1.26 Read the following classic article:

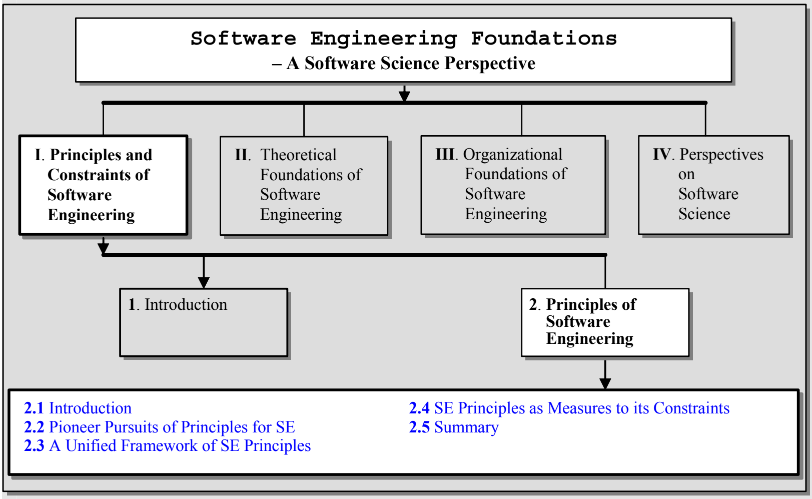
> Edsger W. Dijkstra (1972), The Humble Programmer, The 1972 Turing Award Lecture, *Communications of the ACM*, 15(10), pp.859-866.

Discuss the following topics in a group:

- About the author.
- What was the status of software engineering in the 1970s?
- Why would professional programmers feel humble? What limitations of programmers and/or techniques of software engineering resulted in this perception?
- What conclusions derived in the articles interested you? Why?
- Express your arguments or counter-points on any of the conclusions.

Chapter 2

PRINCIPLES OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
1. Introduction
2. Principles of Software Engineering
2.1 Introduction
2.2 Pioneer Pursuits of Principles for SE
2.3 A Unified Framework of SE Principles
2.4 SE Principles as Measures to its Constraints
2.5 Summary

2. Principles of Software Engineering

Knowledge Structure

❍ Pioneer pursuits of principles for software engineering

- Parnas' principles of SE
- Hoare's principles of SE
- Brooks' principles of SE
- Wasserman's principles of SE
- IEEE SESC's principles of SE
- IEEE Software magazine's principles of SE

❍ A unified framework of software engineering principles

- Elicitation of fundamental principles of SE
- The unified framework of SE principles
- Description of the fundamental principles of SE

❍ Software engineering principles as measures to its constraints

- Principles for coping with the cognitive constraints
- Principles for coping with the organizational constraints
- Principles for coping with the resource constraints
- A systematic view on mapping between the principles and constraints

Learning Objectives

- To view software engineering principles as fundamental theorems and laws that constrain software system behaviors and their design and implementation processes.
- To be aware of the major pioneer work of leading scientists in pursuing of software engineering principles in the last four decades.
- To gain a unified and coherent framework of software engineering principles.
- To understand the interrelationship between software engineering principles and constraints with a systematical view.
- To be able to apply the software engineering principles to tackle the fundamental constraints of software engineering.

"The more science becomes divided into specialized disciplines, the more important it becomes to find unified principles."

Herman Haken (1977)

"As a software development professional, you need knowledge of specific technologies to do your job. But you need knowledge of software engineering principles to do your job well."

Steve McConnell (1999)

2.1 Introduction

Principles of a scientific or engineering field are the basic heuristic rules, based on theoretical and empirical foundations, which provide a fundamental and conceptual means for conducting research, performing practice, and explaining various phenomena in the field.

Engineering sciences are disciplines of human enquiries that seek solutions for complicated problems and systems that could not be done by separate individuals. The key aim of engineering is to repetitively produce complicated artifacts in an efficient way. Thus, to many professionals, engineering means systematic planning, teamwork, rigorous process, repeatability, as well as efficiency.

Software engineering is a maturing engineering discipline that adopts the generic engineering principles in the development of large-scale software, which could not be produced by individuals. Currently, software development is evolving from the laboratory-oriented and all-round-programmer-based practice to an industry-oriented and process-based platform, and software developers are experiencing changes of roles from craftsmen to regulated professionals – the software engineers. The practices of the former are based on personal talents, tastes and art, while those of the latter are based on disciplined processes and repeatable professional activities.

Studying the vast literature of software engineering, readers may find that pioneers in software engineering have addressed many fundamental problems with insightful visions, and a large set of fundamental principles of software engineering has been laid. In the remainder of this chapter, the fundamental principles of software engineering will be presented as follows. Section 2.2 reviews the major pioneer pursuits of principles for software engineering in the last four decades, which provide a whole picture for understanding the theories and foundations of software engineering. Section 2.3 presents a unified framework of software engineering principles with a comprehensive set of 31 commonly identified fundamental principles of software engineering. Then, Section 2.4 treats these fundamental principles of software engineering as a set of powerful measures to tackle the 14 basic constraints of software engineering as identified in Section 1.3.

2.2 Pioneer Pursuits of Principles for Software Engineering

Since the coining of the term software engineering in 1968 [Bauer, 1976; Naur and Randell, 1969], efforts for attempting to identify the principles of software engineering and to explain its implication and extension have been continued. Although technologies have been changing from time to time, the fundamental principles of software engineering have remained constant as the crystallization of theories and methodologies over a long period of time.

A *principle* is a generic theorem, rule, or law of a theory that can be applied to a wide range of cases or instances in a field of study. A principle serves as a fundamental predicate for logical reasoning and deduction. Principles can be classified into two categories known as the *formal* and *empirical/heuristic* principles. Most known principles in software engineering are empirical and heuristic. For supporting rigorous reasoning and decision making in software engineering, formalization of those empirical principles seems profoundly important and necessary.

Definition 2.1 *Software engineering principles* are a set of fundamental and coherent theorems and laws that constrain the behaviours of software systems and the processes of their development.

Software engineering principles are the essential knowledge that a software engineer needs in order to develop software scientifically, effectively, and professionally.

This section surveys the pursuits of fundamental principles in software engineering by pioneers and leading institutions during the last four decades in the development of software engineering. The work and contributions of Davis L. Parnas, C.A.R. Hoare, Frederick P. Brooks, and Anthony I. Wasserman, as well as the IEEE Software Engineering Standardization

Committee (SESC) and IEEE Software Board of Advisors, will be briefly reviewed.

2.2.1 PARNAS' PRINCIPLES OF SOFTWARE ENGINEERING

During the 1970s through the 1990s, David L. Parnas enunciated five important principles of software engineering [Parnas, 1971/72/76/78/86/94a/94b/95/96/97/98; Aspray et al., 1996; Hoffman and Weiss, 2001] as follows:

- DP1: Information hiding
- DP2: Modularization
- DP3: Engineering approach
- DP4: Professional responsibility
- DP5: Documentation

The following subsections describe Parnas' insightful visions on the principles of software engineering.

2.2.1.1 Information Hiding

Information hiding (DP1) is a widely accepted principle for software engineering identified by Parnas [Parnas and Clements, 1986], which supposes that unnecessary details of information of software at a certain level should be masked in lower level implementations.

Parnas viewed that the hierarchical approach to design is useful for representing and describing a system following the principle of information hiding. He enunciated that "a programmer is most effective if shielded from, rather than exposed to, the details of system parts other than his own."

In his classic papers [Parnas, 1971/72], Parnas explained that it was ill-structured information distribution that made software systems dirty by involving almost invisible connections (coupling) between supposedly independent modules. Parnas believed that limitation of the information coupling among modules was the key to improve design quality in software system design.

Information hiding is considered as one of the major methodologies for implementing modularization, functional decomposition, and stepwise refinement in structured programming. It is also the foundation of a number of modern software modeling techniques such as abstract data types, encapsulation, object technology, and software components.

An equivalent expression of information hiding in software engineering is *abstraction*, which describes how common properties and shared information of a set of objects may be elicited and explicitly represented. As a consequence of abstraction, the uncommon properties and unshared information of software components are then hidden at lower level structures.

2.2.1.2 Modularization

Modularization (DP2) is a generic system construction approach in almost all engineering disciplines. Modularization is important in software engineering, particularly in structured programming, for dealing with complexity in software architectural design and implementation.

The central idea of modularization is the assumption that a software system can be broken up, or decomposed, into smaller functional pieces during system design, and when they are implemented, the system can be composed by integrating the pieces. The software pieces were called modules during the 1950s through the 1980s, and then they are known as classes/objects and components beginning in the 1990s. All these adopt the philosophy known as *divide-and-conquer*.

Parnas considered that when systems are decomposed into a large number of modules, structures need to be emphasized [Parnas, 1972], sharing similar ideas with the concept of structured programming proposed by Dahl, Dijkstra, and Hoare in the same period [Dahl et al., 1972] and the contemporary concept of object-oriented and component-based software engineering.

2.2.1.3 Engineering Approach

Parnas (1971/72) as well as Baure (1976) supposed that software development has to take the *engineering approach* (DP3) in which the common best practice matured in other engineering disciplines need to be adopted. Software engineering should focus on 'fundamental knowledge' rather than specific techniques, 'program design' rather than 'language syntax' or things that are 'neither mathematical truths nor facts about the world.' Parnas wrote [Parnas, 1996]:

> "Engineering educators have long known that their students must be prepared to work in rapidly changing fields. We have recognized that the educational program must stress fundamentals – science, mathematics, and design discipline – so that graduates will find their education still valid and useful late in their careers. Most of the books that I used in my own engineering education are still correct and relevant, several decades later. In contrast, many

> introductory programming books are considered out of date before the students who use them have graduated."

In a number of papers [Parnas, 1995/96/97], Parnas stresses the differences between software engineering and computer science, and emphasizes that software engineering is not computer science. As a result of this argument, we now understand more about the nature of software engineering as a much broader field than that of computer science, which encompasses the engineering organization foundations, cognitive informatics foundations, and system science foundations [Wang, 2005g/05i/05*l*]. Further, the objects of study in software engineering are behavioral information of both machine and natural intelligence, which are more fundamental at the root of human knowledge than that of computing.

2.2.1.4 Professional Responsibility

Professional responsibility (DP4) is identified by Parnas as a basic principle for software engineering [Parnas, 1994a]. He classified the professional responsibilities of software engineers into three categories known as the personal, professional, and social responsibilities. According to Parnas, the three categories of responsibilities can be defined as follows:

> "*Personal responsibilities* are those that are shared by all persons, no matter what is their profession or educational background.
>
> "*Professional responsibilities* are additional responsibilities that we take on because we have become members of a particular profession such as medicine, journalism, or engineering.
>
> "*Social responsibilities* are responsibilities toward society as a whole rather than toward other individuals."

The particularly important responsibilities for software engineers emphasized by Parnas are on accepting individual responsibility, solving the real problem, being honest about capability, producing reviewable designs, and software maintainability.

2.2.1.5 Documentation

Parnas and his colleague pointed out that: "Design without documentation is not design [Parnas and Clements, 1986; Hoffman and Weiss, 2001]," and "If it is not documented, it is not done [Parnas, 1994b]." Computer programs were considered as the combination of algorithms and data structures; whilst software is supposed as programs plus documents. Hence, *documentation* (DP5) is a basic property of software systems.

Parnas puts emphases on the importance of documentation in software engineering not only during the design phase of software development, but also during implementation and maintenance [Hester, Parnas, and Utter, 1981]. According to the discussions on the fundamental constraints of software engineering in Section 1.3, documentation is a major approach to dealing with intangibility and cognitive complexity of all intermediate and final work products in software engineering.

2.2.2 HOARE'S PRINCIPLES OF SOFTWARE ENGINEERING

In a number of basic studies [Hoare, 1969/73/75/80/94], C.A.R. (Tony) Hoare identified seven generic principles of software engineering as follows:

- TH1: Professionalism
- TH2: Vigilance
- TH3: Sound theoretical knowledge
- TH4: Using tools
- TH5: Abstraction
- TH6: Structured programming
- TH7: Readability

The above list provides Hoare's insight on the 'sound engineering principles' as Bauer implied. The following subsections describe Hoare's views on the principles of software engineering.

2.2.2.1 Professionalism

Hoare defined professionals as those who earn their living by using their special knowledge and expertise. *Professionalism* (TH1) represents the qualification, identity, ethic, and pride of a group of highly qualified people in a sector of society or engineering fields.

Professional engineers value and maintain professional integrity. They should be able to understand client needs, and have the capability, confidence, and status to advise and persuade clients of their genuine requirements. Their activities are founded on professional skills as well as tried and tested techniques. They will aim to recommend a solution that is cost-effective, simple, efficient, practical, and satisfactory [Hoare, 1975/94].

2.2.2.2 Vigilance

Vigilance (TH2) is the awareness of possible dangers, risks, and/or difficulties in a field. It can also be an action of keeping careful watch for the dangers, risks, and difficulties. Vigilance in software engineering is meant by Hoare as watchfulness or practice with caution [Hoare, 1975]. Dijkstra used a similar term "humble" [Dijkstra, 1972] to describe this important characteristic of professional software engineers.

2.2.2.3 Sound Theoretical Knowledge

Sound *theoretical knowledge* (TH3) refers to principles, mathematical theories, and standard codes of practice based upon which engineering practices are premised [Hoare, 1975; Hoare and Jones, 1989]. This is an analogy to other matured engineering disciplines, such as mechanical engineering and electrical engineering, in which mathematical modeling and logical reasoning are applied to represent system components and the ways in which the components fit together.

2.2.2.4 Using Tools

It has been observed that all engineering disciplines adopt specialized tools in order to improve productivity and assure quality. *Using tools* (TH4) is a sign of professionalism in engineering.

Software engineers should be able to select and use a range of tools of proven quality, effectiveness, efficiency, precision, and convenience [Hoare, 1975; Hoare and Jones, 1989].

2.2.2.5 Abstraction

Abstraction (TH5) is an approach to deal with common properties of objects under study with symbolic representations and with rigorous treatment. Mathematic logic, set theory, and algebra are perfect paradigms of applications of abstraction.

Hoare and his colleagues attempted to introduce abstract means of mathematics into programming language design and programming [Hoare, 1986; Hoare et al., 1987]. Hoare contributed two powerful concepts for abstraction in programming. One is the *process* concept for program behavior abstraction [Hoare, 1978/85]. The other is *axiomatic semantics* for program semantic abstraction and their correctness proof on the basis of the semantics [Hoare, 1969].

2.2.2.6 Structured Programming

In *Structured Programming* (TH6), Dahl, Dijkstra, and Hoare proposed a systematic approach to the design, development, and documentation of computer programs [Dahl, Dijkstra and Hoare, 1972]. The concept of stepwise design refinement is explored. Hoare also studied the logic of engineering design, and developed the axiomatic basis for computer programming [Hoare, 1969].

2.2.2.7 Readability

Readability (TH7) of software refers to the extent of how understandable a program is in a given programming language. Hoare asserted [Hoare, 1973]:

> "The readability of programs is immeasurably more important than their writeability." However, "The objective of readability by human beings has sometimes been denied in favor of readability by a machine; and sometimes even been denied in favor of abbreviation of writing, achieved by a wealth of default conventions and implicit assumptions."

Hoare believed that readability should be an important attribute of all programming languages [Hoare, 1973/80; Hoare and Wirth, 1966]. Readability may also be extended to documentations and design work products in software engineering. This principle also implies a striving for elegance and simplicity in software design and implementation.

2.2.3 BROOKS' PRINCIPLES OF SOFTWARE ENGINEERING

In his well received paper [Brooks, 1987], "*No Silver Bullets – Essence and Accident in Software Engineering*," Frederick Brooks predicated that there will be no individual methodology or tool that can solve all fundamental problems in software engineering and yield a ten-fold productivity boom in software development in the given decade. Brooks drew this conclusion by identifying the four basic constraints of software development known as the *essences* below, in contrast to the trivial difficulties called *accidents* in software engineering:

- FB1: Complexity
- FB2: Conformity

- FB3: Changeability
- FB4: Invisibility

Brooks argued that essential difficulties of software engineering encompass the inherited complexity, real-world conformity, inevitable changeability, and abstractive invisibility as shown in Fig. 2.1. The following subsections describe Brooks' discoveries on the principles of software engineering with the framework as modeled in Fig. 2.1.

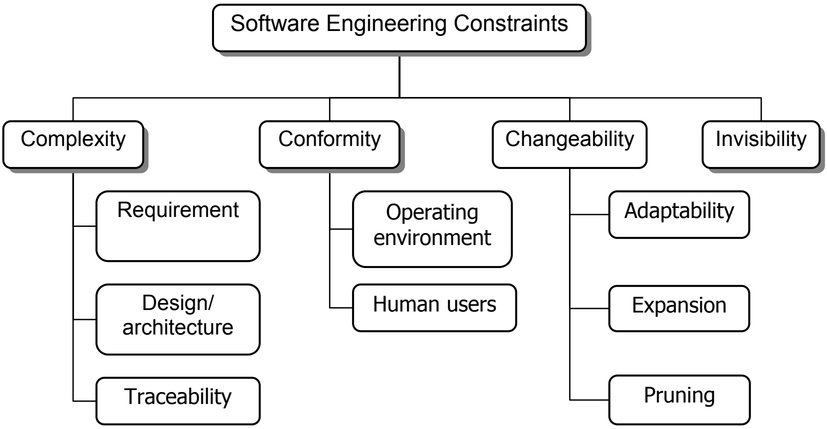


Figure 2.1 Brooks' constraints of software engineering

2.2.3.1 Complexity

Complexity (FB1) of software refers to the fact that software complexity is determined by the nature of problems. No language, tool, or technology may reduce the complexity of a given problem itself.

The complexity of software can be analyzed from the aspects of requirement complexity, design/architectural complexity, and decision traceability complexity. The naturally inherited complexity of the object of study makes software engineering one of the most complicated engineering fields.

Properly designed software has no homogeneity, which prevents the scaling of low-level solutions to higher-level problems. As complexity is an essential property of software, it cannot be abstracted away without losing the correctness and applicability of the result [Brooks, 1975/95].

2.2.3.2 Conformity

Conformity (FB2) of software refers to the natural and human constraints to a software system. The proper functioning of a software system is both human and hardware dependent, because the given software system must seamlessly adapt to an operating platform and to users who apply it in a certain working environment.

In addition, there are existing regulations, standards, requirements, and quality expectations to be confirmed setting forth for a software system. Therefore, software is always being put into a context where decision parameters of the environment are out of the influence of the designers [Brooks, 1975].

2.2.3.3 Changeability

Changeability (FB3) of software refers to the constant need for change and adaptation of functionality. Brooks considered software changeability to encompass adaptability, expansion, and pruning.

For a physical product, a rule of thumb is "if it does not break, do not fix it." However, a software system is under continuous psychological and practical pressure to be improved from both motivations of developers and users of the system [Brooks, 1975]. This is because there is no perfect solution for a software system, and even an originally good solution may become obsolete over time, or due to technical advance and additional user demands.

Changeability is the innate characteristic to explain why people adopt a software solution for a given problem rather than a hardware solution. As a consequence, the requirements and expected functionality of a software system are always a moving target.

2.2.3.4 Invisibility

Invisibility (FB4) of software refers to its abstract trait and intangibility. Software defies physical representation, because its time and operational complexity cannot be successfully captured and explicitly described by using physical models, an invaluable tool for designers and inventors in other fields. Brooks perceived that software invisibility "not only impedes the process of design within one mind, it severely hinders communication among minds [Brooks, 1987]."

The invisibility of software lies in two interlocked aspects: the architecture and behaviors of software. The former is very difficult to be visualized if it is still possible. The latter is almost impossible to be visualized by any diagram-based or "geometrical" means. Recent studies on dealing with software invisibility adopt denotational mathematics such as Real-Time Process Algebra (RTPA) [Wang, 2002a/02b/03c/06a/07a], which will be described in Chapter 5 on mathematical foundations of software engineering.

2.2.4 WASSERMAN'S PRINCIPLES OF SOFTWARE ENGINEERING

In an article on "Toward a Discipline of Software Engineering," Anthony Wasserman surveyed the literature and summarized the following eight principles of software engineering [Wasserman, 1996]:

- AW1: Abstraction
- AW2: Methods and notations
- AW3: Prototyping
- AW4: Modularity and architecture
- AW5: Lifecycle and process
- AW6: Reuse
- AW7: Metrics
- AW8: Tools and integrated environments

Wasserman identified that despite the rapid changes in software engineering techniques, the above fundamental principles have been kept constant and together constitute a viable foundation for software engineering. The following subsections describe Wasserman's perceptions on the principles of software engineering.

2.2.4.1 Abstraction

"*Abstraction* (AW1) is a fundamental technique for understanding and solving problems [Wasserman, 1996]." Abstraction is a common intellectual technique for managing the understanding of complex items. It allows us to concentrate on a problem at some generalized level without regard to irrelevant low-level details. Abstraction is the central concept of information hiding, which lets software developers focus on the appropriate level of detail concerning a software component.

2.2.4.2 Methods and Notations

Wasserman believed that "Analysis and design *methods and notations* (AW2) are basic tools for communication in an engineering discipline. … If software engineering is to mature as a discipline, standard specification and design notations have to be developed [Wasserman, 1996]."

It is perceived that analysis and design methodologies as well as notations are helpful to fill the cognitive leap from requirement specification to system implementation in software engineering. Analyses consider the problem space, address the structure of the problem, and include the logical structures of objects in the real world. Design deals with the structures of the system that implements a solution to the problem.

Wasserman identified that almost all engineering disciplines use standard blueprints, block diagrams, and/or schematic diagrams. But software engineering was an exception, where hundreds of different notations and programming languages were used and the community is still expecting new ones. He suggested that it is possible to develop a method-independent software notation that represents a system design which may be implemented by various development methods.

2.2.4.3 Prototyping

Prototyping (AW3) is a universal practice in engineering disciplines. Many rapid development methods are based on prototyping.

Wasserman considered prototyping was particularly useful in Graphic User Interface (GUI) design and implementation. He wrote: "Prototyping of the user interface is the most effective way to elicit user requirements and to improve usability of applications [Wasserman, 1996]."

As well as a system can be designed and described hierarchically by different levels, it can also be implemented in the same way. Prototyping is a natural approach to implement a complex software system by different level of abstraction from the top-down.

2.2.4.4 Modularity and Architecture

As Parnas pointed out, *modularity and architecture* (AW4) are a pair of techniques that complement each other. Modules are the materials, and architecture is the framework that accommodates the modules and allows them to work as a whole.

Wasserman identified that "*Software architectures* play a major role in determining system quality and maintainability [Wasserman, 1996]." He commented that of all the various qualities of software design, none has proven over time to be more significant than modularity. Objects, components, design patterns, and application frameworks are modern techniques that apply the modularization principle of software engineering.

2.2.4.5 Lifecycle and Process

Wasserman considered software *lifecycles and processes* (AW5) to be the infrastructure of software development. Although small- or medium-sized software may adopt ad hoc and rapid development methods, large-scale software system development, which involves numerous developers, supporting staff, and customers for months or years, requires well-defined processes. Also, large and complex software development is both the goal and the reason of software engineering. Therefore, he suggested that having some defined and manageable process for software development is much better than not having one at all [Wasserman, 1996].

2.2.4.6 Reuse

Wasserman agreed that "*reuse* (AW6) of existing software development assets is an essential part of any software development process [Wasserman, 1996]."

However, he observed that "effective reuse beyond the level of function and class libraries has proven to be more difficult than hoped," because it was hard to build high quality reusable components. He observed that there were fundamental technical barriers in conducting code reuse, such as application diversity, delayed time-to-market, and extra work involved.

2.2.4.7 Metrics

Wasserman perceived that *metrics* (AW7) play an important role in quantifying software engineering measurement. He surveyed a wide range of software metrics on software processes, quality, defects, productivity, schedule, sizes, architectures, tests, and costs. Based on the survey, he asserted that: "improvements in the software development process and system quality cannot be evaluated without an effective metrics effort [Wasserman, 1996]."

2.2.4.8 Tools and Integrated Environments

To increase efficiency and productivity, software engineering processes have to be supported by appropriate *tools and integrated environments* (AW8). Wasserman supposed that the infrastructure of software engineering encompasses integrated tools of object management, process management, communication, and operating systems. "The application development environment and its tools should provide comprehensive and integrated support for the development process [Wasserman, 1996]."

Wasserman identified that tool integration in software engineering falls into five categories, such as platform integration, presentation integration, process integration, data integration, and control integration. The issues surrounding tool integration are complicated and involve both technical and business trade-offs. Therefore, much work needs to be done before comprehensive multivendor and multiplatform CASE tools may be achieved.

2.2.5 IEEE SESC'S PRINCIPLES OF SOFTWARE ENGINEERING

An international effort has been devoted by the IEEE Software Engineering Standards Committee (SESC), in collaboration with the IEEE/ACM Joint Committee on Software Engineering Body of Knowledge (SWEBOK) and ISO/IEC JTC1/SC7, to identify fundamental principles of software engineering during 1996 to 2000 [Davis, 1994; SESC, 1996/97/99; Tripp, 1996; Abran et al., 1999; Dupuis et al., 1999].

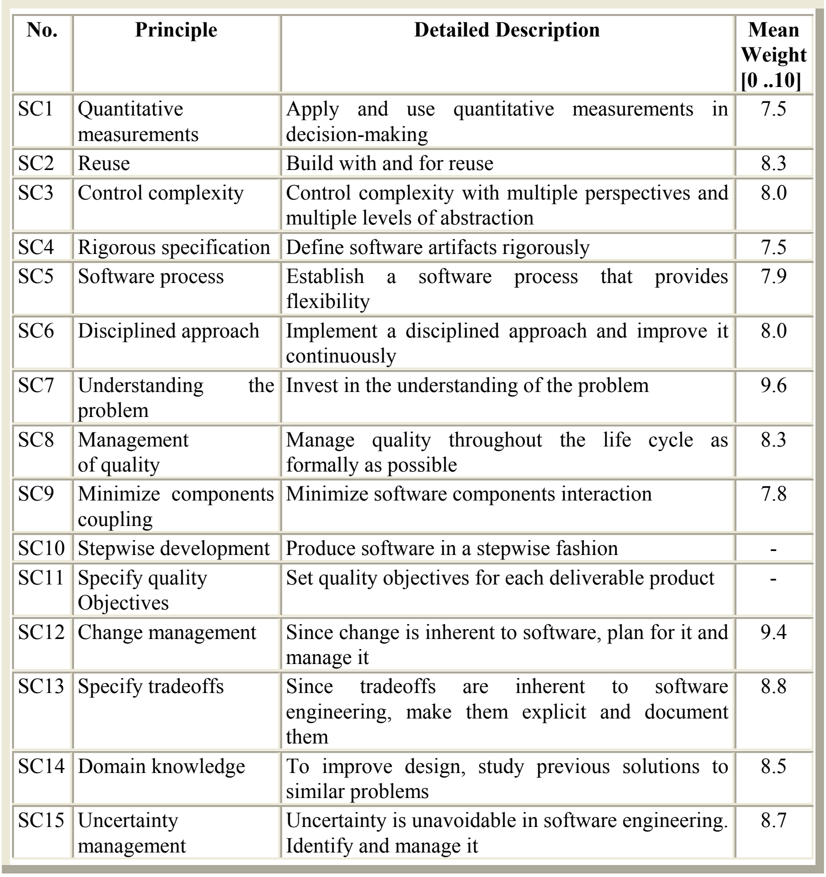
The First IEEE Forum on Software Engineering Standards Issues (SES'96) was held in Montreal, Canada in October 1996 as a part of the International Symposium on Software Engineering Standards (ISESS'96). Following the success of ISESS'96, two events, ISESS'97 and ISESS'99, were held in Walnut Creek, California and Curitiba, Brazil, respectively.

It is believed that based on the cumulated experience, we may identify the underlying principles of software engineering that are fundamental and hence enduring. The purposes of IEEE SESC were to promote the recognition of software engineering as a well-established discipline, and to provide a broader and richer framework for establishing relationships among groups of software engineering standards.

Robert Dupuis and his colleagues reported that a Delphi study was conducted in 1997 over the Internet among 14 renowned researchers to identify a first candidate list of fundamental principles of software engineering. A second workshop was held at the International Software Engineering Standards Symposium (ISESS'97) to reformulate or eliminate some of the principles and the selection criteria. Then, a second Delphi study was conducted in 1998 among 31 IEEE software engineering committee members in order to improve the set of principles. From this study a list of 15 fundamental principles of software engineering has been compiled [Dupuis et al., 1999] as shown in Table 2.1.

All the principles proposed in Table 2.1 provide a fundamental view on software engineering by identifying a set of durable characteristics and processes. According to a number of surveys, the most significant principles are SC7 – understanding the problem, SC12 – change management, and SC13 – specify tradeoffs. The least significant principles are SC9 – minimize components coupling, SC1 – quantitative measurements, and SC4 – rigorous specification. There are two principles, SC10 – stepwise development and SC11 – specify quality objectives, with the weights unidentified.

Table 2.1
The IEEE SESC Proposed Principles of Software Engineering



No.	Principle	Detailed Description	Mean Weight [0 ..10]
SC1	Quantitative measurements	Apply and use quantitative measurements in decision-making	7.5
SC2	Reuse	Build with and for reuse	8.3
SC3	Control complexity	Control complexity with multiple perspectives and multiple levels of abstraction	8.0
SC4	Rigorous specification	Define software artifacts rigorously	7.5
SC5	Software process	Establish a software process that provides flexibility	7.9
SC6	Disciplined approach	Implement a disciplined approach and improve it continuously	8.0
SC7	Understanding the problem	Invest in the understanding of the problem	9.6
SC8	Management of quality	Manage quality throughout the life cycle as formally as possible	8.3
SC9	Minimize components coupling	Minimize software components interaction	7.8
SC10	Stepwise development	Produce software in a stepwise fashion	-
SC11	Specify quality Objectives	Set quality objectives for each deliverable product	-
SC12	Change management	Since change is inherent to software, plan for it and manage it	9.4
SC13	Specify tradeoffs	Since tradeoffs are inherent to software engineering, make them explicit and document them	8.8
SC14	Domain knowledge	To improve design, study previous solutions to similar problems	8.5
SC15	Uncertainty management	Uncertainty is unavoidable in software engineering. Identify and manage it	8.7

2.2.6 IEEE SOFTWARE MAGAZINE'S PRINCIPLES OF SOFTWARE ENGINEERING

In reviewing the best influences during the software engineering's first 50 years [McConnell, 1999], the advisor board of IEEE Software selected the following 11 principles for software engineering developed in the past century.

- SW1: Reviews and inspections
- SW2: Information hiding
- SW3: Incremental development

- SW4: User involvement
- SW5: Automated revision control
- SW6: Internet development
- SW7: Programming languages hall of fame
- SW8: Capacity maturity model (CMM)
- SW9: Object-oriented programming
- SW10: Component-based programming
- SW11: Metrics and measurement

The following subsections describe the IEEE *Software* Editors' perceptions on the principles of software engineering [McConnell, 1999].

2.2.6.1 Reviews and Inspections

> *Reviews and inspections* (SW1): "One of the great breakthroughs in software engineering was Gerald Weinberg's concept of egoless programming – the idea that no matter how smart a programmer is, reviews will be beneficial. Weinberg's ideas were formalized by Michael Fagan into a well-defined review technique called Fagan inspections. The data in supporting of the quality, cost, and schedule impact of inspections are overwhelming. They are an indispensable part of engineering high quality software."

2.2.6.2 Information Hiding

> *Information hiding* (SW2): "David Parnas' 25-year old concept of information hiding is one of the seminal ideas in software engineering – the idea that good design consists of identifying "design secrets" that a program's classes, modules, functions, or even variables and named constants should hide from other parts of the program. While other insights into how to come up with the good design ideas in the first place, information hiding is at the foundation of both structured design and object-oriented design. In an age when buzzword methodologies often occupy center stage, information hiding is a technique with real value."

2.2.6.3 Incremental Development

> *Incremental development* (SW3): "The software engineering literature of the 1970s was full of horror stories of software meltdowns during the integration phase. Components were brought

together for the first time during 'system integration.' So many mistaken or misunderstood interface assumptions were exposed at the same time that debugging a nest of intertwined assumptions became all but impossible. Incremental development and integration approaches have virtually eliminated code-level integration problems on modern software projects. Of these incremental approaches, the daily build is the best example of a real-world approach that works. It minimizes integration risk, provides steady evidence of progress to project stakeholders, keeps quality levels high, and helps team morale because everyone can see that the software works."

2.2.6.4 User Involvement

User involvement (SW4): "We have seen tremendous developments in the past several years in techniques that bring users more into the software product design process. Techniques such as JAD sessions, user interface prototyping, and use cases engage users with product concepts in ways that paper specifications simply cannot. Requirements problems are usually listed as the number one cause of software project failure; these techniques go a long way toward eliminating requirements problems."

2.2.6.5 Automated Revision Control

Automated revision control (SW5): "It takes care of mountains of housekeeping details associated with team programming projects. In the Mythical Man-Month in 1975, Fred Brooks' "surgical team" made use of a librarian. Today, that person's function is handled by software. The efficiencies by today's programming teams would be inconceivable without automated revision control."

2.2.6.6 Internet Development

Internet development (SW6): "What we have seen with the Open Source development is just the beginning of collaborative efforts made possible via the Internet. The potential this creates for effective, geographically distributed programming is truly mind boggling."

2.2.6.7 Programming Languages Hall of Fame

Programming language hall of fame (SW7): "Programming languages hall of fame: FORTRAN, COBOL, Turbo Pascal, Visual

Basic. A few specific technologies have had significant influence on software development in the past 30 years.

"Academics and researchers talked about components and reuse for decades, and nothing happened. Within 18 months of Visual Basic's release, a thriving pre-built components market had sprung from nothing. The direct-manipulation, drag-and-drop, point-and-click programming interface was a revolutionary advance."

2.2.6.8 Capability Maturity Model

Capacity maturity model (SW8): "The Software Capability Maturity Model (CMM) is one of the few branded methodologies that has had any affect on typical software organizations. More than 1,000 organizations and 5,000 projects have undergone CMM assessment, and dozens of organizations have produced mountains of compelling data on the effectiveness of process improvement programs based on the CMM model."

2.2.6.9 Object-Oriented Programming

Object-oriented programming (SW9): "Object-oriented programming offered great improvements in 'natural' design and programming. After the initial hype faded, practitioners were sometimes left with programming technologies that increased complicity, provided only marginal productivity gains, produced unmaintainable code, and could only be used by experts. In the final analysis, the real benefit of object-oriented programming is probably not objects, per se, but the ability to aggregate programming concepts into larger chunks than subroutines or functions."

2.2.6.10 Component-Based Development

Component-based development (SW10): "Component-based development has held out much promise, but aside from a few limited successes, it seems to be shaping up to be another idea that works better in the laboratory than in the real world. Component-version incompatibilities have given rise to massive setup-program headaches, unpredictable interactions among programs, de-installation problems, and a need for utilities that restore all the components on a person's computer to 'last known good state.' This

> might be one of the ten best for the twenty-first century, probably not for the twentieth."

2.2.6.11 Metrics and Measurement

> *Metrics and Measurement* (SW11): "Metrics and measurement have the potential to revolutionize software engineering. In the few instances in which they have been used effectively (NASA's Software Engineering Lab and a few other organizations), the insights that well-defined numbers can provide have been amazingly useful. Powerful as they can be, software process measurements are not the end, they are the means to the end. The metrics community seems to forget this lesson again every year."

Steve McConnell concluded that "An investment in learning software engineering principles is a particular good investment for a software professional to make because that knowledge will last a whole career – not be half obsolete within three years (as those of software development technologies) [McConnell, 1999]."

2.3 A Unified Framework of Software Engineering Principles

The ultimate objective of investigations into the principles of software engineering is to build an integrated inference framework with axioms implicitly defining primitive concepts, principles, and rules in order to construct higher order concepts and theories of software engineering. The fundamental principles are supposed to be self-evident if not tautologies. The implications of these principles, like most theorems on the other hand, may be far from trivial.

Section 2.2 presents a comprehensive survey on existing fundamental software engineering principles and methodologies. There are 50 principles identified individually by different authors or institutions. It would be useful to rethink the parable of the *blind men and the elephant* about searching the truth, which probably tells a quite similar situation as we got in seeking the principles for the giant elephant of software engineering.

This section introduces a method of constraints vs. measures to elicit and integrate the entire set of principles of software engineering. By this approach, each principle for software engineering is introduced as a measure that may be used to deal with one or more constraints and problems in software engineering. Then, all principles can be organized into a coherent framework of software engineering principles. The purposes and usability of the principles can also be clarified by mapping them into the constraints of software engineering.

2.3.1 ELICITATION OF FUNDAMENTAL PRINCIPLES OF SOFTWARE ENGINEERING

A significant problem in the software engineering community is the tendency to think that a brilliant new idea or powerful future tool will solve all current problems. This tendency is so strong that previously proven methods or tools would be forgotten as soon as a new idea gains some support, and consequently problems are re-solved in a new style. Maarten Boasson recognized that this is a perfect recipe for preventing progress in a discipline [McConnell, 2000].

Definition 2.2 A *principle of software engineering* is a generic theorem, rule, law, or methodology that can be applied to a wide range of cases and instances in software engineering.

The IEEE Software Engineering Standards Committee (SESC) proposed that the following criteria can be used to select the fundamental principles for software engineering [SESC, 1996/97/99; Tripp, 1996; Abran et al., 1999; Dupuis et al., 1999]:

- Principles are less specific than methodologies and techniques
- Principles are more enduring than methodologies and techniques
- Principles should be able to withstand the test of time
- Principles should be selected or elicited from best practices
- Principles should not contradict more general engineering or computer science principles
- Principles should be precise enough to be applied and implemented
- Principles should not conceal a tradeoff

Table 2.2
The Integrated Set of Software Engineering Principles

No	Unified Principles	Individually Identified Principles					
		Parnas (DP)	Hoare (TH)	Brooks (FB)	Wasserman (AW)	IEEE SESC (SC)	IEEE Software (SW)
1	Abstraction		TH5		AW1		
2	Decomposition and modularization	DP2	TH6		AW4	SC9	SW9 SW10
3	Information hiding	DP1					SW2
4	Engineering approach	DP3				SC6	
5	Professionalism	DP4	TH1				
6	Tools and environments		TH4		AW8		SW5
7	Documentation	DP5					
8	Stepwise refinement					SC10	SW3
9	Prototyping				AW10		
10	Engineering notations				AW2		SW6, SW7
11	Process models				AW5	SC5	SW8
12	Reuse				AW6	SC2	
13	Measurement and metrics				AW7	SC1	SW11
14	Cognitive complexity control			FB1 - 3		SC3	
15	Formal requirement specification					SC4, SC7	
16	Systematical quality assurance					SC8, SC11	
17	Review and inspection						SW1
18	Management engineering					SC12/15	
19	Domain knowledge					SC14	
20	Customer involvement						SW4
21	Feasibility analysis					SC13	
22	Comprehensibility		TH7				
23	Exception handling		TH2				
24	Divide-and-conquer	DP2					
25	Visualization			FB4			
26	Theoretical foundations		TH3				
27	Architecture and behavior modeling						
28	Standardization						
29	Systems engineering						
30	Engineering organization						
31	Cognitive engineering						

There are 50 software engineering principles identified in Section 2.2 with considerable overlaps as well as gaps. Also, it is noteworthy that the ways in which the principles were identified and proposed were ad hoc and informal. Eliminating overlaps among these proposals, there are 26 software engineering principles identified in the literature as shown in Table 2.2. Those principles that have not been identified in Section 2.2 are newly proposed by the author based on recent studies on the nature of software and software engineering, such as architectural and behavioral modeling, system engineering, engineering organization, cognitive engineering, and theoretical foundations [Wang, 2001g/02j/02g/04a/04b/05i/05k/05j/06a/06h/06i; Wang and Patel, 2000; Wang et al. 2006].

This section attempts to systematically derive a set of fundamental principles of software engineering that is commonly recognized in the work of Parnas, Hoare, Brooks, Wasserman, McConnell, IEEE SESC, Wang [Wang, 2004c/05a/05i/05k/05l; Wang et al., 2004], and of many other authors. It is achieved by a mapping between the principles identified in Section 2.2 as summarized in Table 2.2. A set of 31 fundamental principles of software engineering is then elicited, and each of them will be formally described in the remainder of this section.

2.3.2 THE UNIFIED FRAMEWORK OF SOFTWARE ENGINEERING PRINCIPLES

The relationship between the fundamental principles and basic constraints of software engineering is a complicated relational network. A main thread to analyze their relations is to perceive the constraints are the problems, and the principles are the measures to tackle the problems. On the basis of this thread, a mapping between the 31 fundamental principles developed in this chapter and the 14 basic constraints as identified in Section 1.3 can be carried out as shown in Fig. 2.2. A detailed mapping of the principles of software engineering into the basic constraints in the categories of cognitive, organization, and resources will be presented in Section 2.4 and summarized in Table 2.3.

2.3.3 DESCRIPTION OF THE FUNDAMENTAL PRINCIPLES OF SOFTWARE ENGINEERING

The following subsections describe each of the 31 fundamental principles for software engineering as identified in the unified framework shown in Table 2.2 with a more rigorous and formal treatment.

2.3.3.1 Abstraction

Abstraction is a powerful means of philosophy and mathematics. It is also a preeminent trait of the human brain identified in cognitive informatics studies. All formal logical inferences and reasoning can only be carried out on the basis of generic and abstract properties shared by a given set of objects under study. Abstraction is a powerful key to reduce complexity in software engineering.

Definition 2.3 *Abstraction* (PR1) is a software engineering principle for eliciting essential properties of a set of objects while omitting inessential details of them.

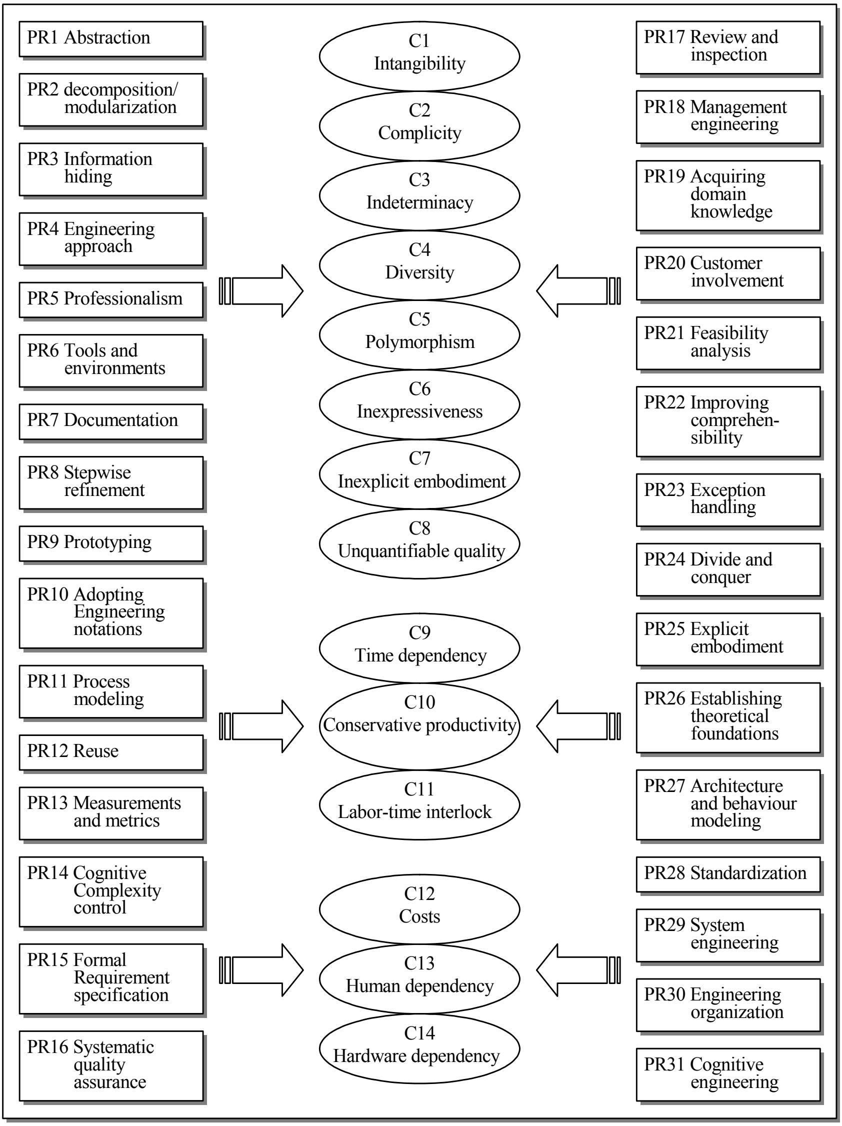

PR1 Abstraction
PR2 decomposition/ modularization
PR3 Information hiding
PR4 Engineering approach
PR5 Professionalism
PR6 Tools and environments
PR7 Documentation
PR8 Stepwise refinement
PR9 Prototyping
PR10 Adopting Engineering notations
PR11 Process modeling
PR12 Reuse
PR13 Measurements and metrics
PR14 Cognitive Complexity control
PR15 Formal Requirement specification
PR16 Systematic quality assurance
C1 Intangibility
C2 Complicity
C3 Indeterminacy
C4 Diversity
C5 Polymorphism
C6 Inexpressiveness
C7 Inexplicit embodiment
C8 Unquantifiable quality
C9 Time dependency
C10 Conservative productivity
C11 Labor-time interlock
C12 Costs
C13 Human dependency
C14 Hardware dependency
PR17 Review and inspection
PR18 Management engineering
PR19 Acquiring domain knowledge
PR20 Customer involvement
PR21 Feasibility analysis
PR22 Improving comprehen-sibility
PR23 Exception handling
PR24 Divide and conquer
PR25 Explicit embodiment
PR26 Establishing theoretical foundations
PR27 Architecture and behaviour modeling
PR28 Standardization
PR29 System engineering
PR30 Engineering organization
PR31 Cognitive engineering

Figure 2.2 Principles of software engineering as measures for its constraints

The mathematical principle of abstraction in set theory will be rigorously described in Section 4.2. Human abstract knowledge about the world may be described by a set of objects and their relations. For seeking generality and universal truth, either the objects or the relations can only be abstractly described and rigorously inferred by abstract models rather than real-world details.

Abstraction is recognized as a gifted capability of human beings. It is a basic cognitive process of the brain that is modeled at the meta cognitive layer of the cognitive models of the brain as described in Chapter 9. Throughout this book, it will be seen that only by abstraction the important theorems and laws of software engineering may be elicited and discovered from a great variety of phenomena and empirical observations in software engineering.

2.3.3.2 Decomposition/Modularization

In software engineering, decomposition as a process results in modularization. The contemporary concepts of software modules are such as objects, components, design patterns, and applications frameworks.

Definition 2.4 *Decomposition and modularization* (PR2) are a software engineering principle by which the functions of a software system are broken up and allocated into individual modules or components.

The central idea of modularization is based on the basic assumption that a software system can be broken up or decomposed into smaller functional pieces during system design. When these pieces are implemented, the system can be composed by integrating these pieces together. This is known as the re-composability of software. The decomposed software pieces were called modules during the 1950s to the 1980s, and then they are known as classes/objects or components since the 1990s.

An important condition of modularization is that *a global view*, i.e., the big picture of the whole system, should be consistently maintained and relationships among components and between the system and the components should be unambiguously understood. During functional decomposition, the modules/components and their interfaces should be clearly defined.

Another rule in modularization is *localization* by which logically related functions and operations will be encapsulated into one module or components. This will also improve module cohesion and decrease module coupling.

However, many evidences of problems in software engineering indicate that the decomposition-integration approach, or the basic assumption of the reassembly-ability of software, may be doubtful. Because, according to system theory as described in Chapter 10, the information loss during decomposition may lead to a *non-lossless conjunction* of the modules into a coherent system.

2.3.3.3 Information Hiding

Information hiding is a consequence of abstraction (PR1) and modularization (PR2). In other words, the methodology for information hiding is abstraction and modularization.

Definition 2.5 *Information hiding* (PR3) is a software engineering principle for the reduction and mask of unnecessary information of software at a given level from the lower level details.

The purpose of information hiding is to keep the uncommon properties and unshared information of software components at lower level constructs. Information hiding and modularization are helpful to limit the degree of coupling between software components, prevent propaganda of faults, changes, maintenance errors into other components of a software system.

Parnas proposes that the key to improve design quality in software system design is to limit the information coupling among modules [Parnas, 1971/72]. Parnas points out that "a programmer is most effective if shielded from, rather than exposed to, the details of systems parts other than his own [Parnas and Clements, 1986]."

Abstraction and set theory provides rigorous means to describe information hiding. The HAMSD model and the mathematical foundations of information hiding and abstraction are discussed in Sections 1.2.4 and 4.2, respectively.

2.3.3.4 Engineering Approach

Although the engineering approach to software development was first proposed by Baure in 1968 [Naure and Randell, 1969; Bauer, 1976], Parnas best expressed the need of software engineering and what the engineering characteristics and professionalism are [Parnas, 1971/72/78/95].

Definition 2.6 *Engineering approach* (PR4) is a software engineering principle that states software development and its organization should adopt proven generic engineering methodologies and practice.

Hoare believed *simplicity* and *elegance* are important characteristics of all engineering disciplines. Hoare proposed to maintain the criteria of simplicity in language and system design, and suggested to build only what is needed for a program [Hoare, 1973/80].

Parnas emphasizes that the focuses of software engineering theories and methodologies must always put on both ‘fundamental knowledge’ and ‘program design’ rather than specific techniques and language syntax [Parnas, 1971/72/96].

One of the discoveries on the nature of software engineering is that the problems of software engineering are an organizational issue, as well as the cognitive and resources issues. The major organizational issues in software engineering are work organization, optimal labor/time allocation, and division of labor. A comprehensive exploration of generic engineering methodologies and approaches will be presented in Chapter 8 on the engineering foundations of software engineering.

2.3.3.5 Professionalism

Professionalism is a common standard in engineering and many other sectors of the society, such as those of medical doctors, lawyers, and accountants.

Definition 2.7 *Professionalism* (PR5) is a software engineering principle that refers to the competence or skill expected for a professional software engineer who is formally trained and certified.

Professionalism of software engineering refers to the qualification, identity, and ethic of qualified software engineers. Professionalism of software engineering also refers to the professional and social responsibilities of software engineers and their effort for maintaining professional integrity and service quality to customers and the society.

Because of the importance of software in many systems and daily life in the modern information society, software engineers take much more responsibility in system design, implementation, and maintenance, as well as in technology development and evaluation. The ultimate objective of professionalism in software engineering is that the software engineers are capable to recommend a solution to customers that is cost-effective, simple, efficient, practical, satisfactory, reliable, and safe.

2.3.3.6 Tools and Environments

A tool is a device with a particular function for doing something. A tool in software engineering is a system or application software that is used to create, design, or implement other software. An environment for software engineering is an integrated set of supporting tools that cover multiple development processes of software engineering.

Definition 2.8 *Tools and environments* (PR6) are a software engineering principle that states software development tools and software engineering supporting environments are facilities that enable efficient organization of coordinative work or extend human physical and intelligent capability in software development.

Tools are crucial means in software engineering to improve both productivity and quality, as well as to extend the capability or lower the requirement for skills of software engineers. It is noteworthy that both tools and supporting environments for software engineering should be treated as a system that integrates a coherent set of certain functions rather than individual means.

Tools and supporting environments are used to extend human physical and/or intelligent capability. The latter is particularly needed in software engineering, such as those of dealing with the cognitive complexity, facilitating coordinative work, improving productivity, and saving resources and labor. Fundamental supporting environment for software engineering should focus on cognitive, organizational, and productivity.

2.3.3.7 Documentation

Documentation is written materials that serve as a record of information and evidence. Software engineering documentation encompasses not only source code, but also all intermediate work products toward the code and its validation and operation, such as memorials, contracts, design architectures and diagrams, reports, configurations, test cases, maintenance logs, design comments, and user manuals.

Definition 2.9 *Documentation* (PR7) is a software engineering principle that is used to embody system design and architectures, record work products, maintain traceability of serial decisions, log problems and maintenance solutions, and enable postmortem analysis.

The need for documentation is to express and embody the intangible and abstract design and implementation information, including all intermediate work products and decision rationales in software engineering that are usually created in software engineers' and managers' minds, explicitly described and formally recorded.

Therefore, documentation is required for all software engineering processes from requirements to system design, from specification to code, and from target configuration to maintenance. Emphases should be put on what Parnas stressed on software engineering documentation: "Design without documentation is not design [Parnas and Clements, 1986; Hoffman and Weiss, 2001]," and "If it is not documented, it is not done [Parnas, 1994b]."

2.3.3.8 Stepwise Refinement

Refinement is a process to improve or clarify a conceptual model or prototype by a series of deductive extensions or incremental development of details.

Definition 2.10 *Stepwise refinement* (PR8) is a software engineering principle for deductively extending a conceptual model of requirements for a given software system by a series of expatiations and incremental specifications at an increased degree of details.

Software design refinement can be carried out by a serial process of improved clarity starting from a conceptual model, in which each step results in an intermediate model that reveals greater degree of details of the architecture and behaviors of the system. The philosophy of refinement is deductive extension, where each of them is based on known principles, laws, and constraints for that particular step of refinement.

The incremental software development methodology is a special case of stepwise refinement. It is noteworthy that incremental development, as well as stepwise refinement, is a useful system design methodology, but not a system implementation technique. Stepwise refinement of system design is the strategy to deal with the great cognitive complexity in the specification and design phases of a system. However, once the design of the system is obtained, its implementation should be completed as customers require rather than done incrementally.

Usually, detailed system specifications may be obtained through three to four steps of refinements. Detailed descriptions of system refinement methodologies can be referred to Section 4.7. If stepwise refinement may be perceived as an incremental design or specification methodology, prototyping that will be introduced below can be perceived as an incremental implementation methodology in software engineering.

2.3.3.9 Prototyping

A *prototype* in engineering is a rough or preliminary model of an implementation based on which refined designs or improved models can be developed. *Prototyping* is an experimental process in which the design of a required system can be evaluated and validated via a prototype of the system.

Definition 2.11 *Prototyping* (PR9) is a software engineering principle for evaluating or validating a design and feasibility of a required system based on the implementation of a prototype of the system.

Prototyping is a common practice in almost all engineering disciplines. Prototyping is particularly important in software engineering because a prototype is an executable model that embodies abstract system specifications and design concepts. The focus of software engineering prototyping should be on evaluation and validation of the design on the basis of the system specifications. Therefore, a matured process of the design phase in software engineering must include requirement analysis, system specification, feasibility evaluation, system design, prototyping, and design evaluation, before a full scale implementation may be commenced.

Corollary 2.1 As a necessary and sufficient condition for full scale implementation of a software system, the *design phase* of software engineering shall be extended from *requirement analysis, system specification,* and *system design* to *feasibility evaluation, prototyping,* and *design evaluation*.

2.3.3.10 Adopting Engineering Notations

A *notation* is a set of symbols for representing attributes of real-world or abstract objects such as quantity, quality, characteristics, and classifications. Software engineering notations are formal and descriptive notations designed for the description and embodiment of intangible software architectures and behaviors.

Definition 2.12 *Adopting engineering notations* (PR10) is a software engineering principle for abstracting, denoting, and modeling of user requirements and system specifications expressively and explicitly.

Notations are the formal means of software modeling and design. They also facilitate top-down refinement of software systems from top-level designs to low-level details. This leads to the realization of automatic code generation based on explicitly refined system specifications.

Although, almost all engineering disciplines adopt blueprints, block diagrams, or schematic diagrams in system design and modeling, due to the abstract and intangible nature of the object under study, software engineering requires a descriptive notation system that must be application, method, and language independent. A recent development for such notations for software engineering is RTPA, which is developed based on cognitive informatics studies that help to elicit basic architectural and behavioral processes of software system [Wang, 2002a/02c/03c/06a]. RTPA encompasses a small set of mathematical notations for 17 meta processes and 17 process relations. However, this coherent set of notations is adequate to seamlessly model and describe system and human behaviors from top-level design to low-level implementations based on algebraic rules. RTPA will be described in Sections 4.5 through 4.8, as well as Section 6.6.

2.3.3.11 Process Modeling

A *process* is a series of actions toward a particular goal or a series of transitions toward a particular state. A *process model* is a formal description of the sequence of actions or transitions and their conditions.

Definition 2.13 *Process modeling* (PR11) is a software engineering principle for dealing with organizational and managerial issues in software engineering, as well as software behaviors.

Understanding the need to examine the software engineering process follows naturally from the premise that has been found to be true in other engineering disciplines, that is, that better products result from better processes. For the expanded domain of software engineering, the existing methodologies that cover individual subdomains are becoming inadequate. Therefore, an overarching approach is sought for a suitable theoretical and practical infrastructure to accommodate all the modern software engineering practices and requirements. An interesting approach, which is capable of accommodating the complete domain of software engineering, has been recognized and termed the “software engineering process”. Adoption of the software engineering process paradigm will enhance software engineering methodologies and techniques in the aspects of management and organization, and quality assurance.

RTPA can be used as a formal tool for process system modeling and description. In other words, software engineering processes can be formally treated and manipulated on the basis of algebraic rules.

2.3.3.12 Reuse

Reuse refers to using a software object more than once. It is perceived that design for reuse can improve programming efficiency and productivity in software engineering.

Definition 2.14 *Reuse* (PR12) is a software engineering principle for adopting higher-level building blocks, such as algorithms, methods, processes, patterns, frameworks, in order to improve efficiency, productivity, and quality of software engineering.

There is a whole spectrum of reuse opportunities from language statements up to system design notations. However, reuse is first a philosophy suggesting that a completely new development from scratch is anti-productive and always error-prone in software engineering. Reuse is a universal practice in all engineering disciplines.

In software engineering, reuse had been focused on code at a higher level than statements of programming languages. However, considering that software is a specific solution to a given application on a general hardware platform and every application is a one-off activity, code reuse will not be significantly beneficial to software engineering as the technology might promise, because there are not so many generic and universal software components available.

On the other hand, if a broad perception of reuse in software engineering is adopted from reuse of code to design patterns, frameworks, tests, processes, and documentation, there will be a wide range of applications of reuse. The constraints and bottleneck for software reuse beyond language statement level is not programmers' willingness but the standardization of highly reusable components. In software engineering, the majority of reusable functions and components have been already a part of different kinds of system software. Then, generic reusable components in application software are system frameworks, patterns, algorithms, processes, and documentation.

A recent progress in reuse methodologies for software engineering is the finding that tests can be reused as well as that of code via the techniques known as Built-in Tests (BITs) [Wang et al., 1997/98a/99d/2000]. BITs provide a new focus on reuse for both system and application software systems, especially for real-time and safety-critical systems.

2.3.3.13 Measurements and Metrics

A *measurement* can be perceived as a process to evaluate and quantify an attribute of physical or abstract object against a certain standard or a unit system. *Metrics* is a system of the standard of measurement.

Definition 2.15 *Measurements and metrics* (PR13) are a software engineering principle that is applied to elicit generic software attributes, quantify their measurement, and unify their metrics.

Measurement and metrics play an important role in quantitative software engineering, because where there is no measure, there is no control on both quality and productivity [Fenton, 1991; Melton, 1996; Zuse, 1997; Fenton and Pfleeger, 1997; Wang, 2001f/02d/02f]. However, software measurement and metrics have been overlooked in software engineering for decades, perhaps partially because of the immaturity of software measurement theories, and partially because the software industry has not been convinced with the benefits of *quantitative software engineering*. Nevertheless, the day of the maturity of software measurement theories and practices will be the day of the maturity of software engineering itself.

2.3.3.14 Cognitive Complexity Control

Complexity is an attribute of a physical or abstract system that represents the magnitude of its internal parts and the extent of their intricate connections. Complexity in nature is closely referred to the cognitive difficulty in comprehension and analysis of the system or objects.

Definition 2.16 *Cognitive complexity control* (PR14) is a software engineering principle for dealing with the innate difficulty in both architectural and behavioral design and implementation of software systems by a variety of means such as abstraction, modularization, descriptive notations, stepwise refinement, and prototyping.

As the scale of software increases continually and at an ever faster rate, greater complexity of software engineering becomes critical. Although, no language, tool, or technology may reduce the inherited complexity of a given problem, proven principles and methodologies can help system architects and programmers to comprehend, represent, and manipulate the cognitive complicity better in software engineering.

Except the technical complexity in software engineering, a whole set of cognitive difficulty lies in the areas of organization and management of large-scale software system development. It will be shown in Section 8.5 that a certain workload of a given software project may be amplified several even hundreds times due to improper organization and poor management. This can be considered the major reason that results in the failures of more than half of large software engineering projects with severely overspent budgets and/or overrun schedules [Schonberger, 1981; Jones, 1996; Wang, 2006c; Wang and King, 2000a; Wang and Shao, 2003; Shao and Wang, 2003].

2.3.3.15 Formal Requirement Specification

A *requirement* is a need or wish for a certain object or a function for a particular purpose and with given conditions. A *specification* is a precise statement of a requirement for identifying its attributes and functions, describing its design and implementation approaches, and proving evaluation standards.

Definition 2.17 *Formal requirement specification* (PR15) is a software engineering principle that states customers' nonprofessional requirements for a software system should be formally and rigorously specified in the development team in order to avoid any misinterpretation and ambiguity, and to eliminate any conceptual gap and inconsistency.

Formal system specification helps to identify conceptual gaps about the perceived system in customer's requirements, between the development team and the customers, and among team members. It is also a perfect means for documenting designs of software systems that is independent of methodologies, languages, and implementation techniques.

Because the major difficulty in early phase design in software engineering is the cognitive complexity, a rush into coding by a large group of programmers may create more problems than those of solved, which may dramatically increase the real workload for a given project. Instead, the payoff of a formally specified system approach will be seen in the implementation phase by an improved productivity, shorter duration, and predictable quality.

2.3.3.16 Systematic Quality Assurance

Quality is a distinctive attribute or the extent of excellence of an object against a certain standard. *Quality assurance* is the maintenance of a required level of quality for a given object by systematical controls and evaluations throughout all processes.

Definition 2.18 *Systematic quality assurance* (PR16) is a software engineering principle that states software quality is multiple faceted; therefore a systematic tackle is needed on all attributes of software quality and their quantitative measurement.

The study on software quality and quality assurance is a particularly weak area in software engineering and the software industry. The vague perception on the nature of software quality, the lack of quantitative measures for software quality, and the stranding to software quality assurance indicate that software quality is a theoretical problem rather than a technical one yet to be solved in software engineering. A rigorous description of software quality and quality assurance as a system problem will be presented in Section 11.4.

2.3.3.17 Review and Inspection

Review is a process to assess a given object by carefully and critical readings of peers or more experienced experts. *Inspection* is a process to examine if a given object confirms with a certain standard or requirement.

Definition 2.19 *Review and inspection* (PR17) is a software engineering principle for finding and eliminating software design and implementation defects via reading and examining the work products by peer or more experienced reviewers.

Review and inspection have been found effective and useful in design phase of software development rather than implementation phase, because in the early phase there are no other verification techniques or tools available, and no thorough testing can be carried out before the code is ready. Weinberg believed that review and inspection are an indispensable part of engineering high quality software, because no matter how smart a programmer is, reviews will be beneficial [McConnell, 1999]. Watts Humphrey proposed that review should be a mandatory process before code compiling and testing [Humphrey, 1995/96].

The theory behind the empirical usage of review and inspection in software engineering is based on the theory of the *randomness of human errors* in performing tasks [Wang, 2005f]. Because different programmers are unlikely to make the same mistake for the same task at the same time in software development, review and inspection are entitled to find errors and eliminate bugs efficiently. Section 13.5.3 will formally prove that review and inspection are effective techniques dealing with creative work products such as software systems and related documents.

2.3.3.18 Management Engineering

Management is a process to deal with coordinated work and how people and resources may be optimally allocated on the work.

Definition 2.20 *Management engineering* (PR18) is a software engineering principle that states a crucial facet of software engineering is the need for a suitable theory for organizing and coordinating large groups in large-scale projects.

As the scale of software increases continually, the complexity of the problem grows at an ever faster rate. In contemporary software engineering, the central role is no longer that of the programmers; project managers and corporate management have critical roles to play. As programmers use programming technologies, software corporation managers seek organizational and strategic management methodologies, and project managers seek professional management and software quality assurance methodologies. These developments have resulted in an expanded domain of software engineering, which includes three important aspects: development methodology, organization and infrastructure, and management [Wang and King, 2000a].

An interesting discovery in a recent survey on the international curricula of software engineering is that project management is the most popular course commonly offered in almost all universities worldwide [Wang and Liu, 2004]. Management theories and methodologies are an important facet of the theoretical and empirical framework of software engineering. A formal treatment of the coordinative work organization theory and the management foundations of software engineering will be presented in Chapters 8 and 11, respectively.

2.3.3.19 Acquiring Domain Knowledge

Domain knowledge is the knowledge about application areas, the nature of categories of problems, and the environment and context in which a given problem is encountered, and conventional customer practice for dealing with the problem.

Definition 2.21 *Acquiring domain knowledge* (PR19) is a software engineering principle that states four aspects of domain knowledge, such as (a) the nature of the problem, (b) the environment and context of the problem, (c) current customer practice for dealing with the problem, and (d) existing regulations and constraints in the application area, should be acquired before a system design for the given problem may proceed.

The approaches to obtain essential domain knowledge are as follows:

- To observe how the customer deals with the problem traditionally.
- To understand the environment of the problem and processes preceding and following the given problem.
- To be aware of any regulations, standards, and constraints related to the given problem.

- To survey alternative practices, best practices, and domain norms on the given problem.
- To search possible existing solutions in the literature.

The need for domain knowledge in software engineering is a necessary condition to be able to design a professional system that best suits customers' requirements and environment. This need leads to the following principle of software engineering known as customer involvement, and it also results in a new role in the software industry called domain engineers who are specialized in one or more application domains and familiar with typical software solutions for problems in these domains.

2.3.3.20 Customer Involvement

Customer involvement is a key to success in software engineering. *Customers* are representatives of a project and/or end users of a software system, who should be involved in all processes of system development. In case the customer and user of a certain system are different, the target users have to be identified clearly.

Definition 2.22 *Customer involvement* (PR20) is a software engineering principle that states all stakeholders, particularly the end users of a software system, should be involved throughout the entire lifecycle of the system by customer reviews and joint meetings.

Regular joint meetings and collection of customer review feedback are two useful techniques that enable customers to be involved in the development process of software system and feel the growth of the system well before adopting it in practice.

It is noteworthy that the successful acceptation and adaptation of a new system is not only dependent on technical performance, but also dependent on users' attitudes toward the system. Since the introduction of a new system requires considerable working behavioral changes even cultural changes in an organization, training of users and orientation of related stakeholders are a crucial process for the success of newly developed systems.

2.3.3.21 Feasibility Analysis

Feasibility is the extent of possibility or practicality to carry out a task within the given constraints. There are technical and economical feasibilities for any given task or project. The former refers to the feasibility that is constrained by technology availability and adequacy, while the latter is determined by the analysis of the benefit-cost ratio of a given project.

Definition 2.23 *Feasibility analysis* (PR21) is a software engineering principle that states both the technical and economical feasibilities of a given software project should be rigorously estimated and evaluated before the later-phase processes may be continued.

According to Corollary 2.1, feasibility analysis should be a mandate process between system specification and detailed system design. The rule of thumb in software engineering is that both technical and economical feasibilities should be evaluated before a software project may proceed. The technical feasibility of a given software engineering project is addressed through rigorous modeling methodologies in this book. The economic feasibility in software engineering is discussed in Section 12.6 on economic analyses of software projects.

2.3.3.22 Comprehensibility

Comprehension is a process to understand or a capability for understanding. *Comprehensibility* is the degree of understanding or cognitive capability about a particular object or issue.

Definition 2.24 *Improving comprehensibility* (PR22) is a software engineering principle for explicitly and expressively describing the intangible problem and its solution with improved understandability, readability, and cognitive capability.

The natural strategies for dealing with the intangible and abstract objects under study in software engineering are explicit description and facilitating comprehensibility, which include readability, cognitive complexity, and intellectual manageability about the problem and its software solution. Therefore, the emphases of software engineering theories and techniques should be put on supporting human comprehensibility in dealing with large-scale and extremely complicated software systems. Software engineering notations and explicit documentation methodologies are some typical means for improving comprehensibility of software.

It is noteworthy that the requirement for comprehensibility covers all work products in software engineering including design specification, code, decision making records, documentation, and testing cases and results.

The cognitive process of comprehension will be formally described in Section 9.5. A formal software engineering notation system, RTPA, will be presented in Sections 4.5 through 4.8 (notations and methodologies), Sections 5.3 through 5.6 (usage), and Section 6.6 (deductive semantics).

2.3.3.23 Exception Handling

An *exception* is an unusual event or behavior that is not expected according to a given rule or norm.

Definition 2.25 *Exception handling* (PR23) is a software engineering principle that states system design and specification should consider not only customer required functions for a given system, but also all possible exceptions that may drive the system into illegal state(s) in the entire state space of the system.

The size of the state space of a program is determined by a Cartesian product between the number of possible states and the number of possible events. However, the required or legal states of functions for a given system are usually a small portion of the whole space. The remainders are nonrequired or illegal states. Customers of a given system just require the desired and legal functions of the system, but the job of system analysts and architects is to identify the whole state space of the system and predicate what would happen if the system enters an illegal state by any reason such as external interferences, data distortions, human mistakes, or hardware malfunctions. Therefore, in a certain extent for professional architects, system designs are meant not only to consider the required functionality but also to prevent what would go wrong in the given system setting.

Exception handling is first to identify all possible illegal transitions in the entire state space of the system and then prevent them from happening. An exception handling strategy and process should be designed for each exception or each category of equivalent exceptions.

2.3.3.24 Divide and Conquer

Divide-and-conquer is an analytic strategy of system design based on reductionism. In software engineering, divide-and-conquer, functional decomposition, and modularization can all be perceived as strategies for dealing with the cognitive complexity of software systems.

Definition 2.26 *Divide-and-conquer* (PR24) is a software engineering principle that supposes if a complex system may be divided into multiple components, the individual components of the system will be easier to be dealt with than the whole system.

Empirically, to directly solve a large problem is often very difficult and complicated. However, if the problem can be broken up and partitioned into a set of smaller sub-problems, and the sub-problems are usually easier than the original problem, then the problem may be solved individually. This is the philosophy of divide-and-conquer.

As that of the abstraction principle, divide-and-conquer is another key to reduce complexity in software engineering. The techniques for conducting divide-and-conquer in software engineering are modularization and decomposition. It is noteworthy that the essence of the principle of divide-and-conquer is not 'to divide' but 'to conquer.' Therefore the architecture and structure of the decomposed components are the core in any technology that supports system modularization.

When applying the principles of divide-and-conquer, a practical question is how many sub-components should be divided. A heuristic rule is that when the given system is divided into a set of similar sized sub-systems, an optimistic solution would be reached. Especially, when the sub-systems are divided further in the same way, a hierarchical tree structure may be derived by recursively applying the divide-and-conquer principle. The heuristic rule indicates that the best way in modularization is to divide the modules, components, or subsystems into similar and balanced sizes. More formal treatment of this heuristic rule will be provided in Section 10.3 on system topology.

2.3.3.25 Explicit Embodiment

Embodiment is a representation or expression of an abstract object, such as an idea, concept, or feeling, in a tangible or visible form.

Definition 2.27 *Explicit embodiment* (PR25) is a software engineering principle for dealing with the implicitness and inexpressiveness in software engineering by introducing more powerful descriptive means at a higher level of abstraction and precision.

According to the principle of *explicit descriptivity* as given in Theorem 1.3, only a higher level of more abstract and more precise means is adequate and sufficient to express and embody an object at a certain level of abstraction.

Because of the nature of software, architectures are complicatedly interrelated objects with functional variables and constraints, and behaviors are embedded relational processes. These types of abstract and complicated entities may only be expressed without implication and ambiguity by professional notation systems, because only more abstract and precise means is powerful enough to express an object at a given level of abstraction. Therefore, symbolic notations are the key means for expressing and embodying software visualization. A form of denotational mathematics for describing software engineering work products, RTPA, will be described in Chapters 4, 5, and 6.

2.3.3.26 Establishing Theoretical Foundations

A *foundation* is a principle that forms an underlying basis for deriving new knowledge and for supporting rigorous reasoning. A theoretical foundation is a set of formally described foundations derived by rigorous inductive inferences and proven true universally.

Definition 2.28 *Establishing theoretical foundations* (PR26) is a software engineering principle that states rigorous theories and generic laws should be elicited once there are a wide variety of observed phenomena and alternative practices.

The lack of theoretical foundations in software engineering is an essential deficiency for software engineering to be claimed as a matured engineering discipline. Theories in nature are abstract, generic, and mathematically rigor. Software engineering theories and their foundations on the basis of mathematics, particularly denotational mathematics such as logic and process algebra, allow reasoning about the work products in software engineering before they are built, and the optimal organization of large development projects and cooperative human creative work.

This book is devoted to seek the fundamental theories and suitable mathematical means for software engineering. The foundations of software engineering will be systematically established in the remainder of this book on those of philosophy, mathematics, computing, linguistics, information science, cognitive informatics, system science, management science, economics, sociology, and engineering science.

2.3.3.27 Architecture and Behavior Modeling

Modeling is a process to represent complicated objects by systematical, visualized, procedural, or denotative means.

Definition 2.29 *Architecture and behavior modeling* (PR27) is a software engineering principle that states software system models are a hybrid model where both architectures and behaviors should be coherently described.

In software engineering, modeling is focused on the architectures and behaviors of software systems. The former refers to the abstract models of the data objects and their logical structures, relations, and constraints; the latter are computational and interactive operations onto the architectural model and their interacting environments.

RTPA provides a denotative mathematical means for modeling software system architectures and their static and dynamic behaviors via a top-down refinement scheme. Based on the explicit and rigorous models specified in RTPA, corresponding code in a desired programming language can be generated automatically using RTPA supporting tools as extensively described in Chapters 4, 5, and 6.

2.3.3.28 Standardization

Standardization is a process to establish and quantify measures, norms, or models for enabling comparative evaluation of work products or services. Standardization is not only useful in societies and everyday life, but also particularly important and widely applied in engineering for documenting common factors and best practices, and synchronizing individuals and systems behaviors.

Definition 2.30 *Standardization* (PR28) is a software engineering principle for attempting to integrate, regulate, and optimize existing principles and best practices in research and in the industry.

Standardization provides a metric, norm, or benchmark as the standard for a certain attribute of a category of objects. Then, measurement may be carried out based on the standard.

Software engineering standards are not only records of best practices, but also vehicles for reconciling successful practices with the underlying principles of the profession. In software engineering, a large portion of cognitive, technical, and organizational practices are widely optional or even arbitrary when they are conducted in an isolated environment. However, when they are designed and applied for public uses, standardization becomes necessary. Because of the widely optional feature of design and implementation in the software industry, the first system of a kind in the market may become a *de facto* standard.

A comprehensive review of software engineering standards and international effort in software engineering standardization will be provided in Section 8.6.5 [Wang, 2001b].

2.3.3.29 Systems Engineering

A *system* is a complex whole of interacting components toward a particular goal. *System engineering* is the application of system science that adopts a systematical view to treat complicated objects and their interactions with the external environment.

Definition 2.31 *Systems engineering* (PR29) is a software engineering principle that states system science theories and methodologies should be adopted to deal with complicated architectures and behaviors of software.

One facet of software as explored in Section 1.5.8 is that it can be perceived as a system. The design and implementation of both architectures and behaviors of software are a system issue in software engineering. A rigorous description of system theories by system algebra [Wang, 2006d] and system science foundations of software engineering will be presented in Chapter 10. An important finding according to the formal system model is that the complexity of a system is on the order of $O(n^2)$ in general, where n is the number of components or objects in the system [Wang, 2006c/06d].

2.3.3.30 Engineering Organization

Organization is a process to systematically and efficiently coordinate human activities and interactions for a given work or social event.

Definition 2.32 *Engineering organization* (PR30) is a software engineering principle that states the coordinative work organization theory should be adopted in order to optimize team, project, and enterprise organizations.

It is identified that an essential facet of the problems in software engineering is an organizational issue, which is as equally important as those of the cognitive and technical ones. Therefore, organizational theories and management methodologies can play an important role in software engineering.

A formal treatment of generic engineering methodologies and the exploration of the engineering foundations of software engineering will be presented in Chapter 8, particularly the *coordinative work organization theory* [Wang, 2007d]. Applications of the organization theories will be discussed in Section 8.5 at the project level and Sections 13.4 and 13.5 at the society level.

2.3.3.31 Cognitive Engineering

Cognition is a knowledge acquisition process to understand the external world via sensation, perception, and reasoning. *Cognitive engineering* is the application of cognitive informatics in explaining and solving engineering problems where human beings are involved as part of the system or the problems.

Definition 2.33 *Cognitive engineering* (PR31) is a software engineering principle that states the cognitive complexity and human intelligent manageability should be addressed as the dominant problem in almost all processes of software design, implementation, and maintenance.

As described in Section 1.3, a large portion of the software engineering problems can be classified as a cognitive issue and is constrained by human cognitive capability and manageability of complexity for given problems. Formal descriptions of cognitive informatics foundations of software engineering will be presented in Chapter 9.

2.4 Software Engineering Principles as Measures to its Constraints

In Section 2.3 a comprehensive set of 31 fundamental principles of software engineering has been obtained by eliciting the common core proposals of a number of software scientists and institutions. Based on this, a unified framework of software engineering principles is established.

Principles are powerful means for facilitating deductive reasoning. This section examines the relationships between the basic constraints and fundamental principles of software engineering. The comprehensive set of basic principles of software engineering will be treated as the fundamental measures for coping with the basic constraints of software engineering. According to the discussions in Section 1.3, the 14 basic software engineering constraints can be classified into three categories known as the cognitive, organizational, and resources constraints as shown in Fig. 2.2. The applications of the 31 fundamental principles to each category of these basic constraints will be explored in the following subsections.

2.4.1 PRINCIPLES FOR COPING WITH THE COGNITIVE CONSTRAINTS

The first set of the basic software engineering constraints as shown in Fig. 2.3 is the cognitive constraints. Many fundamental principles elicited in Section 2.3 are suitable to deal with these cognitive constraints in software engineering. A mapping of the fundamental principles into the basic cognitive constraints is shown in Figs. 2.3 and 2.4, respectively.

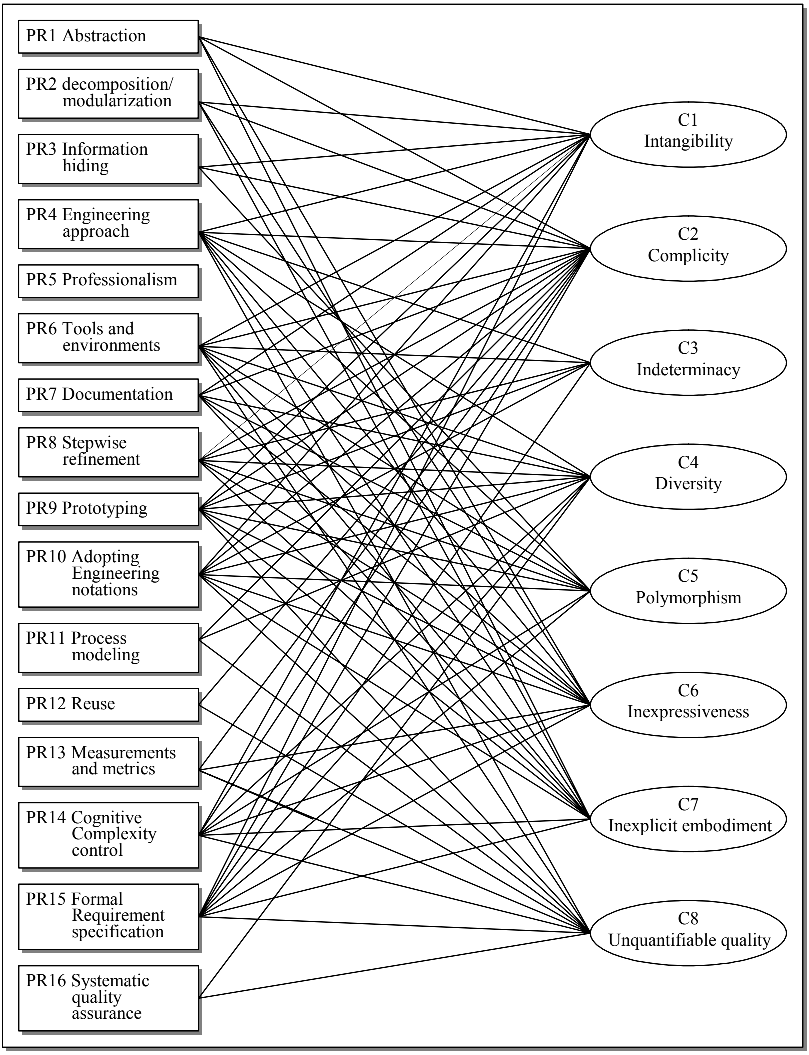


Figure 2.3 The software engineering principles vs. the cognitive constraints (I)

It can be observed in Figs. 2.3 and 2.4 that multiple principles may be applied to tackle a specific problem and constraint in software engineering. For instance, 27 principles are applicable to deal with the cognitive constraint, such as abstraction, modularization, information hiding, divide-and-conquer, modularization, stepwise refinement, prototyping, and decomposition.

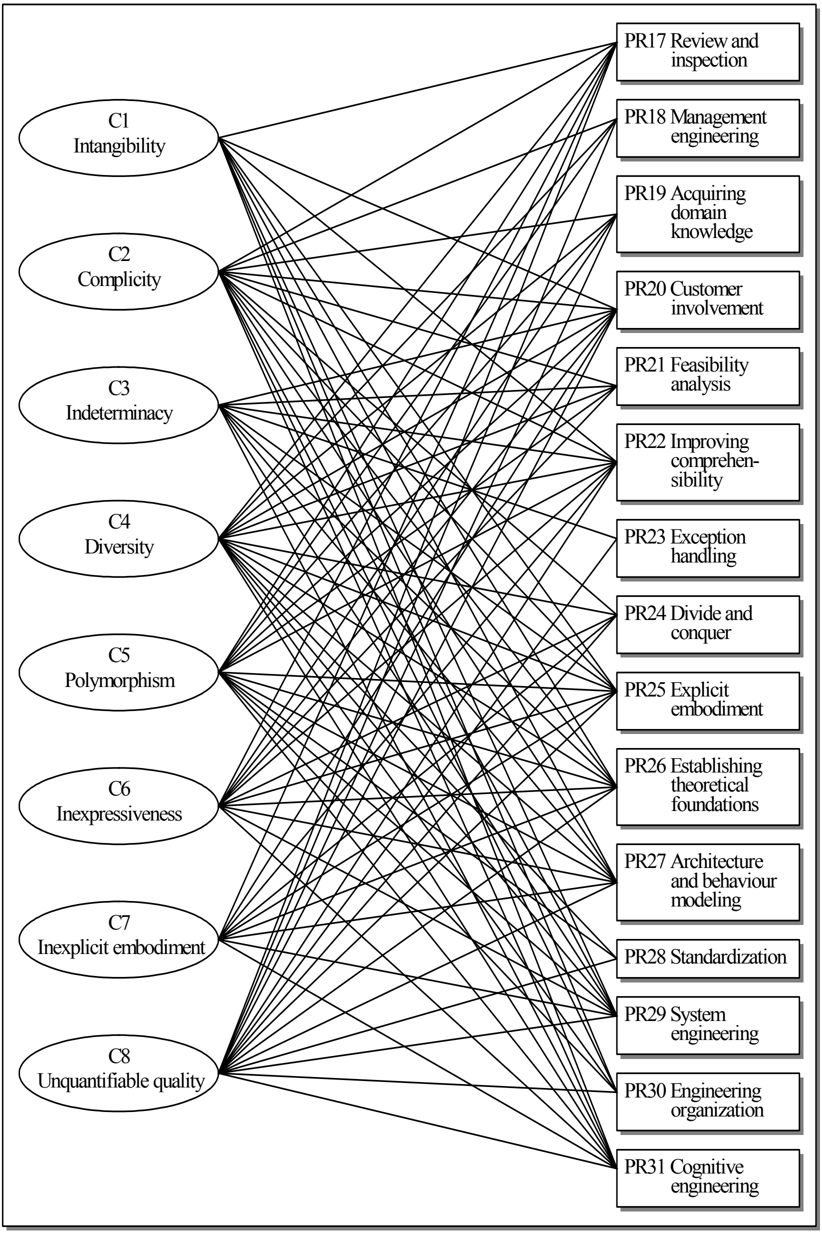

C1
Intangibility
C2
Complicity
C3
Indeterminacy
C4
Diversity
C5
Polymorphism
C6
Inexpressiveness
C7
Inexplicit embodiment
C8
Unquantifiable quality
PR17 Review and inspection
PR18 Management engineering
PR19 Acquiring domain knowledge
PR20 Customer involvement
PR21 Feasibility analysis
PR22 Improving comprehen-sibility
PR23 Exception handling
PR24 Divide and conquer
PR25 Explicit embodiment
PR26 Establishing theoretical foundations
PR27 Architecture and behaviour modeling
PR28 Standardization
PR29 System engineering
PR30 Engineering organization
PR31 Cognitive engineering

Figure 2.4 The software engineering principles vs. the cognitive constraints (II)

2.4.2 PRINCIPLES FOR COPING WITH THE ORGANIZATIONAL CONSTRAINTS

The second set of the basic software engineering constraints as shown in Fig. 2.2 is the organizational constraints. A mapping of the fundamental principles into the basic organizational constraints of software engineering is shown in Fig. 2.5.

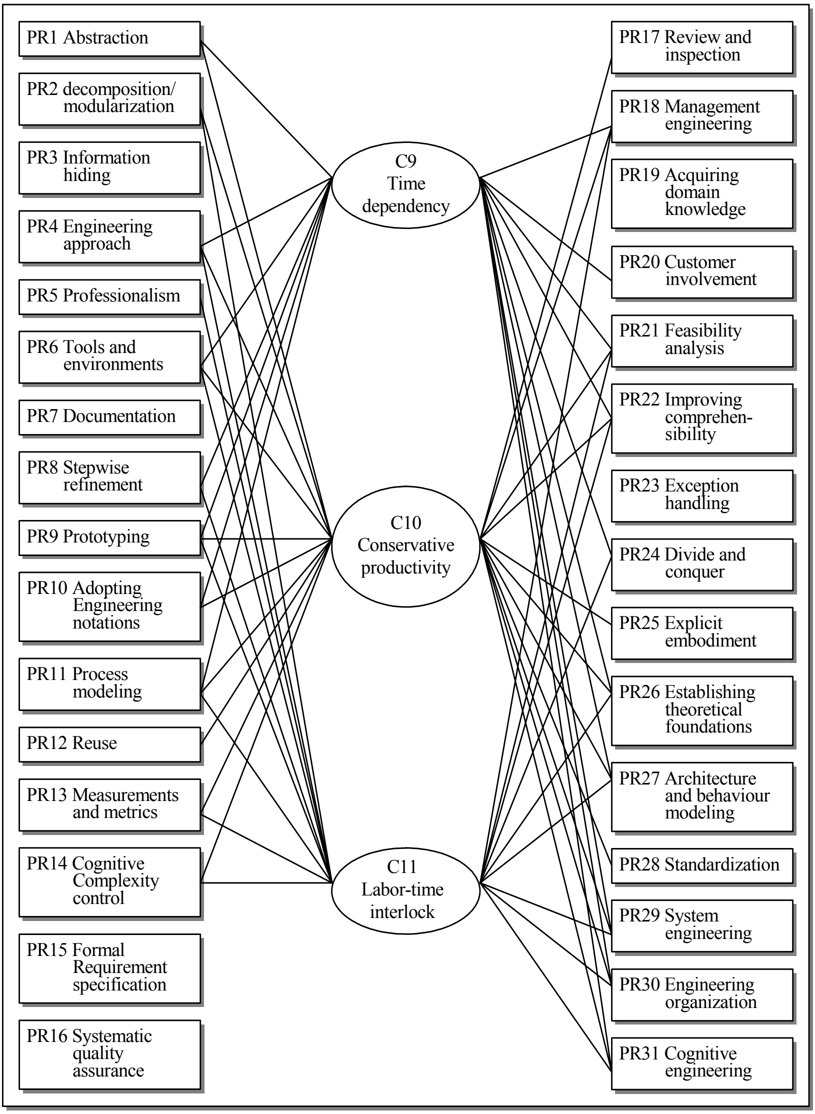

PR1 Abstraction
PR2 decomposition/ modularization
PR3 Information hiding
PR4 Engineering approach
PR5 Professionalism
PR6 Tools and environments
PR7 Documentation
PR8 Stepwise refinement
PR9 Prototyping
PR10 Adopting Engineering notations
PR11 Process modeling
PR12 Reuse
PR13 Measurements and metrics
PR14 Cognitive Complexity control
PR15 Formal Requirement specification
PR16 Systematic quality assurance
C9 Time dependency
C10 Conservative productivity
C11 Labor-time interlock
PR17 Review and inspection
PR18 Management engineering
PR19 Acquiring domain knowledge
PR20 Customer involvement
PR21 Feasibility analysis
PR22 Improving comprehen-sibility
PR23 Exception handling
PR24 Divide and conquer
PR25 Explicit embodiment
PR26 Establishing theoretical foundations
PR27 Architecture and behaviour modeling
PR28 Standardization
PR29 System engineering
PR30 Engineering organization
PR31 Cognitive engineering

Figure 2.5 The software engineering principles vs. the organizational constraints

2.4.3 PRINCIPLES FOR COPING WITH THE RESOURCE CONSTRAINTS

The third set of the basic software engineering constraints as shown in Fig. 2.2 is the resource constraints. A mapping of the fundamental principles into the basic resources constraints of software engineering is shown in Fig. 2.6.

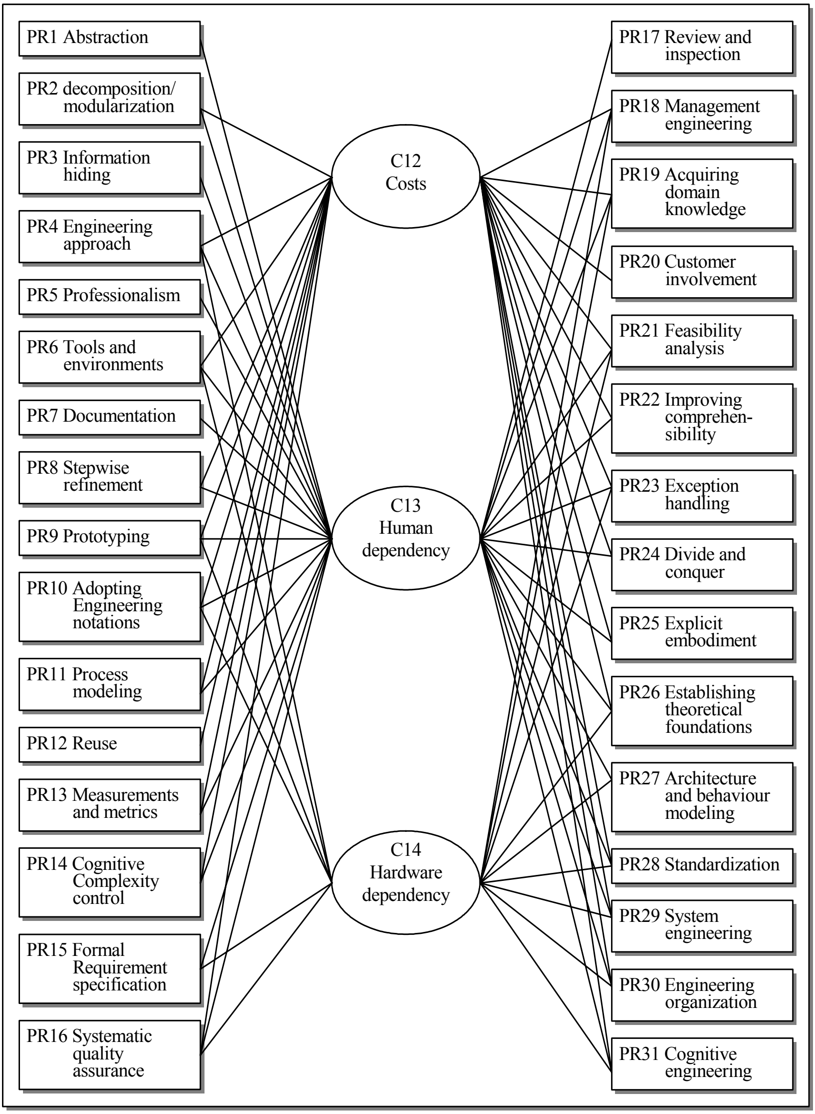
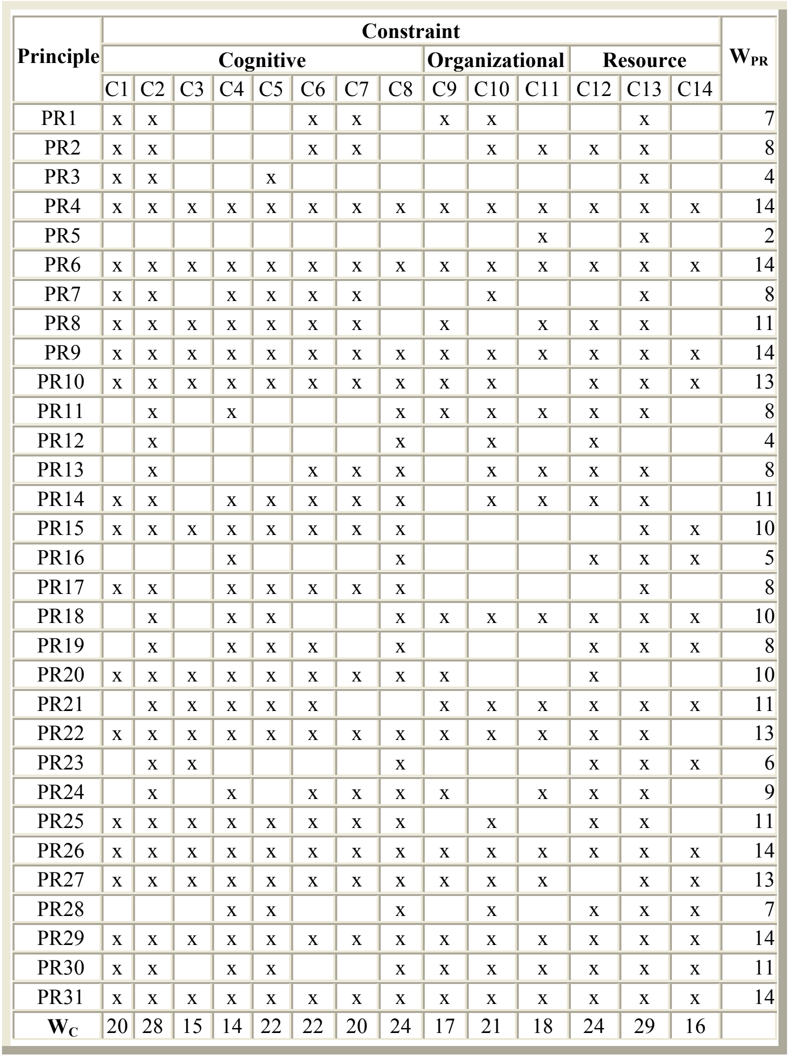

PR1 Abstraction
PR2 decomposition/ modularization
PR3 Information hiding
PR4 Engineering approach
PR5 Professionalism
PR6 Tools and environments
PR7 Documentation
PR8 Stepwise refinement
PR9 Prototyping
PR10 Adopting Engineering notations
PR11 Process modeling
PR12 Reuse
PR13 Measurements and metrics
PR14 Cognitive Complexity control
PR15 Formal Requirement specification
PR16 Systematic quality assurance
C12 Costs
C13 Human dependency
C14 Hardware dependency
PR17 Review and inspection
PR18 Management engineering
PR19 Acquiring domain knowledge
PR20 Customer involvement
PR21 Feasibility analysis
PR22 Improving comprehen-sibility
PR23 Exception handling
PR24 Divide and conquer
PR25 Explicit embodiment
PR26 Establishing theoretical foundations
PR27 Architecture and behaviour modeling
PR28 Standardization
PR29 System engineering
PR30 Engineering organization
PR31 Cognitive engineering

Figure 2.6 The software engineering principles vs. the resource constraints

2.4.4 A SYSTEMATIC VIEW ON MAPPING BETWEEN THE PRINCIPLES AND CONSTRAINTS

So far the basic problems and fundamental methodologies of software engineering have been modeled by the 14 constraints and the 31 principles, respectively. A general view between the principles and constraints of software engineering can be represented by a matrix as shown in Table 2.3.

Table 2.3
Mapping Software Engineering Principles into its Constraints



Principle	Constraint														W_{PR}
	Cognitive								Organizational			Resource			
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	
PR1	x	x				x	x		x	x			x		7
PR2	x	x				x	x			x	x	x	x		8
PR3	x	x			x								x		4
PR4	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
PR5											x		x		2
PR6	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
PR7	x	x		x	x	x	x			x			x		8
PR8	x	x	x	x	x	x	x		x		x	x	x		11
PR9	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
PR10	x	x	x	x	x	x	x	x	x	x		x	x	x	13
PR11		x		x				x	x	x	x	x	x		8
PR12		x						x		x		x			4
PR13		x				x	x	x		x	x	x	x		8
PR14	x	x		x	x	x	x	x		x	x	x	x		11
PR15	x	x	x	x	x	x	x	x					x	x	10
PR16				x				x				x	x	x	5
PR17	x	x		x	x	x	x	x					x		8
PR18		x		x	x			x	x	x	x	x	x	x	10
PR19		x		x	x	x		x				x	x	x	8
PR20	x	x	x	x	x	x	x	x	x			x			10
PR21		x	x	x	x	x			x	x	x	x	x	x	11
PR22	x	x	x	x	x	x	x	x	x	x	x	x	x		13
PR23		x	x					x				x	x	x	6
PR24		x		x		x	x	x	x		x	x	x		9
PR25	x	x	x	x	x	x	x	x		x		x	x		11
PR26	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
PR27	x	x	x	x	x	x	x	x	x	x	x		x	x	13
PR28				x	x			x		x		x	x	x	7
PR29	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
PR30	x	x		x	x			x	x	x	x	x	x	x	11
PR31	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
W_C	20	28	15	14	22	22	20	24	17	21	18	24	29	16	

The mapping between the software engineering constraints and measures as shown in Table 2.3 can be used as a guideline for allocating certain methodologies for coping with a given problem in a software engineering project. Table 2.3 can also be used to seek new theories and principles for software engineering.

Analyzing the mapping between software engineering principles and constraints, it may be found that some of the principles are more fundamental or useful in software engineering, because they deal with more basic constraints in software engineering problem solving.

In Table 2.3, the right-most column provides the *weights of usage coverage* for each principle W_{PR}. The top six most widely applicable principles in software engineering, which may be used to deal with all the basic problems, are as follows:

- PR4 Engineering approach
- PR6 Tools and environments
- PR9 Prototyping
- PR26 Establishing theoretical foundations
- PR29 Systems engineering
- PR31 Cognitive engineering

Following the above list, a set of very useful principles of software engineering, which cover more than ten basic problems, is identified below in the order according to their weights of coverage:

- PR10 Adopting engineering notations
- PR22 Comprehensibility
- PR27 Architecture and behavioral modeling

- PR8 Stepwise refinement
- PR14 Cognitive complexity control
- PR21 Feasibility analysis
- PR25 Explicit embodiment
- PR30 Engineering organization

- PR15 Formal requirement specification
- PR18 Management engineering
- PR20 Customer involvement

Generally, it may be seen from Table 2.3 that the newly identified principles in this book labeled PR25 through PR31 are more effective than those of the conventional ones.

In Table 2.3, the data shown in the bottom row indicate the *weight of methodology coverage* W_C, or how many principles and methodologies of software engineering have been focused on each of the basic constraints. According to the weights of methodology coverage, the constraints and problems in software engineering are covered by multiple principles in the following order: C13 – human dependency, C2 – complexity, C8 – unquantifiable quality measures, C12 – costs, C5- polymorphism, C6 – inexpressiveness, C10 – conservative productivity, C1 – intangibility, C7 – inexplicit embodiment, C11 – labor-time interlock, C9 – time dependency, C14 – hardware dependency, C3 – indeterminacy, and C4 – diversity. It may also be interpreted that the last few problems are tougher to be dealt with because there are fewer methodologies covering them.

The above list also shows the inadequacy of current principles and methodologies for software engineering, because most software engineering principles identified so far are empirical and heuristic. Toward the maturity of a software engineering discipline, there is still a need to seek the theoretical foundations and laws of these fundamental principles, and their rigorous description and empirical studies in software engineering.

The 2nd Principle of Software Engineering

Theorem 2.1 *Formalization of principles* states that the empirical principles for software engineering are heuristic and data-based; while the formal principles for software engineering are rigorous and mathematics-based, which are elicited and refined from the empirical principles.

The development of coherent software engineering theories, methodologies, and techniques should put emphases on these tough challenges and problems in software engineering. To some extent, this is one of the main motivations and purposes of this book in the remaining chapters. By putting together all the principles as well as theoretical and empirical foundations, adequate and sufficient theories and methodologies for software engineering will be developed systematically throughout the book.

2.5 Summary

The **principles of software engineering** are the essential knowledge that a software engineer needs to know in order to develop software scientifically and effectively. A **principle** is a generic theorem, rule, or law of a theory that can be applied to a wide range of cases or instances in a field of study. A principle serves as a fundamental predicate for logical reasoning and deduction.

Software engineering principles are a set of fundamental and coherent theorems and laws that constrain the behaviours of software systems and the processes of their development.

This chapter has attempted to elicit a coherent set of fundamental principles of software engineering. The major pioneer pursuits of principles for software engineering in the last four decades have been reviewed, which provide a whole picture for understanding the fundamental theories and foundations of software engineering. **A unified framework of software engineering principles** has been established with a comprehensive set of 31 commonly identified fundamental principles. These fundamental principles of software engineering have been treated as powerful measures to tackle the 14 basic constraints of software engineering as identified in Chapter 1. As a result, the **unified framework of software engineering principles** has been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

The theme of this chapter is on *fundamental principles of software engineering* and their relations with the basic constraints. Through this chapter, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 2. Principles of Software Engineering

■ Pioneer Pursuits of Principles for Software Engineering

- Parnas' principles of software engineering
  - *Information hiding, modularization, engineering approach, professional responsibility,* and *documentation*
- Hoare's principles of software engineering
  - *Professionalism, vigilance, sound theoretical knowledge, using tools, abstraction, structured programming,* and *readability*
- Brooks' principles of software engineering
  - *Complexity, conformity, changeability,* and *invisibility*
- Wasserman's principles of software engineering
  - *Abstraction, method and notation, prototyping, modularity and architecture, lifecycle and process, reuse, metrics, tools and integrated environments*

- IEEE SESC's principles of software engineering
  - *Quantitative measurements, reuse, control complexity, rigorous specification, software process, disciplined approach, understanding the problem, management of quality, minimize components coupling, stepwise development, specify quality, objectives, change management, specify tradeoffs, domain knowledge,* and *uncertainty management*
- IEEE Software's principles of software engineering
  - *Reviews and inspections, information hiding, incremental development, user involvement, automated revision control, Internet development, programming languages hall of fame, Capacity Maturity Model* (CMM), *object-oriented programming, component-based programming, metrics and measurement*

■ A Unified Framework of Software Engineering Principles

- Elicitation of fundamental principles of software engineering
- The unified framework of software engineering principles
- Description of the fundamental principles of software engineering
  - Abstraction
  - Decomposition/modularization
  - Information hiding
  - Engineering approach
  - Professionalism
  - Tools and environments
  - Documentation
  - Stepwise refinement
  - Prototyping
  - Adopting engineering notations
  - Process modeling
  - Reuse
  - Measurement and metrics
  - Cognitive complexity control
  - Formal requirement specification
  - Systematic quality assurance
  - Review and inspection
  - Management engineering
  - Acquiring domain knowledge
  - Customer involvement
  - Feasibility analysis
  - Improving comprehensibility
  - Exception handling
  - Divide and conquer
  - Explicit embodiment

- Establishing theoretical foundations
- Architecture and behavior modeling
- Standardization
- Systems engineering
- Engineering organization
- Cognitive engineering

■ Software Engineering Principles as Measures to its Constraints

- Principles for coping with the cognitive constraints
- Principles for coping with the organizational constraints
- Principles for coping with the resources constraints
- A systematic view on mapping between the principles and constraints

SIGNIFICANT FINDINGS OF THIS CHAPTER

- **The pursuits of fundamental principles** for software engineering can be traced back to the 1950s by pioneers such as Davis L. Parnas, C.A.R. Hoare, Edsger W. Dijkstra, Friedrich L. Bauer, Frederick P. Brooks, and Barry Boehm. Software engineering principles form the essential knowledge that a software engineer needs to know in order to develop software scientifically and effectively.

- **Theories vs. Technologies:** Although software development technologies have been changing from time to time, the fundamental principles of software engineering have remained constant as the crystallization of theories and methodologies over a long period of time.

- The **relationship** between the fundamental principles and basic constraints of software engineering is a complicated network. A main thread to analyze their relations is to perceive the constraints are the problems, and the principles are the measures to tackle the problems. On the basis of this thread, a mapping between the 31 principles and the 14 constraints is presented in Fig. 2.2. A detailed mapping of the principles of software engineering into its constraints is summarized in Table 2.3.

- Principles can be classified into two categories known as the **formal** and **empirical/heuristic** principles. Most known principles in software engineering are empirical and heuristic. For supporting rigorous reasoning and decision making in software engineering, formalization of those empirical principles seems profoundly important.

- The **most widely applicable principles** of software engineering, which may be used to deal with almost all the basic constraints and problems, are: PR4 – Engineering approach, PR6 – Tools and environments, PR9 – Prototyping, PR26 – Establishing theoretical foundations, PR29 – Systems engineering, and PR31 – Cognitive engineering.

- **Newly identified principles**, based on the recent studies on the nature of software and software engineering, are architectural and behavioral modeling, system engineering, engineering organization, and cognitive engineering, as well as theoretical foundations.

- A profound pattern for preventing progress in software engineering is that the tendency to think that a new idea or future tool will solve all problems. This tendency is so strong that previously solved problems are forgotten as soon as a new idea gains some support, and consequently problems are re-solved in the new style.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

A Unified Framework of Software Engineering Principles

- **The ultimate objective of investigation** into the principles of software engineering is to build an integrated inference framework with axioms implicitly defining primitive concepts, principles, and rules for enabling to construct higher order concepts and theories of software engineering.

- **The unified framework of software engineering principles** models a comprehensive set of fundamental principles as summarized below:

  - PR1. **Abstraction** is to elicit essential properties of a set of objects while omitting inessential details of them.

  - PR2. **Decomposition/modularization** is to break up the functions of a software system and to allocate them into individual modules or components.

  - PR3. **Information hiding** is to reduce and mask unnecessary information of software at a given level from the lower level details.

  - PR4. **Engineering approach** is to adopt proven generic engineering methodology and practice in software development and its organization.

- PR5. **Professionalism** is to set forth the competence or skill required for a professional software engineer who is formally trained and certified.

- PR6. **Tools and environments** are facilities that enable efficient organization of coordinative work or extend human physical and intelligent capability in software development.

- PR7. **Documentation** is a written record that is used to embody system design and architectures, record work products, maintain traceability of serial decisions, log problems and maintenance solutions, and enable postmortem analysis.

- PR8. **Stepwise refinement** is to deductively extend a conceptual model of requirements for a given software system by a series of expatiated and incremental specifications at increased degrees of details.

- PR9. **Prototyping** is to evaluate or validate a design and feasibility of a required system based on the implementation of a prototype of the system.

- PR10. **Adopting engineering notations** is to abstract, denote, and model user requirements and system specifications expressively and explicitly.

- PR11. **Process modeling** is deal with organizational and managerial issues in software engineering.

- PR12. **Reuse** is to adopt higher-level building blocks, such as algorithms, methods, processes, patterns, frameworks, in order to improve efficiency, productivity, and quality of software engineering.

- PR13. **Measurements and metrics** are to elicit generic software attributes, quantify their measurement, and unify their metrics.

- PR14. **Cognitive complexity control** is to deal with the innate difficulty in both architectural and behavioral design and implementation of software systems by a variety of means such as abstraction, modularization, descriptive notations, stepwise refinement, and prototyping.

- PR15. **Formal requirement specification** is to formally and rigorously specify customer's nonprofessional requirements for a software system, in order to avoid any misinterpretation and ambiguity, and to eliminate any conceptual gap and inconsistency.

- PR16. **Systematic quality assurance** is to adopt a systematic tackle of the multifaceted attributes of software quality and their quantitative measurement.

- PR17. **Review and inspection** is to find and eliminate software design and implementation defects via reading and examining the work products by peer or more experienced reviewers.

- PR18. **Management engineering** is to acknowledge the crucial need of a suitable theory for organizing and coordinating large human groups in large-scale projects in software engineering.

- PR19. **Acquiring domain knowledge** is to obtain four aspects of application knowledge: (a) the nature of the problem, (b) the environment and context of the problem, (c) current customer practice for dealing the problem, and (d) existing regulations and constraints in the application area.

- PR20. **Customer involvement** is to incorporate all stakeholders, particularly the end users of a software system, into the entire lifecycle of the system by customer reviews and joint meetings.

- PR21. **Feasibility analysis** is to rigorously estimate and evaluate both the technical and economical feasibilities of a given software project before the later-phase processes may be continued.

- PR22. **Improving comprehensibility** is to explicitly and expressively describe the intangible problem and its solution in software engineering with improved understandability, readability, and cognitive capability.

- PR23. **Exception handling** is to consider not only customer required functions for a given system, but also all possible exceptions that may drive the system into illegal state(s) in the entire state space in system design and specification.

- PR24. **Divide and conquer** is to partition a complex system into multiple components, and then to deal with these individual components in order to reduce complicity.

- PR25. **Explicit embodiment** is to deal with the implicitness and inexpressiveness in software engineering by introducing more powerful descriptive means at a higher level of abstraction and precision.

- PR26. **Establishing theoretical foundations** is to elicit rigorous theories and generic laws of software engineering on the basis of empirical observations and practices.

- PR27. **Architecture and behavior modeling** is a software engineering principle that states software system models are a hybrid model where both architectures and behaviors should be coherently described.

- PR28. **Standardization** is to integrate, regulate, and optimize existing principles and best practices in research and in the industry.

- PR29. **Systems engineering** is to adopt system science theories and approaches to deal with complicated architectures and behaviors of software.

- PR30. **Engineering organization** is to adopt the coordinative work organization theory in order to optimize team, project, and enterprise organizations.

- PR31. **Cognitive engineering** is to address the cognitive complexity and human intelligent manageability as the dominant problem in almost all processes of software design, implementation, and maintenance.

Software Engineering Principles as Measures to its Constraints

- The 31 basic principles of software engineering are used as the **fundamental measures** for dealing with the 14 basic constraints of software engineering.

- The set of 14 basic software engineering constraints can be classified into three categories known as the **cognitive, organizational, *and* resources constraints**.

Questions and Research Opportunities

2.1 A software engineering principle is a generic theorem, rule, or law of a theory that can be applied to a wide range of cases or instances in software engineering methodologies and practice. Search on the Internet and try to identify one or more principles for software engineering, which are not included in this chapter.

2.2 Analyze the similarity and differences between principle DP1 – information hiding proposed by D.L. Parnas and TH5 – abstraction by C.A.R. Hoare.

2.3 Compare the principles of "invisibility" proposed by F. Brooks (FB4) that suggests *visualization* and "intangibility" in the unified set of principles that suggests *abstraction*. Discuss the relationship and differences of these two principles.

2.4 Discuss and comment on S. McConnell's observation: "An investment in learning software engineering principles is a particular good investment for a software professional to make because that knowledge will last a whole career – not be half obsolete within three years (as those of software development technologies) [McConnell, 1999]."

2.5 The IEEE Software Engineering Standards Committee (SESC) proposed a set of criteria on selecting the fundamental principles for software engineering [SESC, 1996/97/99] as given in Section 2.3.1. Compare it and Definition 2.2; discuss what the differences between empirical and theoretical principles are for software engineering.

2.6 A software engineering principle serves as a fundamental proposition for logical reasoning and deduction. Principles can be classified into two categories known as the *formal* and *empirical* (*heuristic)* principles as described in Theorem 1.1.

The 31 fundamental software engineering principles elicited in this chapter may be classified into the above two categories known as the *formal* and *empirical (heuristic)* principles. Use a table to classify these 31 principles into the formal or empirical category, and explain your rationale. (Note: Some of the principles may belong to both categories.)

2.7 A software engineering principle serves as a fundamental proposition for logical reasoning and deduction. Why are most known principles in software engineering empirical and heuristic so far?

2.8 According to Theorem 2.2, for supporting rigorous reasoning and decision making in software engineering, formalization of those empirical principles seems profoundly important and necessary. Try to formalize any of the 31 principles using rigorous mathematical means.

2.9 Briefly describe the relationships between the 31 fundamental principles and the 14 basic constraints of software engineering.

2.10 Reviewing the 31 software engineering principle, did you observe any conflict or contradictory principles that need further study?

2.11 How to formalize the empirical principles of software engineering in order to support rigorous reasoning and decision making in software engineering? Would formalization with quantified models result in a set of laws for software engineering?

2.12 Read the following classic articles:

Parnas, D.L. and Clements, P.C. (1986), A Rational Design Process: How and Why to Fake It, *IEEE Trans. on Software Engineering*, 12(2), pp. 251-257.

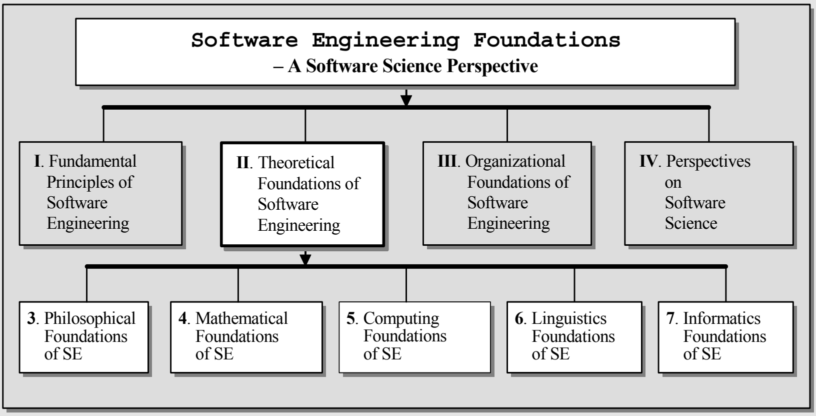
Parnas, D.L. (1994), Software Aging, *Proc. 16th International Conference on Software Engineering*, Sorento, Italy, May, pp.279-287.

Discuss the following topics in a group:

- About the author.
- Why the author put emphases on design without documentation is not design?
- Why, in almost all software engineering processes and/or work products, "if it is not documented, it is not done"?
- What conclusions derived in the articles interested you?
- Express your arguments or counter-points on any of the conclusions.

PART II

THEORETICAL FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations – A Software Science Perspective

I. Fundamental Principles of Software Engineering

II. Theoretical Foundations of Software Engineering

III. Organizational Foundations of Software Engineering

IV. Perspectives on Software Science

3. Philosophical Foundations of SE

4. Mathematical Foundations of SE

5. Computing Foundations of SE

6. Linguistics Foundations of SE

7. Informatics Foundations of SE

Theoretical software engineering studies the nature of software, mathematical models of software architectures, mechanisms of software behaviors, methodologies of large-scale software development, and the laws behind software behaviors and software engineering practices. Part II attempts to present the philosophical, mathematical, computing, linguistic, and informatics metaphors of software and software engineering.

It is recognized that all the fundamental problems in software engineering are complicated theoretical problems rather than only empirical ones. A rigorous and formal approach is needed to seek the fundamental principles and laws of software engineering, and their transdisciplinary foundations required by the nature of the problems in software engineering.

The knowledge structure of Part II on *Theoretical Foundations of Software Engineering* is as follows:

- Chapter 3. Philosophical Foundations of Software Engineering
- Chapter 4. Mathematical Foundations of Software Engineering
- Chapter 5. Computing Foundations of Software Engineering
- Chapter 6. Linguistics Foundations of Software Engineering
- Chapter 7. Information Science Foundations of Software Engineering

This part addresses the theoretical foundations of software engineering with emphases on fundamental theories of software engineering via the cross-fertilization among engineering philosophy, denotational mathematics, computing theories, formal linguistics, and informatics. It is noteworthy that, historically, language-centered programming had been the dominant methodology in computing and software engineering. However, this should not be taken for granted as the only approach to software engineering, because the expressive power of programming languages is inadequate to deal with complicated software systems, and the rigorousness and level of abstraction of programming languages are too low in modeling the architectures and behaviors of software systems. This is why a bridge in mechanical engineering or a building in civil engineering was not modeled or described by natural or artificial languages. This observation leads to the recognition of the need for mathematical modeling of both software system architectures and static/dynamic behaviors, supplemented with the support of automatic code generation systems.

Chapter 3, *Philosophical Foundations of Software Engineering*, explains the relationship of objects and entities between the abstract world and the physical world. The problem domain of software engineering can be seen in the connection between these two worlds, where philosophy provides the basic judgment for evaluating and predicating complicated phenomena in software engineering. Philosophies of sciences and engineering in general, and philosophical methodologies for software engineering in particular, are explored in this chapter. The properties of software and the philosophy of software engineering are presented, complemented by a set of practitioners' philosophies in engineering known as Murphy's laws. Formal inference methodologies such as deductive, inductive, abductive, and analogical inferences are described, which forms the logical means of software engineering.

Chapter 4, *Mathematical Foundations of Software Engineering*, investigates the *logical* and *algebraic* properties and laws of software and software engineering. Mathematics enabling rigorous inferences to be carried out on the basis of simple deductive rules, and the formally documented results are validated without exceptions. Therefore, the entire theory of software engineering is about mathematical modeling of software and denotational mathematics for software engineering. Essential elements of denotational mathematics for modeling software architectures and software system behaviors are analyzed. New mathematical structures such as cumulative relations and Real-Time Process Algebra (RTPA) are developed on the basis of conventional fundamental mathematics such as set theory, Boolean algebra, and mathematical logic. RTPA serves as both a denotational mathematical means and a system design and refinement methodology for software engineering.

Chapter 5, *Computing Foundations of Software Engineering*, analyzes the *computational* and *denotational* properties and laws of software and software engineering. This chapter examines what computer science may provide for software engineering as well as what it may not. A new treatment of computing theories for software engineering is taken, which focuses on the needs for modeling and manipulating complicated data objects, behaviors, programs, and resources in software engineering. Data objects modeling methodologies are presented with the focuses on type theory and architectural modeling of software systems. Behavioral modeling, particularly a set of Basic Control Structures (BCS's), is formally described. Programs are then modeled as the coordination and interaction between computational behaviors and data objects. As a result, the abstract model of a generic computing system is formally described encompassing all computing resources and processes.

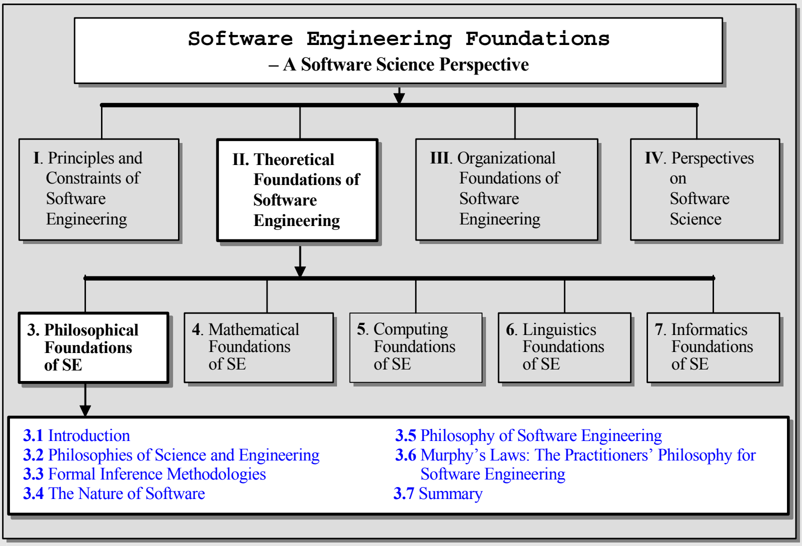
Chapter 6, *Linguistics Foundations of Software Engineering*, presents the *syntactical and semantic* properties and laws of software and software engineering. Linguistics and formal language theories play important roles in computing theories; without them computing and software engineering theories would not be complete. This chapter analyzes not only how linguistics may improve the understanding of programming languages and their work products – software, but also how formal language theories extend the study of natural languages. Formal language theories for rigorous treatment of language elements are described on syntaxes, semantics, grammars, and linguistic analyses from the bottom up. A formal semantics theory known as deductive semantics is presented, which is used to formally describe the semantics of RTPA. Comparative analyses of natural and programming languages, as well as linguistics perceptions on software engineering, are presented.

Chapter 7, *Information Science Foundations of Software Engineering*, analyzes the *informatics* properties and laws of software and software engineering. Information is the product of either natural or machine intelligence. Informatics, the science of information, studies the nature of information, its processing, and ways of transformation between information, matter, and energy. A fundamental discovery in computer science and software engineering is that software, as a unique entity, is not constrained by any law and principle known in the physical world. This chapter demonstrates that software obeys the laws of informatics. The evolvement of information science from classic, contemporary, to cognitive informatics is reviewed. The classic information theories and its perception on information as probability-based properties of signals and channels are introduced. Then, contemporary informatics and modern perceptions on information as abstract entities in computing and software engineering are discussed. A set of informatics laws that constrains the behaviors of software is described, and their applications in software engineering are presented.

Part II will establish a coherent theoretical framework of software engineering with a comprehensive set of formal principles and laws. New structures of denotational mathematical means will be developed to deal with the innate complexity of software systems. The philosophical, informatics, and linguistic theories and laws that constrain software and software engineering practice will be systematically derived. With this part as a basis, the empirical framework of software engineering, in terms of its organizational, system engineering, and cognitive informatics foundations, will be presented in Part III.

Chapter 3

PHILOSOPHICAL FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
3. Philosophical Foundations of SE
4. Mathematical Foundations of SE
5. Computing Foundations of SE
6. Linguistics Foundations of SE
7. Informatics Foundations of SE

3. Philosophical Foundations of SE

Knowledge Structure

❍ Philosophy of sciences and engineering

- The natural world and the abstract world
- The basic axioms about nature
- Epistemology and foundationalism
- Holism vs. reductionism
- Positivism vs. rationalism
- Empiricism and objectivity
- Determinism vs. indeterminism
- Natural intelligence vs. artificial intelligence
- Ethical philosophy of engineering

❍ Formal inference methodologies

- Logical argumentations
- Deductive inferences
- Abductive inferences
- Inductive inferences
- Analogical inferences

❍ The nature of software

- The three situations where software is needed
- The behavioral space of software
- Properties of software

❍ Philosophy of software engineering

- The cognitive characteristics of SE
- The nature of SE

❍ Murphy's laws: the practitioners' philosophy for software engineering

- Murphy's laws on generic engineering
- Murphy's laws on SE

Learning Objectives

- To understand the structure of philosophy for science and engineering.
- To know fundamental philosophical thoughts and views for science and engineering.
- To know basic logical argument methodologies.
- To be familiar with the formal inference methodologies in reasoning.
- To understand the nature of software and its 3-D behavioral space.
- To understand the nature of software engineering and its cognitive characteristics.
- To be able to apply the philosophies of science and engineering to software engineering.

"Philosophy is a subject devoted to evaluating arguments and constructing theories."

Elliott Sober (1995)

"Everything it is possible for us to analyze depends on a clear method which distinguishes the similar from the not similar."

Linneus G. Plantarum (1754)

3.1 Introduction

Having provided an improved understanding of the intensions and extensions of software engineering in Chapters 1 and 2, this chapter attempts to investigate the foundations of software engineering from the perspectives of philosophy, particularly, the means and methodologies of rigorous logical inference and reasoning.

Software is a brainchild of human creativity, and it is created to do something repeatable at high speed, to extend human capability, reachability, and/or memory capacity. Therefore, software systems, to some extent, can be perceived as a virtual agent of human beings.

W.I. Beveridge (1957) questioned that "Elaborate apparatus plays an important part in the science of today, but I sometimes wonder if we are not inclined to forget that the most important instrument in research must always be the mind of man."

Modern sciences have been mainly using analytic methodologies and mathematics in theory development and problem solving. However, the analytic approach has its inherent limitation for possibly losing the forest to the trees in reasoning.

It is a common phenomenon that almost all preeminent scientists are philosophers too. They adopt philosophy, the tool of abstraction, synthesis, induction, and deduction, to develop new theories when there are inadequate laws or lack of intuitive facts to be based for reasoning and draw rational conclusions. For examples, Isaac Newton's *Philosophiae Naturalis Principia Mathematica* (The *Principia* 1687), Max Planck's *The Philosophy of Physics* (1936), and C.A.R. Hoare's *The Philosophy of Engineering* [Hoare and Jones, 1989].

The philosophical foundations of software engineering highlight the relationship between the abstract world modeled by information and the physical world modeled by matter and energy. The problem domain of software engineering can be seen in the connection between the abstract world to the physical world. That is, software engineering deals with abstractions, while manufacturing engineering deals with realization. Due to this ability to call upon multiple levels of abstraction, the problem domain of software engineering is infinite but it also requires that the process is design intensive.

This chapter explores the philosophical foundations and logical means of software engineering. In the remainder of this chapter, Section 3.2 surveys philosophies of sciences and engineering. Section 3.3 develops a set of formal inference methodologies. Section 3.4 examines the nature of software and its properties. Section 3.5 presents the philosophy of software engineering, complemented by Murphy's laws – the practitioners' philosophy in Section 3.6.

3.2 Philosophy of Sciences and Engineering

Philosophy addresses fundamental questions of great generality and ways of reasoning. Philosophy studies the common doctrines, known or unknown, shared by all science disciplines. As described in Section 1.2.4, philosophy is the highest level of abstract knowledge that is general, fundamental, and universally true. Human wonder about the nature and themselves started by philosophical queries and concluded in philosophical doctrines [Aristotle, 1925; Plato, 1961/75; Descartes, 1979; Russell, 1948]. Therefore, philosophy is the common root of all sciences and the crystallization of general knowledge of mankind in the pursuit of understanding the natural rules and utilizing the natural resources.

Philosophy can be divided into the following four branches:

- *Epistemology*: The study of knowledge itself.
- *Metaphysics*: The study of fundamental concepts of the nature such as existence, appearance, reality, and determinism.
- *Logic*: The study of rules of reason.
- *Ethics*: The study of right and wrong, good and evil, obligations and rights, justice, and social organization.

Scientific and engineering philosophy is an aspect of philosophy that studies general phenomena and rules of sciences and engineering methodologies. The following subsections present eight pairs of philosophical thought on science and engineering, as well as the ethical philosophy of engineering.

3.2.1 THE NATURAL WORLD AND THE ABSTRACT WORLD

According to the IME model introduced in Section 1.1.1, *matter*, *energy*, and *information* are the three essences of natural and the abstract worlds as shown in Fig. 1.2 [Wang, 2003a/07a]. In a modern society, information plays more and more important roles because it is the only link between the physical (external) and the abstract (internal) worlds in human life. It is also the fundamental means for modeling the abstract world. In cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06], software is perceived as a special type of *instructive* and *behavioral information* that describes a solution for the design and implementation of a computer system.

The IME model also reveals there are two categories of objects under study in science and engineering known as the *concrete entities* in the real world and the *abstract objects* in the information world. In the latter, an important part of the abstract objects are human or system *behaviors*, which are planned or executed actions onto the real-world entities and abstract objects.

The 7th Law of Software Engineering

Theorem 3.1 The *universal constraints* state that both the natural world and the perceived abstract world are constrained by certain known restrictions and laws, or by those yet to be known due to both current limitations of natural resources and/or human cognitive capability.

The principle of universal constraints indicates that any theory, method, or technology has its own limitations and constraints. In a certain extent, science and engineering are searching the maximum extent of general relations between entities, phenomena, and behaviors under a set of constraints.

Theorem 3.1 will be found useful in a number of disciplines as described in the following chapters throughout this book, such as the principle of information scarcity in information theory (Property 7.19), the law of conservation of basic engineering constraints (Theorem 8.2), the principle of bounded rationality in decision theories (Lemma 11.2), and the principle of resource scarcity in economics (Lemma 12.1).

3.2.2 THE BASIC AXIOMS ABOUT NATURE

Skinner (1948) stated that science is “a search for order, for uniformities, for lawful relations among the events in nature.” The basic assumptions underlying scientific inquiry in the context of the above world view is that the nature of the universe obeys a set of fundamental axioms, referring to uniformity, determinism, reality, rationality, regularity, replication, and discoverability of natural events and their relations [Christensen, 1997].

Definition 3.1 *Uniformity* is the most basic scientific axiom, which assumes that the future will resemble the past.

Uniformity suggests that natural events and phenomena share generic and common laws, which are observable, repeatable, and determinable. The uniformity principle is the foundation of all inductive inferences.

The effort to uncover the uniform laws of the nature may be carried out by identifying the variables that are linked together, and by constructing experiments that attempt to understand the effects produced by given events. Once it is determined that an event produces an expected effect or a set of them, the uniformity of nature has been uncovered.

Definition 3.2 *Determinism* is a basic scientific axiom that assumes there are causes or determinants for any natural event, phenomenon, and effect, and these causes are observable in the sequence of events or reasonable by studying relations of the events.

Definition 3.3 *Reality* is a basic scientific axiom that assumes the natural phenomena observed as a result of our means of sensation are real and objective rather than subjective perceptions.

Definition 3.4 *Rationality* is a basic scientific axiom that assumes there is a rational basis for any event that occurs in nature and they can be understood through the use of logical reasoning.

Definition 3.5 *Regularity* is a basic scientific axiom that assumes events in nature follow the same laws and occur the same way at all times and places under the same context.

Regularity is a natural extension of the uniformity axiom. If there is uniformity in nature, there must be regularity that can be observed and determined.

Definition 3.6 *Replication* is a basic scientific axiom that requires the results of a study must be reproducible under the same condition.

Reproduction is the criterion to achieve objectivity and rationality. Only through replication can we have any confidence that the results of our studies are valid and reliable. Reproducibility is equally important in both sciences and engineering.

Definition 3.7 *Discoverability* is a basic scientific axiom that assumes the existence of entities and occurrence of events in nature can be observed and determined.

Greek philosopher Thales (625-546 BC) made the first extraordinary rationalistic assumption that the world – the cosmos – was a thing whose mechanisms can be understood by human mind [Doren, 1992]. This means that it is possible to discover the uniformity of nature no matter how difficult it would be.

3.2.3 EPISTEMOLOGY AND FOUNDATIONALISM

Definition 3.8 *Epistemology* is a branch of philosophy that studies concepts of knowledge and their rational justification.

According to epistemology [Sober, 1995], there are six approaches to acquire knowledge or gain cognition of nature. These approaches of cognition are *tenacity*, *intuition*, *experience*, *authority*, *reasoning or inferring*, and *logical inquiry*. *Logical inquiry* is a cognitive methodology for knowledge acquisition by scientific investigation that adopts the processes of problem identification, hypothesis proposing, experiment and testing, and theory forming.

To have the knack of reducing a problem to its simplest and basic elements and then finding a solution by the most direct means are commonly recognized as a vital scientific research method, because this approach is rooted in the philosophy of foundationalism.

Definition 3.9 *Foundationalism* is a basic philosophical view that all propositions known to be true can be divided into *foundational* and *superstructural* ones. The former are indubitable and axiomatically to be true. The latter are propositions that bear deductive or implicated relationship to the foundations.

Rene Descartes (1596 – 1650) is regarded as the father of modern philosophy and the epistemology known as *foundationalism*. In *Core Questions in Philosophy* [Sober, 1995], Descartes' foundationalism is described as follows:

> "The word foundationalism should make you think of a building. What keeps a building from falling over? The answer has two parts. First, there is a solid foundation. Second, the rest of the building, which I will call the superstructure, is attached securely to that solid foundation. Descartes wanted to show that (many if not all of) the beliefs we have about the world are cases of genuine knowledge. To show this, he wanted to derive our beliefs into two categories. There are the foundational beliefs, which are perfectly solid. Second, there are the superstructural beliefs, which count as knowledge because they rest securely on that solid foundation.
>
> "A foundationalist theory of knowledge could also be called a Euclidean theory of knowledge. To show that a given body beliefs counts as knowledge, we use the following strategy: First, we identify the beliefs that will provide the foundations of knowledge (the axioms). These must be shown to have some special property, like being absolutely certain. ... Second, we show that the rest of our beliefs count as knowledge because they bear some special relationship to the foundational items. In Euclid's geometry, the special relationship was deductive implication.
>
> "Descartes was interested in the totality of what we believe. But whether the problem is to describe the foundations of geometry or the foundations of knowledge as a whole, there are two ideas that must be clarified. We need to identify what the foundational items are. And we need to describe the relationship that must be obtained between foundational and superstructural items that qualifies the latter as knowledge."

Foundationalism provides a fundamental approach towards justification of knowledge and belief in epistemology and cognition. For example, Turing's thesis on basic computability, Euclid's geometry, and many mathematical branches are developed on the basis of foundationalism.

The design of this book, *Software Engineering Foundations: A Software Science Perspective*, follows Descartes' foundationalism in order to identify the foundational theories and methodologies of software engineering, and to explain how superstructures, the rational software engineering practices, may be built and derived on these foundations.

3.2.4 HOLISM VS. REDUCTIONISM

Philosophies as the fundamental and general methodologies of sciences and engineering were evolving over all the time in human history. However, the core methodologies that remain stable are holism and reductionism.

Definition 3.10 *Holism* is a philosophical view that perceives a phenomenon and system with wholeness in an integrated, synthetic, and systematic approach.

The word holism is originated from the Greek word *holos* meaning the whole. Holism can be traced back to the time of Aristotle during which philosophers believed that *the whole is more than the sum of its parts* [Klir, 2001]. According to holism, complex organisms and systems as a whole possess special properties when its elements and their interactions reach or go above a certain critical mass, which cannot be found from any of the individual elements.

Definition 3.11 *Reductionism* is a philosophical view that investigates a phenomenon and system by using a decomposition and analytic approach.

Reductionism perceives that any system can be analyzed by breaking it down into more fundamental elements, then the system can be reduced by the properties of its elements. As a holistic psychologist, Max Wertheimer described [Ellis, 1938; Ellis and Fred, 1962]:

> "Science means breaking up complexes into their component elements. Isolate the elements, discover their laws, then reassemble them, and the problem is solved. All wholes are reduced to pieces and piecewise relations between pieces."

The evolution of philosophies of science and engineering can be illustrated in Fig. 3.1 from a historical point of view. As shown in Fig. 3.1, holism had been the main philosophical thought starting from Aristotle during a very long period between 400BC to the 1600s. Then, reductionism has been the dominating philosophy since Rene Descartes. Beginning in the later 1990s, there has been a trend to explore the unification of the two philosophical doctrines when most of the modern scientific and engineering problems become increasingly complicated and the means to solve them become increasingly interrelated with multiple disciplines and systems. This trend of shifting in scientific philosophy is in accordance with the development of systems science and engineering as described in Chapter 10.

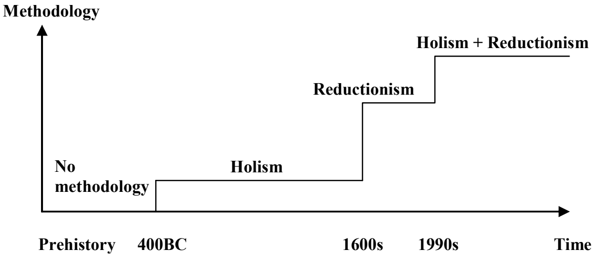


Figure 3.1 Philosophies of sciences and their transitions

3.2.5 POSITIVISM VS. RATIONALISM

The contemporary philosophy behind natural sciences is *positivism* and *rationalism*.

Definition 3.12 *Positivism* is a philosophical view, which states that a thesis about *physical phenomena* must either be analytic or empirical.

Positivism perceives that nature obeys physical laws in the concrete world. According to positivism, an event or fact must be publicly observable and independently repeatable. Validation methodologies in positivism are experiments and logical reasoning.

Natural scientists adopt a common perception that physical phenomena must be re-observable and repeatable. However, most mental phenomena in psychology and cognitive informatics are clearly not, even though all individuals believe they are truly happening based on *rationalism* and *empiricism*.

Definition 3.13 *Rationalism* is a philosophical view to arrive at knowledge in which reasoning is used to acquire, process, derive, and evaluate the knowledge.

Rationalism believes that knowledge or truth can be derived from reason, and the derived knowledge is just as valid as, and even superior to, that gained from observations. Rationalism and reasoning are a vital approach in the scientific process. Reasoning process is used not only to derive hypotheses but also to identify the manner in which these hypotheses are to be tested.

The 3rd Principle of Software Engineering

Theorem 3.2 The *validation of abstract propositions* states that the abstract and information-based propositions and work products, such as a design or a specification of a system, are bounded by logical verifications, mathematical proofs, systematical reviews, behavioral simulations and tests, and/or in field trials.

It is noteworthy that reasoning based on the same input may result in controversially derived information or conclusions, in which one or all of them could be wrong. Therefore, cross evaluation of any derived information by other methodologies, such as positivism and empiricism, is often necessary.

This contradiction can be explained by the IME model as described in Section 1.1.1, that classifies the natural phenomena into two categories known as those of the *natural/concrete world* and of the *abstract/perceptual world*. According to the IME model, the mental phenomena and cognitive processes, particularly perceptivity and thinking, should be recognized as a new category of special phenomena occurring in the abstract and information world that apparently do not obey specific rules observed in the physical world. In other words, all information/mental-process-oriented sciences deal with a totally different category of phenomena that are constricted by informatics and cognitive laws rather than the physical ones. Software and software engineering methodologies fall exactly into this category.

3.2.6 EMPIRICISM AND OBJECTIVITY

Definition 3.14 *Empiricism* is a philosophical view that states knowledge can be gained through the experience of an event, the observation of a fact, or the use of a methodology.

Empiricism perceives that mental phenomena in psychology and cognitive informatics may not be publicly observable and independently repeatable, and they don't obey all rules observed in the physical world. However, they obey informatics and cognitive laws in the abstract world. According to empiricism, an event or fact can be validated by experience, logical reasoning, and mathematical proving.

It is noteworthy that empiricism may result in a subjective observation. Therefore, the criteria of objectivity, replication, and causation are developed in order to maintain rigor and accuracy in scientific inquiry.

Definition 3.15 *Objectivity* is a scientific criterion that requires an observation must be independent of individual opinion, bias, or prejudice.

True or false of an objective matter is independent of what anyone believes or thinks.

Definition 3.16 *Causality* is a cause-and-effect relationship where the manipulation of one event produces another event as the effect of the causal event.

In his work, *A System of Logic*, John S. Mill (1843) set forth canons for identifying causality experimentally:

- *Method of agreement*: The identification of the common element in several instances of an event.
- *Method of difference*: The identification of the different effects produced by variation in only one event.
- *Joint method of agreement and difference*: The combination of the first and the second canons to identify causation.
- *Method of concomitant variation*: The identification of parallel changes in two variables by a correlation between them.

The four canons of Mill enable one to adequately grasp the idea of causation and to identify the relationships between a set of variables. However, they do not allow one to name the single factor that causes an effect. This identifies the need for distinguishing the necessary and the sufficient conditions for the occurrence of an event.

The 8th Law of Software Engineering

Theorem 3.3 The *law of causality* states that a condition must be both necessary and sufficient to qualify as a cause, where the *necessary* condition is a condition that must be present in order for the effect to occur, while the *sufficient* condition is a condition that will always produce the effect.

Therefore, the finding of a cause for an event means that both the necessary and the sufficient conditions have been met in causality analyses, as W.I. Beveradge wrote:

> "The current attitude is that scientific theories aim at describing association between events without attempting to explain the relationship as being causal. The idea of cause, as implying an inherent necessity, raises philosophical difficulties and in theoretical physics the ides can be abandoned with advantage as there is then no longer the need to postulate a connection between the cause and effect. Thus, in this way, science confines itself to description – 'how', not 'why' [Beveradge, 1957]."

3.2.7 DETERMINISM VS. INDETERMINISM

Definition 3.17 *Determinism* is a philosophical view, which states the thesis that a complete description of the causal facts at one time uniquely determines what must happen next.

Determinism states the causes of a natural event, phenomenon, and effect are observable in the sequence of events or reasonable by studying relations of the events. There is only one possible future given a complete description of the present. For example, the Newtonian physics says that the behavior of physical objects is deterministic. In software engineering, most automata and process dispatching algorithms are deterministic based on a current state and given event(s).

Definition 3.18 *Indeterminism* is a philosophical view, which states the thesis that even a complete description of the present does not uniquely determine what will happen in the future.

Indeterminism describes the phenomena in the natural world that not all events are wholly determinable by antecedent causes. According to indeterminism, there is more than one possible future and each with its own probability of coming true under a given complete description of the present. For examples, the quantum theory says that nature is indeterministic. In computing, there are indeterministic automata and process dispatching algorithms, whose behavior or next state is unpredictable caused by internal memory and intricate internal interacting mechanisms.

3.2.8 NATURAL INTELLIGENCE VS. ARTIFICIAL INTELLIGENCE

It is found that the natural intelligence and artificial intelligence share the same cognitive informatics foundations, because the latter is a machine implementation of the former. Conventional machines are invented to extend human physical capability, while modern information processing machines, such as computers, communication networks, and robots, are developed for extending human intelligence, memory, and the capacity for information processing [Turing, 1950; Wang, 2004a/2006b/2007a/07b/07f]. Therefore, any machine that may implement a part of human behaviors and actions in information processing is significantly important.

It is recognized [Wang, 2004a/2007a/07f] that the basic approaches to implement intelligent behaviors can be classified as shown in Table 3.1.

Table 3.1
Approaches to Implement Intelligence

No.	Means	Approach
1	Biological organisms	Naturally grown
2	Silicon automata	Wired
3	Computing systems	Programmed
4	Other (future) means	Hybrid

Observing Table 3.1, software for computation is the third approach to simulate and implement the natural intelligence by programmed logic. This indicates that the nature of software is the simulation and execution of human behaviors, and the extension of human capability, reachability, persistency, memory, and information processing speed. Therefore, the natural and machine (artificial) intelligence share the same cognitive foundation or there is no difference between them in principles and mechanisms rather than implementation means.

The 4th Principle of Software Engineering

Theorem 3.4 The *compatible intelligent capability* states that *natural intelligence* (NI) and *artificial intelligence* (AI) are compatible by sharing the same mechanisms of intelligent capability, i.e.:

$$\text{AI} \propto \text{NI} \qquad (3.1)$$

On the basis of Theorem 3.4, the following theorem can be derived.

The 9th Law of Software Engineering

Theorem 3.5 The *inclusive intelligent capability* states that *artificial intelligence* (AI) is a subset of *natural intelligence* (NI), i.e.:

$$AI \subseteq NI \qquad (3.2)$$

Theorem 3.5 indicates that AI is dominated by NI. Therefore, one should not expect a computer or a software system to solve a problem where human cannot. In other words, no AI or computer systems may be designed and/or implemented for a given problem where there is no solution being known collectively by human beings. Further, Theorem 3.5 explains that without understanding the mechanisms and laws of NI, the development or implementation of AI is not scientifically based yet.

3.2.9 ETHICAL PHILOSOPHIES OF ENGINEERING

Definition 3.19 *Ethics* is a branch of philosophy that develops moral criteria to guide human behavior and professional practice.

A number of ethical theories have evolved since the dawn of civilization. Four of those theories, which have stood the test of time and are relevant to applications in engineering, are Aristotle's virtue ethics, Mill's utilitarianism, Kant's formalism or duty ethics, and Locke's rights ethics.

The *virtue-based ethics* of Aristotle (384 – 322BC) describes that happiness is achieved by developing virtues, or qualities of character, through deduction and reason. An act is good if it is in accordance with reason. This usually means a course of action that is the golden mean between extremes of excess and deficiency.

Mill's *utilitarianism* (1806 – 1873) states that an action is morally correct if it produces the greatest benefit for the greatest number of people. The duration, intensity, and equality of distribution of the benefit should be considered.

The *duty-based ethics* developed by Kant (1724 – 1804) believes that each person has a duty to follow those courses of action that would be acceptable as universal principles for everyone to follow.

The *rights-based ethics*, represented by Locke (1632 – 1704), perceives that all persons are free and equal, and each has a right to life, health, liberty, possessions, and the product of one's labor. However, it is occasionally difficult to determine when one person's rights infringe on another person's rights.

Professionalism is a part of the ethical philosophy. The philosophy of professionalism for software engineering will be discussed in Section 8.4.5.

3.3 Formal Inference Methodologies

Inferences are a formalized cognitive process that reasons a possible causal conclusion from given premises based on known causal relations between a pair of cause and effect proven true by *empirical observations, theoretical inferences,* and/or *statistical regulations*. Formal logic inferences may be classified as causal argument, deductive inference, inductive inference, abductive inference, and analogical inference. All formal logical inferences can only be carried out on the basis of abstract properties shared by a given set of objects under study. In other words, abstraction and formalization described in Sections 1.2.4 and 4.2 are the foundation of formal inferences.

3.3.1 LOGICAL ARGUMENTATIONS

Mathematical logics, such as propositional and predicate logic, provide a powerful means for logical reasoning and inference on truth and falsity [Hurley, 1997], which will be systematically described in Chapter 4.

Definition 3.20 An *argument* $\mathcal{A}$ is an assertion that yields ($\vdash$) a proposition Q called the conclusion from a given finite set of propositions known as the premises $P_1, P_2, \ldots, P_n$, i.e.:

$$\mathcal{A}\mathbf{BL} \triangleq (P_1\mathbf{BL} \wedge P_2\mathbf{BL} \wedge \ldots \wedge P_n\mathbf{BL} \vdash Q\mathbf{BL})\mathbf{BL} \tag{3.3}$$

where the argument and all propositions are in type Boolean ($\mathbf{BL}$). Hence, $\mathcal{A}\mathbf{BL} = \mathbf{T}$ called a *valid* argument, otherwise it is a *fallacy*, i.e., $\mathcal{A}\mathbf{BL} = \mathbf{F}$.

Eq. 3.3 can also be denoted in the following inference structure:

$$\mathcal{A}\mathbf{BL} \triangleq \left| \frac{Premises\mathbf{BL}}{Conclusion\mathbf{BL}} \right. = \left| \frac{P_1\mathbf{BL} \wedge P_2\mathbf{BL} \wedge \ldots \wedge P_n\mathbf{BL}}{Q\mathbf{BL}} \right. \tag{3.4}$$

Example 3.1 The following expressions are concrete arguments:

(a) A concrete deductive argument

$$\mathcal{A}_1\mathbf{BL} \triangleq \left| \begin{array}{l} \text{Information processing is an intelligent behavior } (P_1). \\ \underline{\wedge \text{ Computer is able to process information } (P_2).} \\ \qquad \text{Computer is an itelligent machine } (Q). \end{array} \right. \tag{3.5}$$

(b) A concrete inductive argument

$$\mathcal{A}_2\mathbf{BL} \triangleq \left| \begin{array}{c} \text{Human is able to process information } (P_1). \\ \wedge \text{ Computer is able to process information } (P_2). \\ \hline \text{Information processing is a common property of} \\ \text{itelligence } (Q). \end{array} \right. \tag{3.6}$$

Example 3.2 The following expressions are abstract arguments:

(a) Abstract deductive arguments

$$\mathcal{A}_3\mathbf{BL} \triangleq \left| \begin{array}{c} \forall x \in S, P(x) \wedge a \in S \\ \hline \exists x = a, P(a) \end{array} \right. \tag{3.7}$$

$$\mathcal{A}_4\mathbf{BL} \triangleq \left| \begin{array}{c} \forall n \in \mathbf{N},\ n < n + 1 \\ \hline \exists n = 1 \in \mathbf{N},\ 1 < 2 \end{array} \right. \tag{3.8}$$

where **N** represents the type suffix of natural numbers.

(b) Abstract inductive arguments

$$\mathcal{A}_5\mathbf{BL} \triangleq \left| \begin{array}{c} \exists x = a \in S, P(a) \\ \wedge\ \exists x = b \in S, P(b) \\ \wedge\ \exists x = c \in S, P(c) \\ \hline \forall x \in S, P(x) \end{array} \right. \tag{3.9}$$

$$\mathcal{A}_6\mathbf{BL} \triangleq \left| \begin{array}{c} \exists n = 1 \in \mathbf{N} \Rightarrow \sum_{i=1}^{1} i = \dfrac{1 \bullet (1+1)}{2} \\ \wedge\ \exists n = 14 \in \mathbf{N} \Rightarrow \sum_{i=1}^{4} i = \dfrac{14 \bullet\ (14+1)}{2} \\ \wedge\ \exists n = 15 \in \mathbf{N} \Rightarrow \sum_{i=1}^{5} i = \dfrac{15 \bullet (15+1)}{2} \\ \hline \forall n \in \mathbf{N} \Rightarrow \dfrac{n(n+1)}{2} \end{array} \right. \tag{3.10}$$

where **N** represents the type of natural numbers.

In the above examples that the premier propositions should be arranged in a list that the most general ones are put in the front. This condition preserves the deductive chain in reasoning.

It is noteworthy that propositional arguments can be classified as a kind of *causal* and *static* inference. More *rigorous* and *dynamic* inferences may be modeled and described as a set of cognitive processes encompassing a series of simple inference steps as described in the following subsections.

3.3.2 DEDUCTIVE INFERENCES

For seeking generality and universal truth, either the objects or the relations can only be rigorously described and formally inferred by abstract models rather than real world details.

Definition 3.21 *Deduction* is a cognitive process by which a specific conclusion necessarily follows from a set of general premises.

Deduction is a reasoning process that discovers or generates new knowledge based on generic beliefs one already holds such as abstract rules or principles. The validity of a deductive inference depends on its conformity to the validity of generic principle; at the same time, the generic principle that the deduction is based on is evaluated during the deductive practice.

The 5th Principle of Software Engineering

Theorem 3.6 The *generic formula of deductive inference* states that, given an arbitrary nonempty set $\mathbf{X}$, let $p(x)$ be a proposition for $\forall x \in \mathbf{X}$, a specific conclusion on $\exists a \in \mathbf{X}$, $p(a)$ can be drawn as follows:

$$\forall x \in \mathbf{X}, p(x) \vdash \exists a \in \mathbf{X}, p(a) \quad (3.11a)$$

A composite form of Eq. 3.11a can be given below:

$$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash (\exists a \in \mathbf{X}, p(a) \Rightarrow q(a)) \quad (3.11b)$$

In Theorem 3.6, $\vdash$ denotes yield or a causal relation. Any valid logical statement, established mathematical formula, or proven theorem can be used as the generic promise for facilitating the above deductive inferring process.

Example 3.3 Let $P(n)$**BL** $\triangleq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ be a proposition, $n \in \mathbf{N}$, a deductive inference for a given n =10 can be derived as follows:

$$\forall n \in \mathbf{N}, P(n)\mathbf{BL} \triangleq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\vdash \exists n = 10 \in \mathbf{N},$$

$$P(10)\mathbf{BL} = \{\sum_{i=1}^{10} i = \frac{10(10+1)}{2} = 110/2 = 55\}\mathbf{BL}$$

$$= \mathbf{T}$$

It is noteworthy that not every deduction may reach a sound deductive argument as shown in Table 3.2 [Hurley, 1997]. Based on Table 3.2, Corollary 3.1 can be derived.

Table 3.2
Sound Argument by Deductive Validation

		Conclusion	
		T	**F**
Premises	**T**	Valid and sound	Invalid
	F	Invalid	Invalid

Corollary 3.1 A *sound deductive inference* is yielded *iff* all premises are true and the argument is valid.

Corollary 3.1 may be used to avoid any deductive dilemma and falsity in logical reasoning.

3.3.3 INDUCTIVE INFERENCES

Definition 3.22 *Induction* is a cognitive process by which a general conclusion is drawn from a set of specific premises based mainly on experience or experimental evidences.

Induction is a reasoning process that derives a general rule, pattern, or theory from summarizing a series of stimuli or events. In contrary to the deductive inference approach, induction may introduce uncertainty during the extension of limited observations into general rules. The inductive inference process encompasses rule learning, category formation, generalization, and analogy.

The 6th Principle of Software Engineering

Theorem 3.7 The *generic formula of inductive inference* states that, if $\exists a, k, succ(k) \in \mathbb{X}$, $p(a)$ and $p(k) \Rightarrow p(succ(k))$ are three valid predicates, then a generic conclusion on $\forall x \in \mathbb{X}, p(x)$ can be drawn as follows:

$$((\exists a \in \mathbb{X}, p(a)) \wedge (\exists k, succ(k) \in \mathbb{X}, (p(k) \Rightarrow p(succ(k)))) \\ \vdash \forall x \in \mathbb{X}, p(x) \tag{3.12a}$$

where $succ(k)$ denotes the next element of k in $\mathbb{X}$.

A composite form of Eq. 3.12a can be given below:

$$((\exists a \in \mathbb{X}, p(a) \Rightarrow q(a)) \wedge (\exists k, succ(k) \in \mathbb{X}, ((p(k) \Rightarrow q(k)) \Rightarrow \\ (p(succ(k)) \Rightarrow q(succ(k))))) \vdash \forall x \in \mathbb{X}, p(x) \Rightarrow q(x) \tag{3.12b}$$

Theorem 3.7 indicates that for a finite list or an infinite sequence of recurring patterns, three samplings (two determinate and one random) are usually sufficient to determine the behavior of the given list or sequence of patterns. Therefore, logical induction is a tremendously powerful and efficient cognitive and inferring tool in science and engineering, as well as in everyday life.

It is noteworthy that because of the limitation of samples, logical induction may result in faulty proofs or conclusions. Therefore, as a rule of thumb, the inference results of logic inductions need to be evaluated or validated by more random samples.

Example 3.4 An *iteration* of a process P in programming can be defined as a series of $n+1$ repetitions, R_i, $1 \leq i \leq$ n+1, of P by mathematical induction, i.e.:

$$\begin{aligned} &R_0 = \otimes, \\ &R_1 = P \rightarrow R_0, \\ &\ldots \\ &R_{n+1} = P \rightarrow R_n, \ \ n \geq 0 \end{aligned} \tag{3.13}$$

where ⊗ denotes skip, or doing nothing but exit.

A recursive process should be terminable or noncircular, i.e., the depth of recursive d_r must be finite. The following lemma guarantees that $d_r < \infty$ for a given recursive process or function [Lipschutz, 1964].

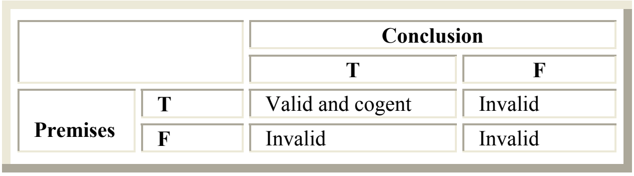
Lemma 3.1 A recursive function is noncircular, i.e., $d_r < \infty$, *iff*:

a) A *base value* exists for certain arguments for which the function does not refer to itself;

b) In each recursion, the argument of the function must be closer to the base value.

It is noteworthy there are certain conditions to reach a cogent inductive argument as shown in Table 3.3 [Hurley, 1997]. Based on Table 3.3, Corollary 3.2 can be derived.

Table 3.3
Cogent Argument by Inductive Validation



		Conclusion	
		T	**F**
Premises	**T**	Valid and cogent	Invalid
	F	Invalid	Invalid

Corollary 3.2 A *cogent inductive inference* is yielded *iff* all premises are true and the argument is valid.

Corollary 3.2 may be used to avoid any inductive dilemma in logical reasoning.

3.3.4 ABDUCTIVE INFERENCES

Definition 3.23 *Abduction* is a cognitive process by which an inference to the best explanation or most likely reason of an observation or event is resulted.

Abduction is widely used in causal reasoning, particularly when a change of events needs to be traced back where not all of the events have been observed.

The 7th Principle of Software Engineering

Theorem 3.8 The *generic formula of abductive inference* states that based on a general implication $\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)$, a specific conclusion on $\exists a \in \mathbf{X}, p(a)$ can be drawn as follows:

$$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash (\exists a \in \mathbf{X}, q(a) \Rightarrow p(a)) \qquad (3.14a)$$

A composite form of Eq. 3.14a can be given below:

$$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x) \land r(x) \Rightarrow q(x)) \\ \vdash (\exists a \in \mathbf{X}, q(a) \Rightarrow (p(a) \lor r(a))) \qquad (3.14b)$$

Abduction is a powerful inference technique for seeking the most likely cause(s) or reason(s) of an observed phenomenon in causal analyses.

3.3.5 ANALOGICAL INFERENCES

Definition 3.24 *Analogy* is a cognitive process by which an inference about the similarity of the same relations holds between different domains or systems, and/or examines that if two things agree in certain respects then they probably agree in others.

Analogy is a mapping process that identifies relation(s) in order to understand one situation in terms of another. Analogy can be used as a mental model for understanding new domains, explaining new phenomena, capturing significant parallels across different situations, describing new concepts, and discovering new relations.

The 8th Principle of Software Engineering

Theorem 3.9 The *generic formula of analogical inference* states that based on a specific predicate $\exists a \in \mathbf{X}, p(a)$, a similar specific conclusion can be drawn *iff* $\exists x \in \mathbf{X}, p(x)$ as follows:

$$\exists x \in \mathbf{X}, p(x) \land \exists a \in \mathbf{X}, p(a) \vdash \exists b \in \mathbf{X} \land b \neq a, p(b) \qquad (3.15a)$$

A composite form of Eq. 3.15a can be given below:

$$(\exists x \in \mathbf{X}, p(x) \land \exists a \in \mathbf{X}, p(a) \Rightarrow q(a)) \\ \vdash (\exists b \in \mathbf{X} \land \mathrm{b} \neq a, p(b) \Rightarrow q(b)) \qquad (3.15b)$$

Analogy is widely used to predict a similar phenomenon or consequence based on a known observation.

Theoretical research is predominantly an inductive process; while applied research is mainly a deductive process. Both inference processes are based on the cognitive process and means of abstraction. The five inference methodologies, causal argument, deduction, induction, abduction, and analogy, form a set of fundamental reasoning processes of the natural intelligence. They are an important set of human cognitive processes as modeled in LRMB [Wang et al., 2006], and have been formally described using RTPA [Wang, 2002a/02b/03c/07a], which will be further described in Chapters 9 and 4, respectively.

3.4 The Nature of Software

The remainder of this chapter will show that philosophy is a powerful means to reveal the nature of software and software engineering. Section 1.2.1 has reviewed the mathematical, product, and informatics metaphors of software. This section presents philosophical thought on the nature of software, and further examines the fundamental properties of software as the objects under study in software engineering.

3.4.1 THE THREE SITUATIONS WHERE SOFTWARE IS NEEDED

The exploration on the nature of software may start from the analysis of usages of software in different contexts. As an explanation, let us consider when one needs a software system in particular, and a computing solution in general. There are three situations, namely the repeatability, flexibility, and run-time determinability, in which a software system is required [Wang, 2004b; Wang et al., 2004b].

Situation 1: The *repeatability* – Software is required when one needs to do something for more than once.

Repeatability is one of the most premier needs for a software solution; whilst it is not the only sufficient condition for requiring a software system because repeatability may also be implemented by wired logic or hardware.

Based on the repeatability, the following situation is introduced.

Situation 2: The *programmability* – Software is required when one needs to repeatedly do something not exactly the same.

In addition to Situations 1 and 2, the following case should be considered in order to complete the necessary and sufficiency usage analyses of software.

Situation 3: The *run-time determinability* – Software is required when one needs to flexibly do something by a series of choices on the basis of varying sequences of events determinable only at run-time.

The third situation may also be considered as the indeterminism at compile-time or design time. This has been identified by Turing (1936), Dijkstra (1975), and Hoare (1978).

The above analysis leads to the following theorem.

The 9th Principle of Software Engineering

Theorem 3.10 The *necessary and sufficient conditions of software usage* state those that warrant the requirements for software solutions are the system behaviors of *repeatability*, *programmability*, and *run-time determinability*.

Theorem 3.10 indicates that the above three situations, namely repeatability, flexibility, and run-time determinability, form the necessary and sufficient conditions that warrant the requirement for a software solution. The third condition is the fundamental issue in computation that determines the complexity of programming.

3.4.2 THE BEHAVIORAL SPACE OF SOFTWARE

It is found that both human and software behaviors can be described by a three-dimensional representative model encompassing *action*, *time*, and *space* [Wang, 2006a]. For software system behaviors, the three dimensions are known as *computational operations*, *event/process timing*, and *memory manipulation* [Wang, 2006a]

Definition 3.25 The *behavior* of a computational statement is a set of observable actions or changes of status of objects operated by the statement.

The 10th Law of Software Engineering

Theorem 3.11 The *behavior space of software* states that the software behavior space Ω is innately three-dimensional, which can be described by a Cartesian product of computational operations *OP*, time *T*, and memory space *S*, i.e.:

$$\Omega = OP \times T \times S \qquad (3.16)$$

The 3-D behavioral space of software systems can be illustrated in Fig. 3.2. According to Theorem 3.11, the fundamental and general requirements for programming languages and software development tools are the capability to express and manipulate the 3-D behaviors. No language or tool only capable for two of the dimensions, usually $OP \times S$, is adequate to cope with the more general requirements in software engineering. Therefore, a lot of problems have stemmed from the implied treatment of time in software development, particularly for real-time software systems, in conventional programming languages and tools.

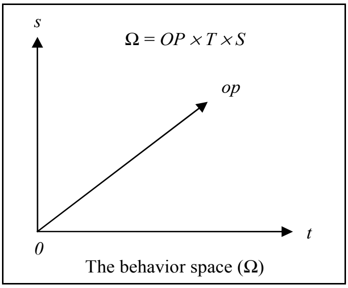


Figure 3.2 The 3-dimensional behavior space of software

3.4.3 PROPERTIES OF SOFTWARE

The creation as software of conventional physical products by the use of programmable and reconfigurable components is a new and quiet industrial revolution. The 19th Century industrial revolutions were oriented on mass production by machinery and standardized process and components [Tayler, 1911; Warner and Low, 1947; Gregory, 1971; Wright, 2002]. The development of soft systems is a revolution that transforms the information processing and intelligent components of the conventional physical products into software.

Therefore, it might be argued that software engineering has become a discipline that is at the root of the knowledge structure of most engineering disciplines. The philosophical considerations explored in this subsection have attempted to clarify a set of fundamental characteristics of software engineering. These considerations also provide a basis for judging the soundness or unsoundness of specific technical solutions for software engineering, while not losing the sight of the woods for the trees.

In addition to the *mathematical, product,* and *informatics properties* of software as discussed in Section 1.2.1, this subsection describes another set of software properties that encompasses the *cognitive, intelligent behavioral,* and *system properties* of software.

3.4.3.1 The Cognitive Properties of Software

The cognitive properties of software refer to its human dependency in almost all processes of software engineering. Eight cognitive properties of software are identified [Wang, 2004b] as shown in Fig. 3.3, such as those of intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures.

One of the unique properties of software is the inherent complexity. Software complexity may be classified into *time, space, symbolic, functional,* and *cognitive* complexities [Wang, 2006c/07a]. The cognitive complexity of software models and measures the cognitive property of software by a product of its architectural and operational complexities. The measure of cognitive complexity of software plays a key role in understanding the fundamental properties of software in all phases of software engineering including the design, implementation, maintenance, and comprehension phases. Detailed description of software cognitive complexity will be provided in Sections 9.6 and 10.7.3.

Another unique property of software is its intangibility in design and comprehension. In Sections 9.2.2 and 9.4.4 it will be revealed that the abstract architecture and behavior of software should be physiologically created in the brain, before they can be represented externally in the forms of system design, code, and documentation.

3.4.3.2 The Intelligent Behavioral Properties of Software

Software is a brainchild of human beings. A software system, to some extent, can be perceived as a virtual agent of humans, because it is created to do something repeatable, to extend human capability, reachability, or memory, just like assistants hired to do the same thing.

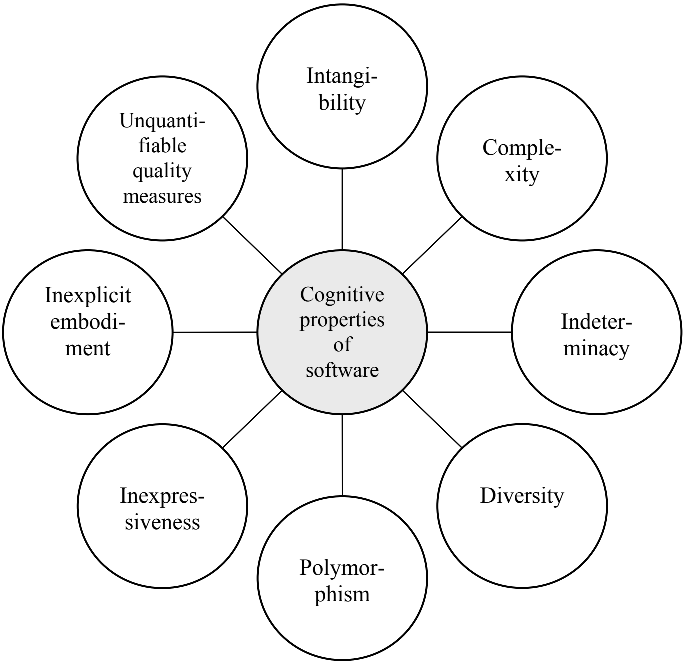


Figure 3.3 The cognitive properties of software

As shown in Table 3.1, software for computation is the third approach to simulate and implement the natural intelligence by programmed logic. According to Theorem 3.4 on inclusive intelligent capability, software is inherently a part of natural intelligence and human behaviors. Because any behavior of machine intelligence is a part of expected human behaviors, according to Theorem 3.4, the following corollary on the relationship between a software behavior and natural intelligence behavior can be derived.

Corollary 3.3 *Software behaviors SB* are a subset of simulated human intelligent behaviors *IB* described by programmed instructive information in a programming language, i.e.:

$$SB \subseteq IB \tag{3.17}$$

Corollary 3.3 reveals that the nature of software is the simulation and execution of human behaviors, as well as the extension of human capability, reachability, persistency, memory, and information processing speed. This leads to the concept of *autonomic computing*, which will be introduced in Section 15.4.1.

3.4.3.3 The System Properties of Software

The nature of software is well fit to the concept of a system, because software is a complex artifact that consists of a large set of different and intricately interconnected components. Changes at one point of a software system may affect the functioning of the entire whole due to propagation of interactions via highly coupled data architectures and intricately interconnected components.

A system is the most complicated object that can be modeled in mathematics. The system science foundations of software engineering define a system as a collection of coherent and interactive entities that has stable functions and clear boundary with the external environment. Systems can be viewed in a hierarchy where one system may be considered part of a larger system. Contemporary system theory [Klir, 1972/2001; Wang, 2005*l*/06d] is a powerful conceptual tool that facilitates a deductive approach to software engineering problem solving. The design of a large software system would divide the system into interrelated subsystems and components. The components may then be developed in parallel by multiple teams before the system is integrated.

System theory seeks to understand the nature of interactions and collaborations of systems, subsystems, and components. A system is considered to be a closure at a certain level of hierarchical architecture of our conceptual world. The system metaphor of software reveals that software in nature is an open system that obeys the abstract system theory and system algebra [Wang, 2006d] as described in Chapter 10 on system science foundations of software engineering.

3.5 Philosophy of Software Engineering

Software engineering is a unique discipline that relies on special philosophical foundations at the top level. Conventional industries produce physical products from raw materials via engineering approaches; the software industry produces software solutions for problems via software engineering. By contrasting the nature of software engineering with other engineering disciplines, it is clear that there are a number of interesting and fundamental differences between them as described in the following parts of this section.

3.5.1 THE COGNITIVE CHARACTERISTICS OF SOFTWARE ENGINEERING

The cognitive properties of software as described in previous section and the cognitive constraints of software engineering as identified in Section 1.3 are helpful to explain the nature of software engineering and its dependency on human cognitive capability. Software as an abstract object under study and its cognitive complexity distinguish software engineering from conventional engineering disciplines. This subsection describes and analyzes the unique characteristics of software engineering stemmed from its basic cognitive constraints.

3.5.1.1 The Abstraction and Intangibility of Software

The abstraction and intangibility of software refer to the nature that a specific software system is a solution for a given problem rather than a physical product. It is abstract and intangible because the solution is represented by code in a programming language. Further, in the time dimension, the software never runs as its static sequence shows in the program and it runs at hundred or thousand times faster than humans can simulate. We may never know the status of some of the intermediate data objects and the maps of dynamic memory allocation of a software system during run-time.

3.5.1.2 The Inherent Complexity and Diversity

The inherent complexity and diversity refer to the difficulty for seeking and implementing solutions in software development and the surprisingly large scope of the problem domain in software engineering. Software development requires high level cognitive capability of abstraction, mental simulation, and relating a static sequential program to its dynamic run-time behaviors. Further, computer science has recognized a whole set of problems that are noncomputable or impossible to implement by programming known as the categories of NP-complete problems [Lewis and Papadimitriou, 1998].

3.5.1.3 The Changeability or Malleability of Software

The changeability or malleability of software refers to the nature that software is considerably vulnerable and unstable. Although a software system may not wear-off, it does decay. Any hardware fault, timing fault, memory allocation problem, and external interference, no matter static or random, may result in unexpected behaviors of a software system. Therefore, focusing on software fault-tolerant and exception handling capability rather than on implementation of ordinary functions that are directly required by customers is a basic sign to distinguish professional or naive software engineers.

3.5.1.4 The Difficulty of Establishing and Stabilizing Requirements

The difficulty of establishing and stabilizing requirements refers to the nature of software engineering that the requirements in the real world are inevitably a moving target. One of the reasons we use a software solution rather than a wired logic for an application is for flexibility and adaptability on a common computer platform. In addition, system development is a learning process for both customers and developers. The involvement of customers in system development stimulates the extension of their requirements for the system. The deeper the customer understands the system, the more the customer seeks for functionality. At the same time, it is interesting that an enthusiastic and idealistic software developer does the same; even this is, sometimes, in contradiction to the financial objective of a project.

3.5.1.5 The Requirement of Varying Problem Domain Knowledge

The requirement of varying problem domain knowledge refers to the nature that software engineering faces unlimited application domains. According to Theorem 3.9, it is recognized that for anything discretely expressive and for any activity needed to be repeated for more than two times, one may consider a software solution. Therefore, the recognition of domain knowledge requirements for an experienced software engineer is a significant issue in software engineering. It will be more important when a program can be automatically generated from requirement specifications in the future.

3.5.1.6 The Indeterminacy and Polysolvability in Design

The indeterminacy and polysolvability in design refer to the nature that software solutions for a given problem are not sufficiently single. There is a combined solution space (options) for system design, functional specification, work products definition, and ways of interaction with operators. Within this extremely large solution space, an application is only one possible implementation that the developer believes is sound. We even can not prove theoretically and economically if the implemented solution is the best or not, because of the size of the solution space. This is the principle of nondeterministic and polysovability in software engineering.

3.5.1.7 The Polyglotics and Polymorphism in Implementation

The polyglotics and polymorphism in implementation refer to the nature that software implementation for a given problem is not sufficiently single in both languages and processes as that of the polysolvability for design described above. There is a combined solution space for programming languages, target machine languages, coding styles, data models, and memory allocations. Any change among these factors may result in a different implementation of a software system.

3.5.1.8 The Dependability of Interactions between Software, Hardware, and Humans

The dependability of interactions between software, hardware, and humans refers to the nature that any software is not running alone. It needs hardware support as an operating platform, and it needs interaction with human beings. Any hardware problem and human error may cause software malfunctions, even if it is believed that software itself does not wear-off.

3.5.2 THE NATURE OF SOFTWARE ENGINEERING

The nature of software engineering may be explained by contrasting its unique characteristics with the conventional engineering disciplines [Wang and King, 2000a]. The uniqueness encompasses virtualization, infinitiveness of problem domains, design-intensive, generic platform, universal logical description, and repeatable processes.

3.5.2.1 Programming: Virtualization vs. Realization

Given manufacturing engineering as an exemplar of conventional engineering, the common approach moves from abstract to concrete, and the final product is the physical realization of an abstract design. However, in software engineering, the approach is reversed. It moves from concrete to abstract. The final software product is the virtualization (coding) and invisible representation of an original design that expresses a real world problem. The only tangible part of a software product is its storage media or its run-time behaviors. As illustrated in Fig. 3.4, this is probably the most unique and interesting feature of software engineering.

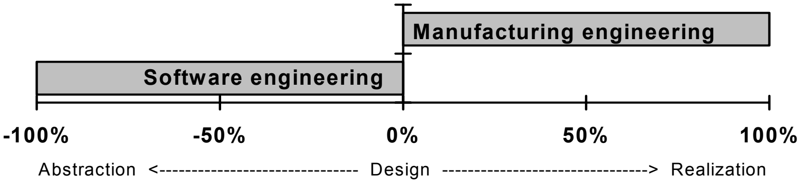


Figure 3.4 Virtualization vs. Realization

3.5.2.2 Problem Domains: Infinitive vs. Limited

The problem domain of software engineering encompasses almost all domains in the real world as shown in Fig. 3.5, from scientific problems and real-time control to word processing and games. It is infinitely large when compared with the specific and limited problem domains of the other engineering disciplines. This stems from the notion of a computer as a universal machine, and is a feature fundamentally dominating the complexity in engineering implementation of large-scale software systems.

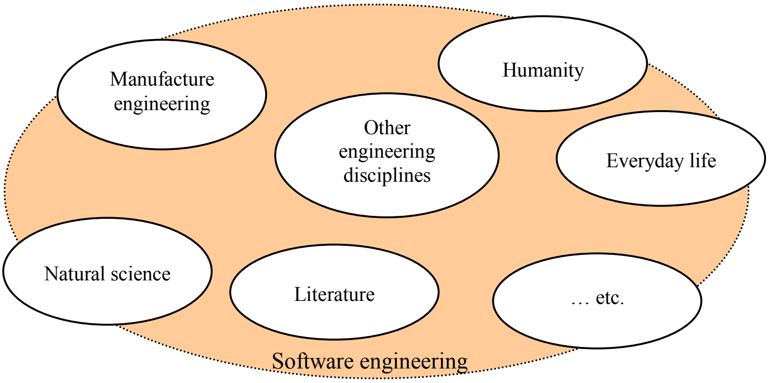


Figure 3.5 Problem Domains: Infinite vs. Limited

3.5.2.3 Effort Distribution: Design Intensive vs. Repetitive Production

As demonstrated in Fig. 3.6, software development is a design-intensive process rather than a mass production process. The design activities include specification, design, implementation, test, and maintenance; the production activities consist of duplication and package. In some extent, software engineering is a design engineering of abstract artifacts, no matter how large of an exaggerated box may be used by the vendor to pack the software intending to imply a considerable production effort.

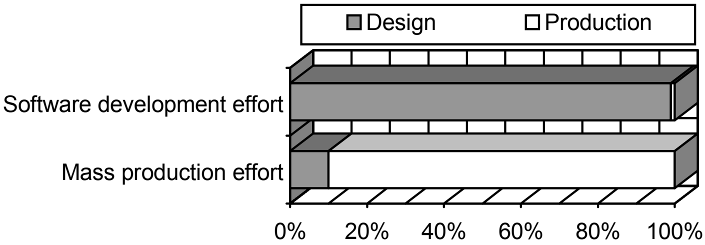


Figure 3.6 Effort distribution in software development and mass production

3.5.2.4 Implementation: Specificity vs. Generality

Nancy Leveson (1997) viewed that the computer revolution enables "machines that were physically impossible or impractical to build become feasible." Therefore, system designs may put "emphasis on steps to be achieved without worrying about how the steps will be realized physically."

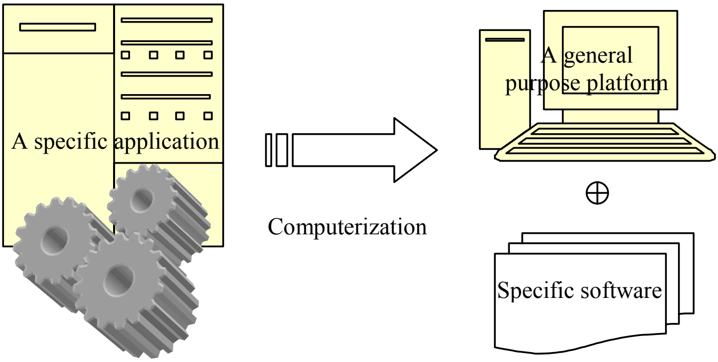


Figure 3.7 The role of software in computerization

The development of soft systems is a revolution that transforms the information processing and intelligent parts of the conventional physical products into software. Computerization as shown in Fig. 3.7 enables a specific problem or a special application be solved or implemented by a general purpose solution plus a specific software system. In this approach, a given problem is divided into two categories: (a) the standardized control drivers and interfaces to the general purpose computer, and (b) the special purpose software for the application. Therefore, a new design and implementation of a specific system is reduced to a problem of changing the software subsystem rather than that of refabricating the entire machine.

3.5.2.5 Universal Logical Description vs. Domain-Specific Description

Software engineering adopts only a few fundamental logical structures, such as sequence, branch, iteration, recursion, interrupt, and concurrency. However, these provide a powerful descriptive and abstractive capability for dealing with any real-world problem. In contrary, in other engineering disciplines, domain-and-application-specific notations have to be adopted that have limited descriptivity.

3.5.2.6 Process Standardization vs. Product Standardization

Directly related to the fact that software engineering is design intensive, it is recognized that the development of specific application software is characterized as mainly a one-off activity in design and production. This is because there are fewer standard software applications or products that can be mass produced except a few kinds of system software or general utilities.

Thus, for the design-intensive software development, the only elements that can possibly be standardized and reused significantly are mainly the software engineering principles and processes, not the final products themselves as in other manufacturing engineering disciplines.

3.5.3 SOFTWARE ENGINEERING VALIDATION METHODOLOGIES

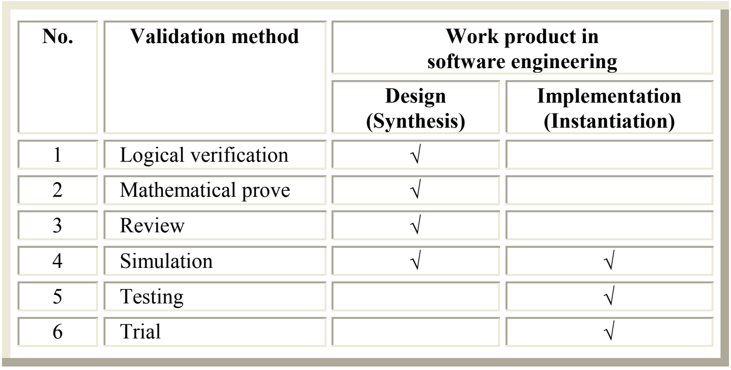
On the basis of rationalism, it can be seen that the methodology for software engineering validation and software quality assurance are profoundly different from the physical engineering disciplines, because the objects under study in the former are abstract and behavioral information.

According to Theorem 3.2, the following corollaries can be derived.

Corollary 3.4 The *validation methodologies* of software design and implementation can be sufficiently categorized as shown in Table 3.4.

Corollary 3.4 indicates that the validation methodologies in software design and implementation are totally different. As shown in Table 3.4, the validation of the work products of software designs can be: a) Formal verification on the basis of logic and formal models of the design; b) Review or inspection of the design; and/or c) Simulation or prototyping. However, software implementations can only be validated by simulation, testing, and/or empirical trial.

Table 3.4
Methodologies for Validating Software Products



No.	Validation method	Work product in software engineering	
		Design (Synthesis)	Implementation (Instantiation)
1	Logical verification	√	
2	Mathematical prove	√	
3	Review	√	
4	Simulation	√	√
5	Testing		√
6	Trial		√

Corollary 3.4 also indicates that software implementation itself is actually a validation method for the design of the given software system. This is an inherent property of complex systems where a long chain of processes is adopted in the design and implementation lifecycle to validate the system. Therefore, no simple or single process may be adequate in software system validation. As Brooks said, there is “no silver bullet.”

The philosophical considerations explored in this subsection have attempted to clarify a set of fundamental characteristics of software engineering. Based on the above discussion it might be argued that software engineering has become a discipline that is at the root of human knowledge structure, and it deals with the most abstract and complex objects among all disciplines. These considerations also provide a basis for judging soundness or unsoundness of any specific technical solution for software engineering, while not losing the sight of the woods for the trees in various practice.

3.6 Murphy's Laws: The Practitioners' Philosophy for Software Engineering

In the preceding sections it may be seen that important philosophical thoughts were contributed by preeminent philosophers and scientists, who think at the highest level of abstraction and in a systematical way. However, this is not necessary to say that philosophy is only the brainchild of philosophers and scientists. Actually, ordinary engineers and practitioners do recognize the usefulness of philosophy in eliciting the generic truth from everyday life. Murphy’s laws presented in this section are good examples of people’s philosophy elicited from empirical practice.

The first Murphy's Law was named after Edward A. Murphy, an engineer working at Edwards Air Force Base in the 1940s. It says: “If anything can go wrong, it will.” A manager kept it as one of the “laws” called Murphy's Law. This law has helped the organization to maintain a good safety record for years.

Murphy's laws are people's wisdom that represents a number of simple and intuitive philosophical views on engineering practice. There are hundreds of Murphy's laws proposed and posted on the Internet. Selected ones are provided in the following subsections.

3.6.1 MURPHY'S LAWS ON GENERAL ENGINEERING

The following is a list of selected Murphy's laws on generic engineering practice. Readers may see they are also useful for daily life.

- If anything can go wrong, it will.
- Nothing is as easy as it looks.
- Everything takes longer than one expected.
- The complexity and frustration factor is inversely proportional to how much time you have left to finish a project.
- Experience is something you do not get until just after you needed it most.
- Confidence is the feeling you get just before you fully understand the problem.
- He who laughs last probably made a back-up.
- You will always discover errors in your work after you have printed/submitted it.
- The troubleshooting guide contains the answer to every problem except yours.
- He who hesitates is probably smart.
- If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.
- Great discoveries are made by mistake.

- A meeting is an event at which the minutes are kept and the hours are lost.
- Any system which depends on human reliability is unreliable.
- If an experiment works, something has gone wrong.
- The remaining work to finish in order to reach your goal increases as the deadline approaches.
- Never trust modern technology. Trust it only when it is old enough.
- It is simple to make something complex, and complex to make it simple.
- Impossible failures will happen at the test site.

3.6.2 MURPHY'S LAWS ON SOFTWARE ENGINEERING

The following is list of selected Murphy's laws related to software engineering.

- A program will always do what you tell it to do, but never what you want it to do.
- Program complexity grows until it exceeds the capability of the programmer who must maintain it.
- Every nontrivial program has at least one bug.
- The subtlest bugs cause the greatest damage and problems.
- Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited.
- Adding labor to a late software project makes it later.
- A working program is one that has only unobserved bugs.
- No matter how much resources you have, it is never enough.
- It is futile to try to get more disk space. Data expands to fill any void.
- A failure in a device will never appear until it has passed final inspection.
- Computers don't make errors. What they do they do on purpose.
- Every nontrivial program can be simplified by at least one line of code.

- For any given software, the moment you manage to master it, a new version appears. And the new version always manages to change the very feature that you need most.
- A patch is a piece of software which replaces old bugs with new bugs.
- The chances of a program doing what it's supposed to do are inversely proportional to the number of lines of code used to write it.
- Failure is not an option, it's a feature with software.
- The worst bugs in your program will show up only during the final review.
- The likelihood of problems occurring is inversely proportional to the amount of time remaining before the deadline.
- The error is human. To blame you computer for your mistakes is even more human.
- A complex system that does not work is invariably found to have been evolved from a simple system that worked well.
- You can always spot an expert in the crowd. It is the person who says that the project will take the longest to complete and will cost the most.
- Investment in software reliability will increase until it exceeds the probable cost of errors.

More and newly proposed Murphy laws may be found at a number of websites such as: http://www.murphys-laws.com/, http://www.fourjokers.co.uk/murphy/, and http://www,hardcorenyc.com/murphys_law_main.htm, etc.

3.7 Summary

Philosophy is the tool of abstraction, synthesis, and deduction, which enables new theories to be developed when there are inadequate laws or intuitive facts to be based for reasoning and draw inductive conclusions. Philosophy is the highest level of knowledge that is universally true without regard of time and places. **Science and engineering philosophy** is an aspect of philosophy that studies general phenomena and rules of sciences and engineering technologies.

The **philosophical foundations of software engineering** highlight the relationship between the abstract world that deals with information and the physical world that deals with matter and energy. The problem domain of software engineering can be seen in the connection between the abstract world of information to the physical world of matter and energy.

This chapter has investigated the philosophical foundations and logical means of software engineering. Philosophies of science and engineering with inspiring philosophical thought have been surveyed. A set of formal inference methodologies based on logical arguments, deduction, induction, abduction, and analogy has been explored. The nature of software and its properties have been examined from a philosophical view. The philosophy of software engineering, complemented by Murphy's laws – the practitioners' philosophy, has been presented. As a result, the **philosophical foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Philosophical Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge architecture as summarized below.

Chapter 3. Philosophical Foundations of SE

- Philosophies of Science and Engineering
    - The physical world vs. the abstract world
    - The basic axioms about nature
    - Epistemology and cognition
    - Holism vs. reductionism
    - Positivism vs. rationalism
    - Empiricism and objectivity
    - Determinism vs. indeterminism
    - Approaches to implement intelligence
    - Ethical philosophy of engineering
- Formal Inference Methodologies
    - Logical argumentations
    - Deductive inferences
    - Inductive inferences

- Abductive inferences
- Analogical inferences

■ The Nature of Software

- The three situations where software is needed
- The behavioral space of software
- Properties of software
  - The cognitive properties
  - The intelligent behavioral properties
  - The system properties

■ The Philosophy of Software Engineering

- The cognitive characteristics of software engineering
  - The abstraction and intangibility
  - The inherited complexity and diversity
  - The changeability or malleability
  - The difficulty of establishing and stabilizing requirements
  - The requirements of varying problem domain knowledge
  - The indeterminacy and polysolvability in design
  - The polyglotics and polymorphism in implementation
  - The dependability of interactions between software, hardware, and humans

- The nature of software engineering
  - Programming: virtualization vs. realization
  - Problem domains: infinitive vs. limited
  - Effort distribution: design intensive vs. repetitive production
  - Implementation: specificity vs. generality
  - Universal logical description vs. domain-specific description
  - Process standardization vs. product standardization

- Software engineering validation methodologies

■ Murphy's Laws: The Practitioners' Philosophy for Software Engineering

- Murphy's laws on general engineering
- Murphy's laws on software engineering

SIGNIFICANT FINDINGS OF THIS CHAPTER

• **Philosophy is needed** when there are inadequate laws or lack of intuitive facts to be based for reasoning and draw inductive conclusions. It is also needed when there is no other clue for the judgment of a new problem or technology.

- The **philosophy of software engineering** reveals that software engineering is a discipline at the root of human knowledge structure, and it deals with the most abstract and complex objects among all engineering disciplines.

- **Theoretical research** is predominantly an **inductive process**; while **applied research** is mainly a **deductive process**. Both inference processes are based on the cognitive process and means of abstraction.

- Philosophy, as well as mathematics, is the **top level abstraction means** and therefore the **most general human knowledge**.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Philosophies of Science and Engineering

- **Philosophy** can be divided into four branches known as *epistemology*, *metaphysics*, *logic*, and *ethics*. **Logical inquiry** is a generic cognitive methodology for knowledge acquisition by scientific investigation that adopts the processes of problem identification, hypothesis proposing, experiment and testing, and theory forming.

- The **principle of universal constraints** states that both the natural world and the perceived abstract world are constrained by certain known restrictions and laws, or by those yet to be known due to current limitations of natural resources and/or human cognitive capability.

- **The basic axiom of scientific inquiry** is that nature obeys a set of fundamental axioms, referring to *uniformity, determinism, reality, rationality, regularity, replication,* and *discoverability* of natural events and their relations.

- **Epistemology** is a branch of philosophy that studies concepts of knowledge and their rational justification, in which the six approaches to acquire knowledge or gain cognition of the nature are: *tenacity*, *intuition*, *experience*, *authority*, *reasoning or inferring*, and *logical inquiry*.

- **Foundationalism** is a basic philosophical view that all the propositions we know to be true can be divided into *foundational* and *superstructural* ones. The former are indubitable and axiomatically to be true. The latter are propositions that bear deductive or implicated relationship to the foundations. To have the knack of reducing a problem to its simplest and basic elements and then finding a solution by the most direct means is commonly recognized as a vital scientific research method rooted in foundationalism.

- **Holism** is a philosophical view that perceives a phenomenon and system with wholeness in an integrated, synthetic, and systematic approach. **Reductionism** is a philosophical view that investigates a phenomenon and system by using a decomposition and analytic approach.

- **Positivism** is a philosophical view, which states that a thesis about physical phenomena must either be analytic or empirical. **Rationalism** is a philosophical view to arrive at knowledge in which reasoning is used to acquire, process, derive, and evaluate the knowledge.

- **Empiricism** is a philosophical view that states knowledge can be gained through the experience of an event, the observation of a fact, or the use of a methodology. **Objectivity** is a scientific criterion that requires an observation must be independent of individual opinion, bias, or prejudice. The essence of objectivity is that true or false of an objective matter is independent of what anyone believes or thinks. **Replication** is a scientific criterion which requires that the results of a study must be replicable under the same condition. **Causation** is the cause-and-effect relationships where the manipulation of one event produces another event as the effect of the causal event.

- **Determinism** is a philosophical view, which states the thesis that a complete description of the causal facts at one time uniquely determines what must happen next. **Indeterminism** is a philosophical view, which states the thesis that even a complete description of the present does not uniquely determine what will happen in the future.

- The **natural intelligence** (NI) and **machine intelligence** (AI) share the same **cognitive informatics** (CI) foundation, because AI is a machine implementation of a subset of NI.

- The **approaches to implement intelligence** can be classified into four categories known as *biological organisms, silicon automata, computing systems,* and *hybrid systems*. **Autonomic computing** is proposed as a new and advanced computing technique built upon the *routine, algorithmic,* and *adaptive systems.*

Formal Inference Methodologies

- **Inference** is a cognitive process that inferences a possible causal conclusion from given premises based on known causal relations between a pair of cause and effect proven true by *empirical observations*, *theoretical inferences*, or *statistical regulations*. Reasoning can be classified as *causal argument, deduction, induction, abduction, and analogy*.

- A **causal argument** is an assertion that yields a proposition called the conclusion from a given finite set of propositions known as the premises. The argument is valid if the conclusion is true; otherwise, the argument is a fallacy.

- **Deduction** is a cognitive process by which a specific conclusion necessarily follows from a set of general premises (Eq. 3.11). A sound deductive inference is yielded *iff* all premises are true and the argument is valid.

- **Induction** is a cognitive process by which a general conclusion is drawn from a set of specific premises based mainly on experience or experimental evidence (Eq. 3.12). A cogent inductive inference is yielded *iff* all premises are true and the argument is valid.

- **Abduction** is a cognitive process by which an inference on a causality or the most likely reason of an observation or event may be derived (Eq. 3.14). A generic inference formula of *logical abduction* states that based on a general implication, a specific conclusion can be drawn. Abduction is a powerful inference technique for seeking the most likely cause(s) and reason(s) of an observed phenomenon in causal analyses.

- **Analogy** is a cognitive process by which an inference about the similarity of the same relations holds between different domains or systems, and/or examines that if two things agree in certain respects then they probably agree in others (Eq. 3.15). A generic inference formula of *logical analogy* states that based on a specific proposition, a similar specific conclusion can be drawn. Analogy is widely used to predict a similar phenomenon or consequence based on a known observation.

The Nature of Software

- The **necessary and sufficient conditions** warranting the requirement for a software solution are the *repeatability*, *programmability*, and *run-time determinability* of system behaviors.

• The **behavior** of a computational statement is a set of observable actions or changes of status of objects operated by the statement. The **behavior space** of software Ω is a three-dimensional space with the dimensions of *operations, time,* and *memory space*.

• The **cognitive properties of software** refer to its human dependency in almost all processes of software engineering, such as *intangibility, complexity, inderterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment,* and *unquantifiable quality measures.*

• The **intelligent properties of software** refer to the subset of simulated human intelligent behaviors described by programmed instructive information. The nature of software is the simulation and execution of human behaviors, and the extension of human capability, reachability, persistency, memory, and information processing speed. Both human and software behaviors can be described by a 3-dimensional representative model comprising *action*, *time*, and *space*.

• The **system properties of software** refer to the intricate artifact that consists of a large set of different and intricately interconnected components. Changes at one point may affect the functioning of the entire whole due to propagation of interactions via highly coupled data architectures and intricately interconnected components. The system metaphor of software reveals that software in nature is an open system that can be rigorously treated using abstract system theory and system algebra.

The Philosophy of Software Engineering

• Conventional industries produce artifacts from raw materials via engineering approaches; the software industry produces software solutions for problems via software engineering. The nature of software engineering is explained by contrasting its unique characteristics with the conventional engineering disciplines. The **uniqueness of software engineering** encompasses *virtualization in both design and implementation, infinitiveness of problem domains, design-intensive, generic platform, universal logical description,* and *repeatable processes*.

• The abstract objects under study and their cognitive complexity distinguish software engineering from conventional engineering disciplines. The **cognitive characteristics of software engineering** have been identified as follows:

- The abstraction and intangibility
- The inherited complexity and diversity

- The changeability or malleability
- The difficulty of establishing and stabilizing requirements
- The requirement of varying problem domain knowledge
- The indeterminacy and polysolvability in design
- The polyglotics and polymorphism in implementation
- The dependability of interactions between software, hardware, and humans

• The **philosophy of software engineering** reveals that software engineering is a discipline at the root of human knowledge structure, and it deals with the most abstract and complex objects among all engineering disciplines.

Questions and Research Opportunities

3.1 Philosophy addresses fundamental questions of great generality and ways of reasoning. Try to explain that philosophy is the common root of all sciences and the crystallization of general knowledge of mankind in the pursuit of understanding and utilizing the natural resources and their rules.

3.2 Why is philosophy needed when there are inadequate laws and principles to be based for deductive reasoning, and when there is no other rule of thumb to be based for a judgment of a given new problem or technology?

3.3 Discuss why philosophy is the highest level of abstract knowledge that is universally true without regard of time and places.

3.4 What are the four branches of philosophy? Why is logic treated as one of the branches?

3.5 Explain why almost all science disciplines were originally emerged from philosophy. Why is it perceived that human wonder about the nature and themselves started by philosophical queries and concluded in philosophical doctrines?

3.6 Discuss the semantic differences between 'epistemology' in philosophy and 'cognition' in psychology and cognitive informatics.

3.7 Foundationalism is a philosophical view and an important scientific research method that reduces a problem to its simplest and basic elements until direct solutions are known or can be derived. Try to provide an example problem in software engineering that can be solved in the foundationalism approach.

3.8 According to Theorem 3.2, rationalism views that the abstract and information-based propositions and work products, such as a design or a specification of a system, are bounded by logical verifications, mathematical proofs, systematical reviews, behavioral simulations and tests, and/or in field trials. Try to compare rationalism with positivism, in which a thesis about physical phenomena must be empirically verifiable or repetitively observable, in a software engineering context.

3.9 Theorems 3.3 and 3.4 indicate that natural and artificial intelligence are compatible and inclusive. On the basis of this philosophy, discuss whether a programmer may design a software system, which can solve a problem that no individual in the world knows how to solve.

3.10 Why do ethics and professionalism play important roles in software engineering?

3.11 What are the differences between the deductive and inductive inference methodologies?

3.12 Explain why empirical research is mainly deductive, while scientific research is mainly inductive.

3.13 In addition to Theorem 3.6, can induction be carried out by statistical methods on the basis of multiple observations of recurring phenomena? Why?

3.14 Develop an instance of deductive inference in software engineering.

3.15 Develop an instance of inductive inference in software engineering.

3.16 Develop an instance of abductive inference in software engineering.

3.17 Develop an instance of analogical inference in software engineering.

3.18 According to Theorem 3.9, discuss why the run-time determinability is the most important condition for implementing software behaviors.

3.19 Theorem 3.10 states that the software behavior space is innately three-dimensional, i.e., $\Omega = OP \times T \times S$. However, almost all programming languages implement only two of them, but leave the time dimension T implied rather than explicitly expressed. What are the possible advantages and disadvantages of this convention in software engineering?

3.20 In order to understand the nature of software, this chapter presents the cognitive, intelligent behavior, and system metaphors of software. Chapter 1 has also presented the mathematics, product, and information metaphors.

Summarize the set of the six metaphors of software, and consider if the traditional mass-manufacturing-based methodologies and quality assurance techniques in software engineering fit with all the six metaphors that reveals the nature of software from different angles.

3.21 The concept of soft-systems is a revolution developed in computing and software engineering, which transform the information processing and intelligent parts of the conventional physical products into software. Try to explain the relationship between the universal machines and the soft-systems.

3.22 A set of software engineering validation methodologies is presented in Table 3.4, such as logical verification, mathematical prove, review, simulation, testing, and trial. On the basis of Ex. 3.8, discuss why the validation techniques should be different in software design and implementation in software engineering.

3.23 Select one of the Murphy's laws on software engineering and try to validate it by any of the following methodologies such as logical verification, mathematical prove, review, simulation, testing, trial, case study, experiment, and/or survey.

3.24 Read the following classic article in software engineering:

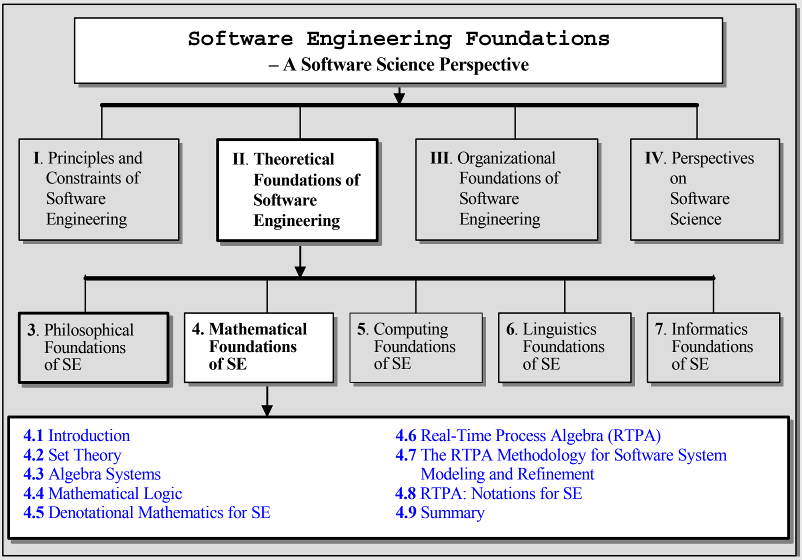
> Dennis Ritchie (1984), Reflections on Software Research, The 1983 Turing Award Lecture, *Communications of the ACM*, 27(8), pp.758-760.

Discuss the following topics in a group:

- About the author.
- What was the nature of software research in the 1980s?
- What distinguishes software research from computing and programming?
- What conclusions of the article interested you? Why?
- Your argument(s) or counter-points on any of the conclusions derived in this article.

Chapter 4

MATHEMATICAL FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective

I. Principles and Constraints of Software Engineering

II. Theoretical Foundations of Software Engineering

III. Organizational Foundations of Software Engineering

IV. Perspectives on Software Science

3. Philosophical Foundations of SE

4. Mathematical Foundations of SE

5. Computing Foundations of SE

6. Linguistics Foundations of SE

7. Informatics Foundations of SE

4. Mathematical Foundations of SE

Knowledge Structure

❍ Fundamental mathematics

- Set theory
- Functions
- Predicate logic
- Relations
- Propositional logic
- Algebraic systems

❍ Denotational mathematics for software engineering

- Fundamental elements in modeling software systems
- The need for a denotational mathematics in SE
- The big-R notation

❍ Real-time process algebra (RTPA)

- The process metaphor of software systems
- The structure of RTPA
- Meta processes of RTPA
- The type system of RTPA
- Process relations of RTPA

❍ The RTPA methodology for software system modeling and refinement

- The RTPA methodology
- System architecture modeling and refinement in RTPA
- System static behavior modeling and refinement
- System dynamic behavior modeling and refinement

❍ RTPA: notations for software engineering

- Modeling component-level problems using RTPA
- Modeling system-level problems using RTPA
- Modeling cognitive processes of the brain using RTPA

Learning Objectives

- To understand the *central role of mathematics* for software engineering.
- To be aware of the usages and *limitations of classic mathematics* for software engineering, particularly set theory, functions, and mathematical logic.
- To understand the new structure of *denotational mathematics* for software engineering, particularly real-time process algebra (RTPA).
- To be familiar with the *notation system of* RTPA and its defined algebraic operations on basic processes.
- To be familiar with the RTPA *methodology* for software engineering in system architecture, static, and dynamic behavior modeling and manipulation.
- To understand the need for a rigorous *notation system* for software engineering.

"Software development is a tough engineering discipline with a strong mathematical flavor."

Edsgar W. Dijkstra (1982)

"The basic insight is that programs themselves, as well as their specifications, are mathematical expressions. ... The great advantage of mathematics is that the rules are much simpler than those of a natural language, and the vocabulary is much smaller. Consequently, when presented with something unfamiliar it is possible to work out a solution for yourself, by logical deduction and invention rather than by consulting books or experts."

C.A.R. Hoare (1985)

4.1 Introduction

Many branches of mathematics have been created in sciences and engineering in order to meet their *abstract, rigor,* and *expressive* needs. These phenomena may be conceived as that new problems require new forms of mathematics.

The entire computing theory, as Lewis and Papadimitriou (1998) perceived, is about mathematical models of computers and algorithms. Hence, the entire theory of software engineering is about mathematical models of software systems and denotational mathematics for software engineering.

Applied mathematics can be classified into two categories: *analytic* and *denotational* mathematics [Wang, 2002a]. The former are mathematical structures that deal with functions of variables and their operations and behaviors; while the latter are mathematical structures that formalize rigorous expressions and inferences of system architectures and behaviors with data, concepts, and dynamic processes. Denotational mathematics also provides a formal semantics for other forms of means based on diagrams or languages in software engineering.

The problems of software engineering are large-scaled and at the system level with abstract, complex architectures, and long chains of computing behaviors. Therefore, denotational mathematics is a system-level mathematics in which detailed individual computing behaviors may still be modeled by conventional analytical mathematics. Typical forms of denotational mathematics [Wang, 2002a/06d/06e/06j/07a] are system algebra, concept algebra, and Real-Time Process Algebra (RTPA). This chapter covers RTPA, while the other denotational mathematics will be presented in Chapters 10 and 15, respectively.

The 11th Law of Software Engineering

Theorem 4.1 *The utility of mathematics* in software engineering states that denotational mathematics is the means and rules to rigorously and explicitly express design notions and conceptual models on abstract architectures and complex interactive behaviors at the highest level of abstraction and in the largest scope of systems.

This chapter explores essential elements of mathematics for modeling software architectures and software system behaviors encompassing fundamental mathematics such as set theory, algebra systems, and mathematical logic, as well as contemporary denotational mathematics such as process algebra [Mills, 1975; Hoare, 1978/85; Milner, 1980/89] and RTPA [Wang, 2002a/02b/03c/07a], which provides an essential formal notation for software engineering.

In the remainder of this chapter, existing mathematical means in terms of set theory, Boolean algebra, mathematical logic, and their applications in software engineering are orientated in Sections 4.2 through 4.4. The findings of their inadequacy in dealing with software engineering problems reveal an age-long overlooked problem in software engineering: the lack of a denotational and adequate mathematical means. This profound issue is explored further based on the analysis of essential elements of mathematics for modeling software systems and behaviors in Section 4.5. Then, a new mathematical structure, RTPA, is introduced in Section 4.6 as an expressive and practical notation system and methodology for rigorous treatment of software engineering problems. The RTPA methodology for software system modelling and refinement is presented in Section 4.7. Then, the usages of RTPA as a least complete yet powerful set of notations for software engineering modelling are described in Section 4.8 with case studies at system and component levels.

4.2 Set Theory

Abstraction and categorization of external or internal objects in order to form concepts for reasoning are basic cognitive processes of human beings. Sets are the mathematical means for modeling such abstract objects, and not a surprise, it is also the foundation of almost all mathematical branches. This section briefly introduces set theories in the context of computing and software engineering. Detailed descriptions of set theory may be referred to Lipschutz (1964), and Arnold and Guessarian (1996).

4.2.1 SETS AND PROPERTIES

Set theory was created by Cantor in 1895. A set can be viewed as a collection of objects.

4.2.1.1 Set Notations and Terminologies

Definition 4.1 A *set S* is a collection of elements *e* with a common property *p*, denoted by:

$$S \triangleq \{e \mid p(e)\} \tag{4.1}$$

where $\triangleq$ denotes a *definition*, and an expression following the vertical bar | defines the constraints or membership conditions of an element *e*, and *p*(*e*) means each *e* of *S* possesses the property *p*, i.e., $\forall e \in S \Leftrightarrow p(e)$.

Set can be used to express types of data structures, class models, and program syntaxes in software engineering. For examples, some fundamental sets in software engineering are as follows:

- The set of natural numbers: $\mathbf{N} = \{n \mid n \text{ is a positive integer}\} = \{1, 2, 3, ...\}$ (4.2)
- The set of integers: $\mathbf{Z} = \{z \mid z \text{ is an integer}\} = \{..., -2, -1, 0, 1, 2, ...\}$ (4.3)
- The set of real numbers: $\mathbf{R} = \{r \mid r \text{ is a real number}\} = \{-\infty, ..., +\infty\}$ (4.4)
- The set of byte: $\mathbf{B} = \{b \mid b \text{ is a byte of binary numbers}\} = \{0, 1, ..., 255\}$ (4.5)

It is noteworthy that in software engineering, the mathematical domain of natural number $\mathbb{N}_0$ is usually extended to include zero because of the binary system convention for data representation, i.e.:

$$\begin{aligned} \mathbb{N}_0 &= \{n \mid n \text{ is 0 or a positive integer}\} \\ &= \{0, 1, 2, 3, ...\} \end{aligned} \tag{4.6}$$

Definition 4.2 The *membership* between an element e and a set S can be determined by checking if e belongs to S or not, denoted by $e \in S$ and $e \notin S$, respectively.

Definition 4.3 Some important *relationships* between two sets A and B can be defined below:

- *Subset*: $A \subseteq B \triangleq \forall a \in A \Rightarrow a \in B$ (4.7)
- *Superset*: $A \supseteq B \triangleq \forall b \in B \Rightarrow b \in A$ (4.8)
- *Equal:* $A = B \triangleq A \subseteq B \wedge B \subseteq A$ (4.9)
- *Proper subset:* $A \subset B \triangleq A \subseteq B \wedge A \neq B$ (4.10)
- *Power set:* $\text{Þ}A \triangleq \{A_i \mid A_i \subseteq A \wedge 1 \leq i \leq 2^{\#\text{A}}\}$ (4.11)

where $\Rightarrow$ denotes an implication, and Þ a power set.

Set is a fundamental and powerful mathematical concept for abstracting and eliciting objects that share certain common properties. *Abstraction* is an elicitation of common properties of elements from a given set.

The 10th Principle of Software Engineering

Theorem 4.2 The *principle of abstraction* states that, given an arbitrary set S and any property p, abstraction is to elicit a subset E such that the elements of it, e, possess the property $p(e)$, i.e.:

$$\forall S, p \Rightarrow \exists E \subseteq S, \forall e \in E \wedge p(e) \tag{4.12}$$

Definition 4.4 A *universal set U* is a superset of all sets under investigation.

The universal set U may be an infinite set, but without itself as a member. Otherwise, it may result in a number of fundamental dilemmas

[Lipschutz, 1964]. The universal set U may be used to denote the environment or context of a software system.

Definition 4.5 The *empty set* $\varnothing$ is a set that contains no element.

4.2.1.2 Set Operations

The basic set operations are *union, intersection, difference,* and *cardinal size*. Useful operations derived from the basic operations are *complement, symmetric difference, Cartesian product*, and *partition*. Table 4.1 provides a summary of useful set operations collectively known as *algebra of sets*. Each operation in Table 4.1 is illustrated by an example using the following three sets: $A = \{1, 2, 3\}$, $B = \{3, 4\}$, and $U = \mathbb{N}$.

Table 4.1
Definitions of Basic Set Operations

Operation	Definition	Example
Union	$X \cup Y \triangleq \{e \mid e \in X \vee e \in Y\}$	$A \cup B = \{1, 2, 3, 4\}$
Intersection	$X \cap Y \triangleq \{ e \mid e \in X \wedge e \in Y\}$	$A \cap B = \{3\}$
Difference	$X \setminus Y \triangleq \{ e \mid e \in X \wedge e \notin Y\}$	$A \setminus B = \{1, 2\}$
Cardinal size	$\# X \triangleq \sum(1 \mid e \in X)$	$\#A = 3$ $\#B = 2$
Complement	$\overline{X} \triangleq U \setminus X$ $= \{ e \mid e \in U \wedge e \notin X\}$	$\overline{A} = \mathrm{U} \setminus \mathrm{A} = \mathbb{N} \setminus \mathrm{A}$ $= \{4, 5, 6, ...\}$
Symmetric difference	$X \oplus Y \triangleq (X \cup Y) \setminus (X \cap Y)$ $= \{e \mid e \in X \vee e \in Y \wedge e \notin X \cap Y\}$	$\mathrm{A} \oplus B = \{1, 2, 4\}$

The set operations *union, intersection*, and *symmetric difference* defined in Table 4.1 can be extended to multiple finite sets as follows:

$$\begin{aligned} S_1 \cup S_2 \cup \ldots \cup S_n &= \bigcup_{i=1}^{n} S_i \\ &= \{s \mid s \in S_1 \vee s \in S_2 \vee \ldots \vee s \in S_n\} \end{aligned} \tag{4.13}$$

$$\begin{aligned} S_1 \cap S_2 \cap \ldots \cap S_n &= \bigcap_{i=1}^{n} S_i \\ &= \{s \mid s \in S_i \wedge s \in S_2 \wedge \ldots \wedge s \in S_n\} \end{aligned} \tag{4.14}$$

$$S_1 \oplus S_2 \oplus \ldots \oplus S_n = \{s \mid s \in \bigcup_{i=1}^{n} S_i \wedge s \notin \bigcap_{i=1}^{n} S_i\} \tag{4.15}$$

Definition 4.6 A *partition* of a set S, $S \neq \varnothing$, is a subdivision of S into n, $n \geq 2$, subsets S_i, $1 \leq i \leq n$, such that:

$$\text{(a) } S = \bigcup_{i=1}^{n} S_i \tag{4.16}$$

$$\text{(b) } S_i \neq \varnothing,\ 1 \leq i \leq n \tag{4.17}$$

$$\text{(c) } S_i \cap S_j = \varnothing,\ i \neq j,\ 1 \leq i, j \leq n \tag{4.18}$$

where S_i is called a *cell.*

Partition is a useful concept in component-based software engineering, in which a *component* can be modeled as a cell or a nonempty and non-overlapping subset of a software system. Definition 4.6 also explains that a component-based system is a composition (union) of its components as partitions, which meets the three conditions.

Example 4.1 Given set $A = \{1, 2, 3\}$, all possible partitions of A can be derived according to Definition 4.6 as follows:

$A = \{\{1\}, \{2\}, \{3\}\}$, or
$A = \{\{1\}, \{2, 3\}\}$, or
$A = \{\{1, 2\}, \{3\}\}$, or
$A = \{\{1, 3\}, \{2\}\}$

Definition 4.7 Let A and B be two arbitrary nonempty sets, the *Cartesian product* of $A \times B$ is a set of all ordered pairs (a, b) where $a \in A$ and $b \in B$, i.e.:

$$A \times B \triangleq \{(a, b) \mid a \in A \wedge b \in B\} \tag{4.19}$$

where × reads *cross*, and an ordered pair (a, b) is a directed connection from a to b.

An important property of a Cartesian product is that the number of its elements (ordered pairs) is predictable, i.e.:

$$\#(A \times B) = \#A \bullet \#B \tag{4.20}$$

Eq. 4.20 indicates that if the cardinal sizes of the finite and nonempty sets A and B are known, the number of combinations between their elements is determined.

Example 4.2 Given sets $A = \{x, y, z\}$ and $B = \{1, 2\}$, the size of the Cartesian product $A \times B$ can be predicated as follows:

$$\#(A \times B) = \#A \bullet \#B = 3 \bullet 2 = 6$$

where the 6 pairs of $A \times B$ are:

$$A \times B = \{(x,1), (x, 2), (y, 1), (y, 2), (z, 1), (z, 2)\}$$

Definition 4.7 can be extended to multiple finite numbers of sets, where each set is nonempty and finite.

Definition 4.8 *Cartesian product* of n sets $S_1 \times S_2 \times \ldots \times S_n$ is a set of n-tuples $(s_1, s_2, \ldots, s_n)$ where $s_i \in S_i$, $1 \le i \le n$, i.e.:

$$\begin{aligned} S_1 \times S_2 \times \ldots \times S_n &\triangleq \bigtimes_{i=1}^{n} S_i \\ &= \{(s_1, s_2, \ldots, s_n) \mid s_i \in S_i, 1 \le i \le n\} \end{aligned} \qquad (4.21)$$

where

$$\#(S_1 \times S_2 \times \ldots \times S_n) = \prod_{i=1}^{n} \# S_i \qquad (4.22)$$

Cartesian products have a wide range of applications in modeling software systems and their behaviors.

4.2.1.3 Algebraic Laws of Sets

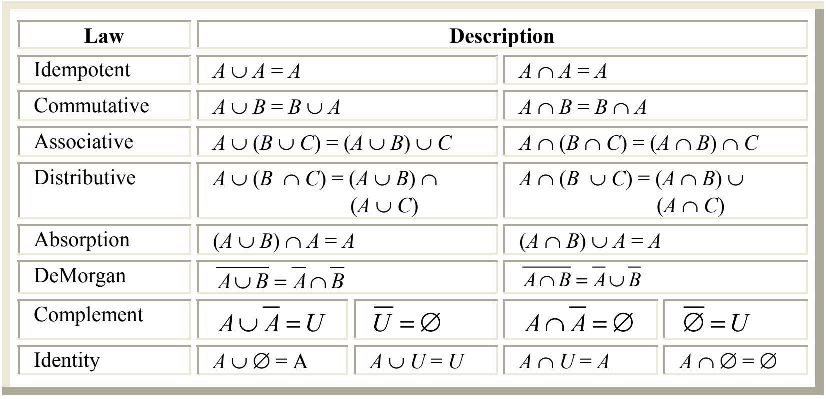
Algebra of sets obeys the laws as summarized in Table 4.2. These algebraic laws play important roles to simplify set operations and to compose complex set relations.

Observing Table 4.2 it can be found that the set operations of union $\cap$ and intersection $\cup$ are *symmetric* on arbitrary sets including the dual of sets U and $\varnothing$.

There are additional laws and properties on sets [Lipschuts, 1964] such as the *involution* law that states:

$$\overline{(\overline{A})} = \mathrm{A} \qquad (4.23)$$

Table 4.2
Laws of Set Algebra



Law	Description			
Idempotent	$A \cup A = A$		$A \cap A = A$	
Commutative	$A \cup B = B \cup A$		$A \cap B = B \cap A$	
Associative	$A \cup (B \cup C) = (A \cup B) \cup C$		$A \cap (B \cap C) = (A \cap B) \cap C$	
Distributive	$A \cup (B \ \cap C) = (A \cup B) \cap (A \cup C)$		$A \cap (B \ \cup C) = (A \cap B) \cup (A \cap C)$	
Absorption	$(A \cup B) \cap A = A$		$(A \cap B) \cup A = A$	
DeMorgan	$\overline{A \cup B} = \overline{A} \cap \overline{B}$		$\overline{A \cap B} = \overline{A} \cup \overline{B}$	
Complement	$A \cup \overline{A} = U$	$\overline{U} = \varnothing$	$A \cap \overline{A} = \varnothing$	$\overline{\varnothing} = U$
Identity	$A \cup \varnothing = \mathrm{A}$	$A \cup U = U$	$A \cap U = A$	$A \cap \varnothing = \varnothing$

4.2.2 SEQUENCES AND ORDERED SETS

According to Definition 4.1, the position, or the sequential order, of an element in a set has no meaning, i.e., $\{a, b, c\} = \{c, b, a\}$. However, in some special contexts, the positions of elements do represent important information. In order to deal with such requirements, the concepts of tuple, sequence, and ordered set are introduced in this subsection.

4.2.2.1 Pairs and Tuples

Definition 4.9 A *pair* p is an ordered encapsulation of two objects a and b, or a directed connection from a to b, denoted by:

$$p \triangleq (a, b) \qquad (4.24)$$

where the positions of the elements in p are sensitive, that is, $(a, b) \neq (b, a)$.

Definition 4.10 A *tuple* τ is an ordered encapsulation of multiple objects denoted by:

$$\tau \triangleq (A, B, \ldots, N) \qquad (4.25)$$

where the objects in the tuple can be a number, set, or function.

A tuple encapsulated with n objects is called an n-tuple. In particular, the 3- through 6-tuples are called *triples, quadruples, quintuples*, and *sextuples*, respectively.

The tuple is a powerful modeling means in mathematics and software engineering for denoting a coherent encapsulation or composition of multiple objects. Usually, the objects in the tuple will be further characterized by their attributes or properties.

Example 4.3 The syntax of a statement *s* in a program can be described as a triple, i.e.:

$$s = (I, P, O) \tag{4.26}$$

where

- *P* is a specific operator, $p \in \Xi$, where Ξ denotes the instruction set of a given language;
- *I* is a finite set of input; and
- *O* is a finite set of output.

4.2.2.2 Sequences

Definition 4.11 A *sequence q* is a finite or infinite ordered set where each element e_n, n $\in \mathbb{N}$, is identified by its cardinal position in the set, i.e.:

$$\begin{aligned} q &\triangleq <e_n \mid n \in \mathbb{N}> \\ &= <e_1, e_2, e_3, \ldots> \end{aligned} \tag{4.27}$$

where $\mathbb{N}$ is the set of natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$.

In computing, the set $\mathbb{N}$ is usually extended to $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$ for convenience. If there is no ambiguity, $\mathbb{N}_0$ will not be specially denoted in this book.

4.2.2.3 Lists

Definition 4.12 A finite sequence with *n* elements $e_1, e_2, \ldots, e_n$ is a *list l*, i.e.:

$$l \triangleq <e_1, e_2, \ldots, e_n> \tag{4.28}$$

Example 4.4 A linear (sequential) procedure *P* in a program with *n* statements, $s_1, s_2, \ldots, s_n$, can be described as a list l_P, i.e.:

$$l_P = <s_1, s_2, \ldots, s_n> \tag{4.29}$$

4.2.2.4 Ordered Sets

An ordered set is a set that the positions of its elements satisfy a certain condition. A set of natural or real numbers with the usual order is described below.

Definition 4.13 A *partially ordered set* $S_{\leq}$ is a set in which all elements are listed according to the usual order or their values in an ascending sequence, i.e.:

$$S_P \triangleq \{e_i \mid i \leq j \wedge i, j \in \mathbf{N} \Rightarrow e_i \leq e_j \wedge e_i, e_j \in \mathbf{R}\} \qquad (4.30)$$

Example 4.5 A sort operation on a set $A = \{5, 2, 1, 5, 8\}$ transforms it into a partially ordered set $A_{\leq} = \{1, 2, 5, 5, 8\}$.

More general treatment of ordered sets is dependent on the definition of partially ordered relations, which will be described in Section 4.2.3.3.

4.2.3 RELATIONS

Relation is the most important concept in programming theories, because a program can be modeled as a finite list of relations between individual statements. Relations also play an important role in explaining human internal knowledge representation and the natural intelligence. This section describes the basic mathematical theory of generic relations. The relational theory will be further extended in Section 4.6.5 on RTPA and Section 5.5.1 on the mathematical models of programs and software systems.

4.2.3.1 Binary Relations

Definition 4.14 Let A and B be sets, a *binary relation* $R(a, b)$ is an ordered pair $(a, b) \in A \times B$, i.e.:

$$\begin{aligned} R(a, b) &\triangleq aRb \\ &= (a, b), \quad a \in A, b \in B \end{aligned} \qquad (4.31)$$

Since the pair (a, b) is ordered, a relation $aRb \neq bRa = aR^{-1}b$, where R^{-1} is an *inverse relation* of R.

Usually, a binary relation is simply called a *relation*.

Example 4.6 Given sets $A = \{x, y, z\}$ and $B = \{1, 2\}$, a set of six binary relations can be derived as:

$$\begin{aligned} R(a, b) &= \{R_1, R_2, R_3, R_4, R_5, R_6\} \\ &= \{(x,1), (x, 2), (y, 1), (y, 2), (z, 1), (z, 2)\} \end{aligned}$$

4.2.3.2 Compositions of Relations

The binary relation defined in Eq. 4.31 can be extended to a *ternary* or, in general, an *n-nary* relation. The operation that constructs a combinational relation with more than one binary relation is called a composition.

Definition 4.15 Let A, B, and C be sets. Then, a *composition of two relations aPb* and bQc, $R(\text{a}, b, c)$, is denoted by $P \circ Q$, i.e.:

$$\begin{aligned} R(\text{a}, b, c) &\triangleq P \circ Q \\ &= (aPb) \circ (bQc) \\ &= (aPb)Qc \\ &= (\text{a}, b, c), \quad a \in A, b \in B, c \in C \end{aligned} \tag{4.32}$$

Example 4.7 Given sets $A = \{x, y, z\}$ and $B = \{1, 2\}$, $\text{C} = \{\alpha, \beta\}$, a set of 12 ternary relations can be derived according to Definition 4.15:

$$\begin{aligned} R(a, b, c) &= R(a,b) \circ R(b,c) \\ &= \{(x, 1), (x, 2), (y, 1), (y, 2), (z, 1), (z, 2)\} \times \{\alpha, \beta\} \\ &= \{(x, 1, \alpha), (x, 2, \alpha), (y, 1, \alpha), (y, 2, \alpha), (z, 1, \alpha), (z, 2, \alpha), \\ &\quad (x, 1, \beta), (x, 2, \beta), (y, 1, \beta), (y, 2, \beta), (z, 1, \beta), (z, 2, \beta)\} \end{aligned}$$

Definition 4.16 A *composition of n relations* $s_1R_{12}s_2$, $s_2R_{23}s_3$, ..., $s_{n-1}R_{n-1,n}s_n$, denoted by $R(s_1, s_2, ..., s_n)$, is an ordered n-tuple $(s_1, s_2, ..., s_n) \in S_1 \times S_2 \times \ldots \times S_n$, i.e.:

$$\begin{aligned} R(s_1, s_2, ..., s_n) &\triangleq R_{12} \circ R_{23} \circ \ldots \circ R_{n-1,n} \\ &= (\ldots ((s_1R_{12}s_2)R_{23}s_3) \ldots s_{n-1})R_{n-1,n}s_n \\ &= (s_1, s_2, ..., s_n), \quad s_i \in S_i, 1 \le i \le n \end{aligned} \tag{4.33}$$

Example 4.8 A string **S** can be denoted as a list of characters α_i, $1 \le i \le n$, composed by the *concatenation* relation $\frown$, i.e.:

$$\begin{aligned} \mathbf{S} &= \alpha_1 \frown \alpha_2 \frown \ldots \frown \alpha_n \\ &= \alpha_1\alpha_2 \ldots \alpha_n \end{aligned} \tag{4.34}$$

Because any arbitrary *n*-nary relation can be reduced to *n*-1 embedded binary relations, the following subsections will be focused on operations and properties of binary relations.

4.2.3.3 Properties of Relations

Major properties of relations, such as associative, reflexive, symmetric, and transitive, are summarized in Table 4.3, where R, R_1, R_2, and R_3 are *relations,* a, b, c are elements in set S, respectively.

Table 4.3
Properties of Relations

Property	Description
Associative	$(R_1 \circ R_2) \circ R_3 = R_1 \circ (R_2 \circ R_3)$
Reflexive	$\forall a \in S \Rightarrow aRa$
Irreflexive	$\forall a, b \in S, aRb \Rightarrow a \neq b$
Symmetric	$\forall a, b \in S, aRb \Rightarrow bRa$
Asymmetric	$\forall a, b \in S, aRb \wedge bRa \Rightarrow a = b$
Transitive	$\forall a, b, c \in S, aRb \wedge bRc \Rightarrow aRc$

Based on the basic properties of relations, two categories of relations known as equivalence and partial order relations can be derived.

Definition 4.17 An *equivalence relation* is a relation R on a nonempty set S satisfying the following properties:

(a) R is *reflexive*, i.e., $\forall a \in S \Rightarrow aRa;$
(b) R is *symmetric*, i.e., $\forall a, b \in S, aRb \Rightarrow bRa;$
(c) R is *transitive*, i.e., $\forall a,b,c \in S, aRb \wedge bRc \Rightarrow aRc.$

Definition 4.18 A *partially ordered relation* R_p is a relation R on a nonempty set S satisfying the following properties:

(a) R is *reflexive*, i.e., $\forall a \in S \Rightarrow aRa;$
(b) R is *asymmetric*, i.e., $\forall a, b \in S, aRb \wedge bRa \Rightarrow a = b;$
(c) R is *transitive*, i.e., $\forall a,b,c \in S, aRb \wedge bRc \Rightarrow aRc.$

Typical partially ordered relations are $\leq$, $\geq$, $<$, $>$, and $\subseteq$.

Definition 4.19 A *partially ordered set* S_p is a set in which all elements satisfy a given partially ordered relation R_p.

Typical partial ordered sets are $\mathbf{N}_{\leq}$, $\mathbf{N}_{\geq}$, $\mathbf{N}_{<}$, $\mathbf{N}_{>}$, $\mathbf{R}_{\leq}$, $\mathbf{R}_{\geq}$, $\mathbf{R}_{<}$, $\mathbf{R}_{>}$, and $S_{\subseteq}$.

Definition 4.20 A *totally ordered relation* R_t is a relation R on a nonempty set S that is a partial order; in addition, every two elements in it are comparable.

According to Definition 4.20, $\mathbf{N}_{\leq}$, $\mathbf{N}_{\geq}$, $\mathbf{N}_{<}$, $\mathbf{N}_{>}$, $\mathbf{R}_{\leq}$, $\mathbf{R}_{\geq}$, $\mathbf{R}_{<}$, and $\mathbf{R}_{>}$ are also totally ordered sets, since any pair of elements a and b in them are comparable by $a < b$, $a = b$, or $a > b$.

4.2.3.4 Cumulative Relations of Programs

A program can be treated as a composition of a list of statements by predefined relational or composing rules. The relations between statements are a special type relation known as *cumulative relations* [Wang, 2006a/06h/06j], that is, a relation R_i is related to all previous relations R_1 through R_{i-1}, $1 \leq i \leq n$, as defined below.

Definition 4.21 A *cumulative relation* ® is an ordered list of embedded relations where a relation R_{ij}, $j = i + 1$, $1 \leq i < n-1$, $1 < j \leq n$, is related to all previous relations R_{12} through $R_{i-1,j}$, i.e.:

$$
\begin{aligned}
®(s_1, s_2, \ldots, s_n) &= (\ \ldots ((R_{12}) \circ R_{23}) \circ \ldots) \circ R_{n-1,n} \\
&= (\ \ldots ((s_1 R_{12} s_2) R_{23} s_3) \ \ldots\ s_{n-1}) R_{n-1,n}\, s_n, \\
&\qquad s_i \in \Xi,\ R_{ij} \in \Re \qquad (4.35)
\end{aligned}
$$

where Ξ is a set of predefined instructions in a given programming language, and $\Re$ a set of designated compositional rules in the same language.

Definition 4.21 indicates that program composition is left associative. The finding on the cumulative relations for modeling composing rules in programming will be further discussed in Section 4.6 on RTPA and Section 5.5.1 on the unified mathematical models of programs.

The composing rules $\Re$ in programming can be classified into sequential, branch, switch, iteration, procedure call, recursion, parallel, concurrence, interleave, pipeline, interrupt, jump, and system dispatches. Detailed descriptions will be provided in Section 4.6.5 known as the 17 fundamental process relations in RTPA.

4.3 Algebra Systems

In the preceding section, set theory has been described as the foundation not only for the entire mathematical family, but also for the modeling and manipulation of software objects and software system behaviors. However, only sets and their operations are not adequate and convenient in dealing with the whole scope of problems in software engineering, particularly the intricate interrelations among software objects. This section describes algebra and algebraic operations on functions, which are an extended mathematical concept beyond sets.

4.3.1 ABSTRACTION IN ALGEBRA SYSTEMS

Definition 4.22 *Algebra* is a branch of mathematics in which objects and their relations are represented by abstract symbols and formulae.

Abstraction as described in Theorem 4.2 is the essence of algebra. Using algebra, generic relations between variables and quantities may be formally, precisely, and efficiently described. Rigorous reasoning can then be conducted based on established algebraic rules and properties.

By extending the objects under study and their relations beyond sets, a number of advanced and special algebraic systems are developed, such as abstract algebra, Boolean algebra, process algebra, concept algebra, and system algebra.

4.3.1.1 Abstract Algebra

Definition 4.23 *Abstract algebra* studies a set of abstract algebraic structures beyond sets, such as semigroups, groups, rings, fields, and lattices.

Discussions on these algebraic structures are out of the scope of fundamental requirements for software engineering. Detailed materials may be referred to Arnold and Guessarian (1996) and Lipschutz and Lipson (1997).

4.3.1.2 Boolean Algebra

Definition 4.24 *Boolean algebra* is an abstract algebraic system on binary valued entities developed by George Boole (1813 - 1864).

Boolean algebra is built on the axioms of the algebraic laws of sets and/or logic as described in Sections 4.2 and 4.4. A more formal definition of Boolean algebra is as follows.

Definition 4.25 A *Boolean algebra* $A_{\mathbf{BL}}$ is a 6-tuple, i.e.:

$$A_{\mathbf{BL}} = (V\mathbf{BL}, 0, 1, +, *, \bar{\ }) \qquad (4.36)$$

where $V\mathbf{BL}$ is a set of Boolean variables, 0 and 1 are the Boolean constants, +, *, and $\bar{\ }$ are Boolean operations known as *sum, product,* and *complement*, respectively.

4.3.1.3 Process Algebra

A process algebra is a sequence of state transitions that may be used to denote system behaviors. More rigorous definitions of processes will be given in Sections 4.6.1 and 5.5.1.

Definition 4.26 *Process algebra* is an abstract algebraic system in which the entities of algebraic operation are computational processes.

Process algebra is a kind of dynamic algebra that focuses on computational operations modeled as processes, their algebraic properties, and relations. Process algebra provides a set of formal notations and rules for describing algebraic objects of processes and their algebraic relations [Hoare, 1978/85; Milner, 1980/89]. An extended form of process algebra, RTPA [Wang, 2002a/02b/03c/07a], which deals with the 3-D properties of software behaviours as introduced in Section 3.4.2, will be intensively described in Section 4.6.

4.3.1.4 Concept Algebra

Definition 4.27 *Concept algebra* (CA) is a new mathematical structure for the formal treatment of abstract concepts and their algebraic relations, operations, and associative rules for composing complex concepts and knowledge.

Concept algebra deals with the algebraic relations and associational rules of abstract concepts. The associations of concepts form a foundation to denote complicated relations between concepts in knowledge representation. The associations among concepts can be classified into nine categories, such as *inheritance, extension, tailoring, substitute, composition, decomposition, aggregation, specification*, and *instantiation* [Wang, 2006e]. Further details will be presented in Section 15.3.3.

4.3.1.5 System Algebra

Definition 4.28 *System algebra* is an algebraic system on entities known as systems, which are beyond sets, functions, and processes.

System algebra [Wang, 2006d] is useful in abstract systems modeling, system analysis, and system operations. System algebra will be introduced in Chapter 10 on system science and its applications in software engineering.

4.3.2 FUNCTIONS

Function is an important mathematical concept developed in algebra for denoting complicated relations between abstract objects. Almost all discrete or continuous relations between sets can be described as functions.

4.3.2.1 Notations of Functions

Definition 4.29 A *function f* is a mapping relation → between two sets *X* and *Y* in a generic signature as follows:

$$f: X \rightarrow Y \tag{4.37}$$

where *X* is called the *domain* of the function, and *Y* the *codomain*.

Another form for denoting a function, particularly a continuous function, is given below:

$$y = f(x), \quad x \in X, y \in Y \tag{4.38}$$

Definition 4.29 denotes that for each given *independent variable* $x \in X$, function *f* maps it into a unique *dependent variable* $y \in Y$. Thus, $f: X \rightarrow Y$ is called a *total* function. The function that maps a subset of X into Y is called a *partial* function, denoted by $f: X \nrightarrow Y$.

A function usually results in a transformation of values between variables in either the same or different types. Most mathematical functions are in the former form, i.e.:

$$f: \mathbf{R} \rightarrow \mathbf{R} \tag{4.39}$$

where **R** denotes the type or a set of real numbers.

However, more generally in software engineering, most computational operations may be defined in the latter form. That is, the mappings, in a more general case, are conducted between different types.

Example 4.9 An important *addressing* function π in software engineering, which maps a given identified *id* in type $\mathbf{S}$ (string) into an associated memory address located by a pointer *ptr* in type $\mathbf{H}$ (hexidecimal), can be denoted as follows:

$$\pi\text{: } id\mathbf{S} \rightarrow ptr\mathbf{H} \tag{4.40a}$$

or

$$ptr\mathbf{H} = \pi(id\mathbf{S})\mathbf{H} \tag{4.40b}$$

4.3.2.2 Inverse Functions

Definition 4.30 An *inverse function* f^{-1} is an inverse mapping relation between the codomain *Y* and domain *X* of a given function *f*, i.e.:

$$f^{-1}\text{: } Y \rightarrow X \tag{4.41a}$$

or

$$x = f^{-1}(y), \quad x \in X, y \in Y \tag{4.41b}$$

Example 4.10 The inverse function of *addressing* π as defined in Example 4.9 converses a given memory address $ptr\mathbf{H}$ into an associated logical name $id\mathbf{S}$, known as *memory allocation* π^{-1}, can be denoted as follows:

$$\pi^{-1}\text{: } ptr\mathbf{H} \rightarrow id\mathbf{S} \tag{4.42a}$$

or

$$id\mathbf{S} = \pi^{-1}(ptr\mathbf{H})\mathbf{S} \tag{4.42b}$$

Addressing π and memory allocation π^{-1} are fundamental computing functions widely used in software engineering, which will be modeled in RTPA in Section 4.6 using the notations ⇒ and ⇐, respectively.

4.3.2.3 Composition of Functions

As that of relations, complex functions can be constructed by a composition of simple ones. Reversely, complex functions can also be decomposed into primitive ones.

Definition 4.31 A *composition of two functions* $f: X \rightarrow Y$ and $g: Y \rightarrow Z$, which results in a composed function $c: X \rightarrow Z$, is denoted by $g \circ f$, i.e.:

$$\begin{aligned} c: X \rightarrow Z &\triangleq g(\mathrm{y}) \circ f(\mathrm{x}) \\ &= g(f(\mathrm{x})) \end{aligned} \tag{4.43}$$

The composition operation can be extended to multiple functions as described below.

Definition 4.32 A *composition of multiple functions* among f_1, f_2, … f_n is a multi-layer embedded function f^n, i.e.:

$$\begin{aligned} f^n &= f_n \circ \ldots \circ f_2 \circ f_1 \\ &= f_n (\ldots (f_2 (f_1(\mathrm{x})) \ldots) \end{aligned} \tag{4.44}$$

Example 4.11 If each statement in a list of linear (sequential) process in a program, s_1, s_2, … , s_n, is treated as a function, the process P is a composition of all the sequential statements, i.e.:

$$\begin{aligned} P &= s^n \\ &= s_n \circ \ldots \circ s_2 \circ s_1 \\ &= s_n (\ldots (s_2 (s_1(I)) \ldots) \end{aligned} \tag{4.45}$$

where I denotes a set of concrete data objects as the initial inputs in computing.

It is noteworthy that the compositional description of the process P is only valid for sequential processes. There are more complicated relations between statements in a process such as branch, iteration, interrupt, and parallel. A process in a program that consists of those complicated relations rather than sequential ones will be given in Theorem 4.10 in Section 4.6.1.

4.3.3 ALGEBRAIC OPERATIONS

The third powerful property of algebra is that it is an open system allowing various algebraic operations to be introduced as functions between the objects under operation.

Definition 4.33 An *operation* on a nonempty set A is an abstract function *, i.e.:

$$* : A \times A \rightarrow A \tag{4.46}$$

denoted by *(a, b) or $a * b$, where $a, b \in A$.

The abstract operation * can also be perceived as a binary relation, or more generally, an n-nary relation, $n \geq 1$, between n variables in A.

Example 4.12 Let $\mathbf{R}$ be the set of real numbers, the arithmetical operations * ={+, -, •, ÷} on $\mathbf{R}$ can be defined as follows:

$$* : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \tag{4.47a}$$

and

$$+ : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \tag{4.47b}$$

$$- : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \tag{4.47c}$$

$$\bullet : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \tag{4.47d}$$

$$\div : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \tag{4.47e}$$

It is noteworthy that - and ÷ are not a valid operation on set $\mathbf{N}$ of natural numbers, because the results of these two operations may be out of the domain of $\mathbf{N}$, such as negative or fractional numbers.

The properties of operations can be associative and/or commutative, dependent on whether the following conditions are met respectively.

Definition 4.34 An operation * on a set A is *associative iff*:

$$\forall a_1, a_2, a_3 \in A \Rightarrow (a_1 * a_2) * a_3 = a_1 * (a_2 * a_3) \tag{4.48}$$

Definition 4.35 An operation * on a set A is *commutative* if:

$$\forall a_1, a_2 \in A \Rightarrow a_1 * a_2 = a_2 * a_1 \tag{4.49}$$

Example 4.13 Examining the arithmetic operations defined on $\mathbf{R}$ as shown in Example 4.12, * ={+, -, •, ÷}, it can be found that both + and • are associative and commutative, but - and ÷ are not. Further, it can be seen that • over + are *distributive*, i.e.:

$$\forall a, b, c \in \mathbf{R} \Rightarrow a \bullet (b + c) = ab + ac \tag{4.50}$$

4.4 Mathematical Logic

Mathematical logic formalizes the structures and procedures used in deductive manipulation of objects and relations. Symbolic logic is developed for a wide range of application in *argument*, *reasoning*, *deduction*, *induction*, and *proof*. George Boole (1815-1864) developed the mathematical theories of logic and probabilities, particularly Boolean algebra, which he considered as the laws of thought. Russell and Godel advanced the art of mathematical logic [van Heijenoort, 1997].

This section describes prepositional logic, predicate logic, and their applications in software engineering. Further studies on higher order logic structures may be referred to [Hurley, 1997; Tomassi, 1999].

4.4.1 PROPOSITIONAL LOGIC

Propositional logic deals with a given logical statement as a whole known as a proposition. Compound statements can be built based on simple statements through the use of logical operators. Once an argument is symbolically represented by propositional logic, mere inspection will often determine whether it is valid or invalid.

4.4.1.1 Propositions

Before discussing the concept of a proposition, three conditional logical relations known as yield, implication, and equivalence are introduced.

Definition 4.36 A *yield relation* γ is a conditional logical relation $\vdash$ that denotes a causal relationship between two Boolean objects $o_1\mathbf{BL}$ (the cause) and $o_2\mathbf{BL}$ (the consequence), i.e.:

$$\begin{aligned}\gamma\mathbf{BL} &\triangleq o_1\mathbf{BL} \vdash o_2\mathbf{BL} \\ &= o_2\mathbf{BL} = \mathbf{T} \textit{ iff } o_1\mathbf{BL} = \mathbf{T}\end{aligned} \tag{4.51}$$

where the attached bold symbol such as **BL** is called the type-suffix to denote the Boolean type of a logical variable or statement, and similarly **T** or **F** denote the Boolean values true or false, respectively.

A yield operation is used to express a causality between the cause and consequence where the consequence is true *iff* the cause is true. In other words, there is no definition when the cause is not true.

Definition 4.37 An *implication* ι is a conditional logical relation $\Rightarrow$ that denotes *iff* $o_1\mathbf{BL} = \mathbf{T}$ then $o_2\mathbf{BL} = \mathbf{T}$ is determinable, otherwise $o_2\mathbf{BL}$ is indeterminable, i.e.:

$$\begin{aligned}\iota\,\mathbf{BL} &\triangleq o_1\mathbf{BL} \Rightarrow o_2\mathbf{BL} \\ &= (o_1\mathbf{BL} = \mathbf{T} \vdash o_2\mathbf{BL} = \mathbf{T}) \vee (o_1\mathbf{BL} = \mathbf{F} \vdash (o_2\mathbf{BL} = \mathbf{T} \vee o_2\mathbf{BL} = \mathbf{F}))\end{aligned} \tag{4.52}$$

It is noteworthy that implication is often overloaded to denote the *yield* relation $\vdash$ between two propositions in logical inferences when it causes no confusion.

Definition 4.38 An *equivalence* ε is a conditional logical relation $\Leftrightarrow$ that denotes both if $o_1\mathbf{BL} = \mathbf{T}$ then $o_2\mathbf{BL} = \mathbf{T}$ and if $o_1\mathbf{BL} = \mathbf{F}$ then $o_2\mathbf{BL} = \mathbf{F}$ are determinable, and vice versa, i.e.:

$$\begin{aligned}\varepsilon\mathbf{BL} &\triangleq o_1\mathbf{BL} \Leftrightarrow o_2\mathbf{BL}\\ &= (o_1\mathbf{BL} = \mathbf{T} \vdash o_2\mathbf{BL} = \mathbf{T}) \vee (o_1\mathbf{BL} = \mathbf{F} \vdash o_2\mathbf{BL} = \mathbf{F})\\ &\wedge (o_2\mathbf{BL} = \mathbf{T} \vdash o_1\mathbf{BL} = \mathbf{T}) \vee (o_2\mathbf{BL} = \mathbf{F} \vdash o_1\mathbf{BL} = \mathbf{F})\end{aligned} \tag{4.53}$$

For contrasting the differences of the three conditional operations, readers are suggested to refer to Table 4.4. Based on the above conditional operations, logical propositions can be formally described as follows.

Definition 4.39 A *proposition* ρ is a declarative statement that expresses a Boolean concept ($\mathbf{BL}$) or a '*to be'* relation $\vDash$ between two or more logical objects $o_i\mathbf{BL}$, $1 \le i \le n$, which can be evaluated as either true ($\mathbf{T}$) or false ($\mathbf{F}$), i.e.:

$$\rho\mathbf{BL} \triangleq o_1\mathbf{BL} \vDash o_2\mathbf{BL} \tag{4.54}$$

where the *to be* relations $\vDash \in$ {*is*, *are,* $=$, $\equiv$, $\vdash$, $\Rightarrow$, $\Leftrightarrow$}, and $\equiv$ is treated as equivalent to $\Leftrightarrow$.

Example 4.14 The following statements are propositions:

$$\rho_1\mathbf{BL} \triangleq (1 \equiv 1)\mathbf{BL} = \mathbf{T}$$

$$\rho_2\mathbf{BL} \triangleq (\text{North is not the opposite of south})\mathbf{BL} = \mathbf{F}$$

Propositions defined according to Definition 4.39 are called *primitive* propositions, which cannot be broken down into more simpler ones.

Definition 4.40 *Propositional logic* is a branch of symbolic logic that deals with propositions as a whole and the Boolean logical relations between them.

4.4.1.2 Propositional Logic Operations

The basic operations in propositional logic are *conjunction* ($\wedge$), *disjunction* ($\vee$)*, implication* ($\Rightarrow$)*, equivalence* ($\Leftrightarrow$), and *negation* ($\neg$). The operations of propositional logic may also be classified into *connective* operations and *conditional* operations as shown in Table 4.4 defined using truth tables.

The truth table is useful for defining and analyzing composite propositions. It is also useful for evaluating the equivalence between two composite propositions.

Table 4.4
Truth Tables of Connective and Conditional Logical Operations

Propositions		**Connective Operations**			**Conditional Operations**		
p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \vdash q$	$p \Rightarrow q$	$p \Leftrightarrow q$
T	**T**	**F**	**T**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**	**F**	**F**
F	**T**	**T**	**F**	**F**	**F**	**T**	**F**
F	**F**	**T**	**F**	**F**	**F**	**T**	**T**

Definition 4.41 A *composite proposition P* is a composition of multiple *primitive* propositions by *connectives* $\mho$, such as conjunction $\wedge$, disjunction $\vee$, and negation $\neg$, i.e.:

$$P\mathbf{BL} \triangleq \rho_1\mathbf{BL} \mho \rho_2\mathbf{BL} \tag{4.55}$$

where $\mho \in \{\wedge, \vee, \neg\}$.

Example 4.15 Using the primitive propositions defined in Example 4.14, the following composite propositions can be derived:

$$\begin{aligned} P\mathbf{BL} &\triangleq \rho_1\mathbf{BL} \wedge \rho_2\mathbf{BL} \\ &= (1 \equiv 1)\mathbf{BL} \wedge (\text{North is not the opposite of south})\mathbf{BL} \\ &= \mathbf{T} \wedge \mathbf{F} \\ &= \mathbf{F} \end{aligned}$$

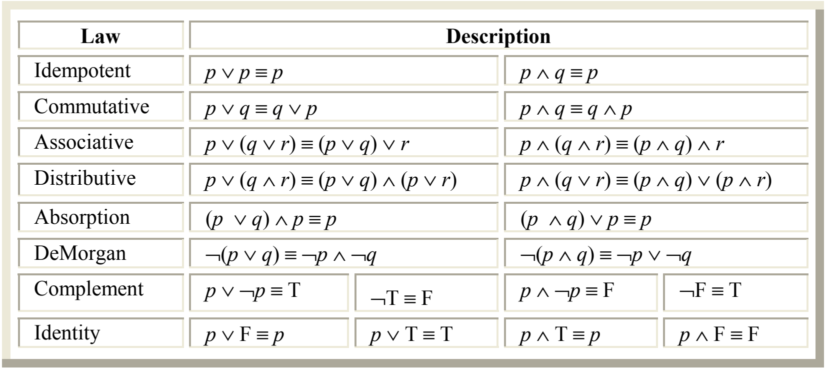
$$\begin{aligned} P'\mathbf{BL} &\triangleq \rho_1\mathbf{BL} \vee \rho_2\mathbf{BL} \\ &= (1 \equiv 1)\mathbf{BL} \vee (\text{North is not the opposite of south})\mathbf{BL} \\ &= \mathbf{T} \vee \mathbf{F} \\ &= \mathbf{T} \end{aligned}$$

4.4.1.3 Laws of Propositional Algebra and Logical Inferences

Propositional algebra and the rules of logical inference obey the algebraic laws as summarized in Table 4.5. These laws play important roles to compose complex propositions and to facilitate logical reasoning. Table

4.5 shows that the propositional operations of conjunction, disjunction, and negation are *symmetric*. It is interesting to observe the phenomena of duality as well as similarity between the logical and set laws as shown in Tables 4.5 and 4.2, respectively.

Table 4.5
Laws of Propositional Algebra



Law	Description			
Idempotent	$p \vee p \equiv p$		$p \wedge q \equiv p$	
Commutative	$p \vee q \equiv q \vee p$		$p \wedge q \equiv q \wedge p$	
Associative	$p \vee (q \vee r) \equiv (p \vee q) \vee r$		$p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$	
Distributive	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$		$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	
Absorption	$(p \vee q) \wedge p \equiv p$		$(p \wedge q) \vee p \equiv p$	
DeMorgan	$\neg(p \vee q) \equiv \neg p \wedge \neg q$		$\neg(p \wedge q) \equiv \neg p \vee \neg q$	
Complement	$p \vee \neg p \equiv \mathrm{T}$	$\neg\mathrm{T} \equiv \mathrm{F}$	$p \wedge \neg p \equiv \mathrm{F}$	$\neg\mathrm{F} \equiv \mathrm{T}$
Identity	$p \vee \mathrm{F} \equiv p$	$p \vee \mathrm{T} \equiv \mathrm{T}$	$p \wedge \mathrm{T} \equiv p$	$p \wedge \mathrm{F} \equiv \mathrm{F}$

Another law known as the *involution law* of propositions is:

$$\neg\neg p \equiv p \tag{4.56}$$

The fundamental unit of propositional argument is the whole statement. Therefore, propositional logic lacks the capability to look into the low level structures of statements that it treats as a black box. When more analytical power is needed to deal with the contents of a statement known as terms, and when the universal or existential quantifications need to be deduced, the extension of prepositional logic by predicate logic is required as described in the following subsection.

4.4.2 PREDICATE LOGIC

Predicate logic introduces variables, properties, and quantifiers on the basis of prepositional logic. Predicate logic combines the five operators of propositional logic with symbols for predicates and quantifiers. Based on this, a more powerful symbolic system is formed to represent the content of logical arguments and natural language statements.

Definition 4.42 A *predicate* p is a declarative assertion that affirms or denies an object o**BL** or a relation R as true or existing, i.e.:

$$p(o)\mathbf{BL} \triangleq (o \vDash p)\mathbf{BL} \tag{4.57a}$$

$$p(o_1, o_2)\mathbf{BL} \triangleq R(o_1,o_2)\mathbf{BL} = (o_1 R o_2)\mathbf{BL} \tag{4.57b}$$

where the *to be* relations $\vDash \in \{is, are, =, \equiv, \vdash, \Rightarrow, \Leftrightarrow\}$, $o_1\mathbf{BL}$ and $o_2\mathbf{BL}$ are two arbitrary logical objects.

Example 4.16 The following statements are predicates:

$$p_1(x)\mathbf{BL} \triangleq (\mathrm{x} \vDash p_1)\mathbf{BL} = \mathbf{T}$$

$$p_2(2,1)\mathbf{BL} \triangleq >(2,1)\mathbf{BL} = (2 > 1)\mathbf{BL} = \mathbf{T}$$

Definition 4.43 *Predicate logic* is a branch of symbolic logic that deals with propositions containing predicates, variables, and quantifiers.

Once an argument is translated into the symbols of predicate logic, natural deduction is enabled to derive a sound and valid conclusion.

4.4.2.1 Taxonomy of Predicates

There are three types of predicates in predicate logic known as singular, universal, and particular predicates or statements.

Definition 4.44 A *singular predicate* is a specific statement that asserts a specific object *o* satisfies a given predicate *P*, i.e.:

$$P(o)\mathbf{BL} \tag{4.58}$$

where *o* is called an *individual* constant.

Definition 4.45 A *universal predicate* is a general statement that asserts every element *x* in a set *S* satisfies a given predicate *P*, i.e.:

$$\forall \mathrm{x} \in S \vdash P(x)\mathbf{BL} \tag{4.59}$$

where ∀ is the *universal quantifier*, and *x* is called an *individual variable*.

Definition 4.46 A *particular predicate* is a specific statement that asserts at least one element *x* in a set *S* satisfies a given predicate *P*, i.e.:

$$\exists \mathrm{x} \in S \vdash P(x)\mathbf{BL} \tag{4.60}$$

where ∃ is the *existential quantifier*, and x is an *individual* variable.

4.4.2.2 Concept Construction with Predicate Logic

In predicate logic, complex logical concepts and arguments can be hierarchically constructed by predicates, functions, terms, and formulae, from the bottom up.

4.4.2.2.1 Logical Functions

Definition 4.47 A *function* $\mathcal{F}$ of predicate logic is a '*to be*' relation ⊨ between a predicate F and a given logical variable x, i.e.:

$$\mathcal{F}(x)\mathbf{BL} \triangleq (\mathrm{x} \vDash F)\mathbf{BL} \tag{4.61}$$

where x is a *free* variable that is not bounded by a quantifier.

4.4.2.2.2 Logical Terms

Definition 4.48 A *term* $\mathcal{T}$ of predicate logic is defined recursively as follows:

(a) Every logical *variable* and every *constant* is a term.

(b) If $t_1, t_2, ..., t_n$ are terms and f is a function that takes n arguments, then $f(t_1, t_2, ..., t_n)$ is a term.

(c) Every term is obtained in this manner.

4.4.2.2.3 Logical Formulae

Definition 4.49 A *formula* $\mathcal{L}$ of predicate logic is defined recursively as follows:

(a) If $t_1, t_2, ..., t_n$ are terms and P is a predicate that takes n arguments, then $P(t_1, t_2, ..., t_n)$ is a formula called an *atomic* formula.

(b) If α and β are formulae, so are $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, $(\alpha \Leftrightarrow \beta)$.

(c) If x is a variable and α is a formula, then $(\forall x\ \alpha)$, $(\exists x\ \alpha)$ are formulae.

(d) Every formula is obtained in this manner.

Example 4.17 Given three natural language statements and corresponding definitions of predicates as follows:

$$(1)\ \frac{\text{Colored flowers}}{C(x)} \text{ are } \frac{\text{always scented}}{S(x)}.$$

$$(2)\ \frac{\text{I dislike flowers}}{D(x)} \text{ that are not } \frac{\text{grown in the open air}}{\mathrm{G}(x)}.$$

$$(3)\ \text{No flowers } \frac{\text{grown in the open air}}{\mathrm{G}(x)} \text{ are } \frac{\text{colorless}}{\neg C(x)}.$$

The equivalent formulae in predicate logic can be derived as follows:

$$(1)\ \mathcal{L}_1 \triangleq \forall x\ C(x)\ \vdash S(x)$$

$$(2)\ \mathcal{L}_2 \triangleq \forall x\ (\neg G(x) \vdash D(x))$$

$$(3)\ \mathcal{L}_3 \triangleq \neg\ (\exists x\ (G(x) \wedge \neg C(x)))$$

4.4.2.3 Inferences in Predicate Logic

Once a natural language statement is represented by a symbolic formula in predicate logic, a conclusion of argument may be systematically and rigorously derived via logical deduction on the basis of known inference rules as developed in Section 3.3.

The basic propositional inferences are summarized in Table 4.5. Four additional rules in predicate logic for dealing with the quantifiers in predicate formulae can be derived. They are the rules of universal instantiation, universal generalization, existential instantiation, and existential generalization.

For individual variables $x, y \in S$, and individual constant $a, b \in S$, given $\mathcal{L}(a)$ be a statement, and $\mathcal{L}(x)$ or $\mathcal{L}(y)$ be a formula or function, the following definitions present the rules of inference on predicate formulae.

Definition 4.50 The *rule of universal instantiation* states that a specific instance statement can be deduced from a general predicate formula or function, i.e.:

$$\forall x, \mathcal{L}(x) \vdash \mathcal{L}(a) \tag{4.62}$$

Definition 4.51 The *rule of universal generalization* states that a general predicate formula or function can be deduced from a specific instance formula or function, i.e.:

$$\mathcal{L}(a) \vdash \forall x, \mathcal{L}(x) \tag{4.63}$$

Definition 4.52 The *rule of existential instantiation* states that a specific instance statement can be deduced from at least one predicate formula or function, i.e.:

$$\exists x, \mathcal{L}(x) \vdash \mathcal{L}(a) \tag{4.64}$$

Definition 4.53 The *rule of existential generalization* states that a generic predicate formula or function can be deduced from a specific instance formula, function, or statement, i.e.:

$$\mathcal{L}(a) \vdash \exists x, \mathcal{L}(x) \tag{4.65}$$

More complicated logical argument methodologies and formal inference processes may be referred to Section 3.3. It is noteworthy that first order predicate logic is timid. That is, if something is uncertain, it makes no assumptions. In order to deal with nontraditional problems in system description and uncertainty reasoning, a number of nonconventional logical forms have been proposed, such as multiple-valued logic [Dunn and Epstein, 1977], temporal logic [Pnueli, 1997; Emerson, 1990], and fuzzy logic [Zadeh, 1965/73/82].

However, it is noteworthy that mathematical logic is good at addressing the *to be* |= reasoning. More dynamic and diverse problems in the category of *to do* in system modeling require new forms of mathematical means known as denotational mathematics, which will be described in the following sections.

4.5 Denotational Mathematics for Software Engineering

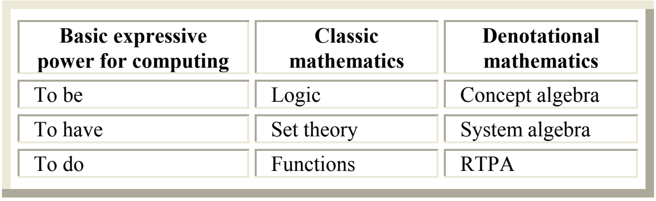
Denotational mathematics is a set of contemporary mathematical structures for dealing with the unique mathematical entities, abstract objects, relations, and formal manipulations in abstract system modeling, which encompasses concept algebra, system algebra, and RTPA. Concept algebra has been introduced in Section 4.3.1 and details are provided in Section 15.3.3.1 and in [Wang, 2006e]. System algebra will be intensively discussed in Section 10.4. This section focuses on RTPA and the big-R notation.

4.5.1 FUNDAMENTAL ELEMENTS IN MODELING SOFTWARE SYSTEMS

It is observed that, although there are various ways to express oneself, human and system behaviors can be classified into three categories: to *be*, to *have*, and to *do* [Wang, 2006a]. All mathematical means and forms, in general, are an abstract description of these three categories of human and system behaviors and common rules of them. Taking this view, mathematical logic may be perceived as the abstract means for describing "to be," set theory for describing "to have," and functions for describing "to do."

Three forms of denotational mathematics [Wang, 2002a/06d/06e/06j/07a] such as concept algebra, system algebra, and RTPA are created to enable rigorous treatment of software and knowledge representation and manipulation in a formal and coherent framework, which extend the expressive capability for abstract objects under study from basic mathematical entities of numbers and sets to higher levels, i.e., concepts, systems, and behavioral processes. Table 4.6 contrasts the usages of denotational mathematics and classic discrete mathematics.

Table 4.6
Basic Expressive Power for Denotational Mathematics



Basic expressive power for computing	Classic mathematics	Denotational mathematics
To be	Logic	Concept algebra
To have	Set theory	System algebra
To do	Functions	RTPA

Table 4.6 demonstrates a fundamental view toward the natural and machine intelligence description in general, and software system in particular. This also indicates that only a logic-based approach, which developed in philosophy and pure mathematics, is not adequate to be taken as the sole mathematical foundation for software engineering.

It is recognized in Theorem 3.10 that the behavioral space of any system or human action is three dimensional encompassing *action*, *time*, and *space*. Correspondingly, there are three fundamental categories of *computational behaviors* in a software system: a) computational operations for variable manipulation, b) timing operations for event manipulation, and c) space operations for memory manipulation. Therefore, the behavior of a software system can, in general, be viewed as a set of 3-D *processes* comprising computational operations, time, and memory.

It may be argumentative that some transaction processing systems in computing are 2-D, i.e., those systems' behavioral space may only encompass operational logic and static memory allocation. Then, the 2-D systems can be treated as special cases of the generic 3-D systems. It is mathematically intuitive that any method or technologies that apply to the 3-D problems in software engineering are applicable to the 2-D problems. However, a pure 2-D technology is inadequate and unsafe to be extended to a 3-D problem.

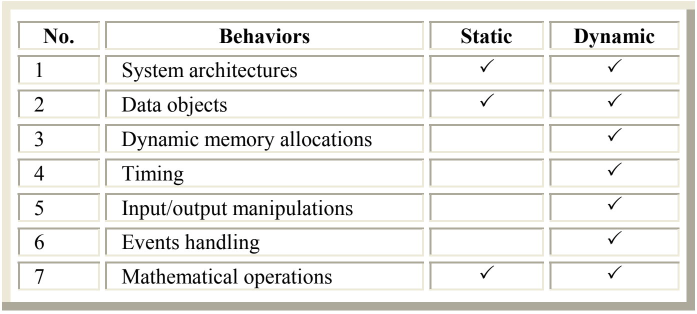
Therefore, the requirement for a denotational mathematics that is suitable for the 3-D problems is theoretically and practically fundamental in software engineering. RTPA has been developed as a coherent notation system and a formal engineering method for addressing the 3-D problems in software system specification, refinement, and implementation, particularly for real-time and embedded systems.

Definition 4.54 A *behavior* of a software system is its computing operations *OPs* and observable outcomes and effects that affect or change the states of a system in the environment modeled by all variables and input/output events, as well as related memory structures *M* over time *T*, i.e.:

$$\textit{Software behavior} \triangleq f\,(OP, T, M) \qquad (4.66)$$

Behaviors of software systems can be classified as static and dynamic ones as shown in Table 4.7.

Table 4.7
Characteristics of Software System Behaviors



No.	Behaviors	Static	Dynamic
1	System architectures	✓	✓
2	Data objects	✓	✓
3	Dynamic memory allocations		✓
4	Timing		✓
5	Input/output manipulations		✓
6	Events handling		✓
7	Mathematical operations	✓	✓

Definition 4.55 A *static behavior* of software systems is a process that can be determined at design or compile time.

Definition 4.56 A *dynamic behavior* of software systems is a process specified by given timing requirements that may only be determined at run-time.

It is noteworthy in Table 4.7 that most software behaviors are dynamic or characterized both dynamic and static. Mathematical logic is found capable to deal with the 'to be' type static behaviors; the rest of the dynamic instructive behaviors in computing have to be manipulated by process algebras, particularly RTPA.

4.5.2 THE NEED FOR DENOTATIONAL MATHEMATICS IN SOFTWARE ENGINEERING

Christopher Strachey (1965), the founder of the Programming Research Group (PRG) in the Computing Laboratory at Oxford University, wrote: "It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing."

4.5.2.1 Problems Yet to be Solved

In software engineering we are still facing the same problems as those the community dealt with 40 years ago. Dijkstra supposed this was because of the spaghetti usage of "goto' architectures and *'the non-constructed approaches'* in programming [Dijkstra, 1968a]. Brooks explained that there is no silver bullet in the foreseeable future [Brooks, 1987].

This special phenomenon indicates that: (a) The *current empirical approach* centered by languages toward programming is *insufficiently practical*, and (b) The *current theories* and *mathematical means* are *inadequate*. Profound problems yet to be solved and the core technology that we need in software engineering are to support the *expression* of *system architectures, behaviors,* and their implementation. Therefore, the major reason for all the difficulties in software engineering is that we did not get our theories, mathematical means, and tools ready before we can effectively and generically deal with the complicated long-chain sequences of instructive behaviors in software engineering.

4.5.2.2 New Problems Require New Forms of Mathematics

The history of sciences and engineering shows that new problems require new forms of mathematics. Many branches of mathematics were emerged in engineering sciences in order to meet their abstract, rigor, and expressive needs. Software science and engineering is a transdisciplinary enquiry that encompasses a wide range of contemporary science and engineering disciplines related to information and knowledge processing, which cannot be described by conventional analytic mathematics. Therefore, new forms of mathematics are sought, collectively known as the *denotational mathematics*, to deal with the unique mathematical entities and abstract objects emerged in the field of software engineering, such as information, concepts, knowledge, processes, behaviors, systems, complex relations, and distributed objects.

Definition 4.57 *Denotational mathematics* is a category of mathematical structures that formalizes rigorous expressions and inferences of system architectures and behaviors with data, concepts, and dynamic processes.

It is observed that all existing mathematics, continuous or discrete, are mainly *analytic*, seeking unknown variables from known factors according to certain functions. Conventional science and engineering disciplines have been mainly using analytic methodologies and mathematics in theory development and problem solving. However, in software engineering, the need is to formally describe and specify software systems, particularly its architecture, static behaviors, and dynamic behaviors in terms of operational logic, timing, and memory manipulation. Because programming languages lack the required *expressive power* to deal with all the dynamic behaviors in the 3-D behavioral space, denotational mathematics that can describe software architectures and behaviors rigorously, precisely, and expressively are sought [Wang, 2002a/06d/06e/06j/07a].

According to Theorems 1.3, 1.4, and 4.1, the utility of denotational mathematics for software engineering is the means and rules to rigorously and explicitly express design notions and conceptual models on system architectures and behaviors at the highest level of abstraction, in the largest scope of systems, and with the complicated long-chain sequences of 3-D computational behaviors.

RTPA is a form of denotational mathematics that provides an expressive, coherent notation system, and systematical refinement methodology for addressing the abstract architectures and 3-D behaviors of software systems. RTPA will be introduced in Section 4.6. On the basis of denotational mathematics, an expressive software engineering notation system, a formal software engineering modelling methodology, and a rigorous specification of software architecture and behaviours may be developed systematically.

4.5.3 THE BIG-R NOTATION

The most generic and fundamental operations in system and human behavioral modeling are iterations and recursions. Because a variety of iterative constructs are provided in different programming languages, the notation for repetitive, cyclic, recursive behaviors and architectures in computing need to be unified.

The *big-R notation* is introduced to deal with this fundamental requirement in computing and software engineering [Wang, 2006f], which is proposed first in RTPA [Wang, 2002a]. In order to develop a general mathematical model for unifying the syntaxes and semantics of iterations and recursions, their inductive nature is analyzed below.

Definition 4.58 An *iteration* of a process *P* can be defined as a series of *n+1* repetitions, R_i, $1 \leq i \leq$ n+1, of *P* by mathematical induction, i.e.:

$$\begin{aligned} &R_0 = \otimes, \\ &R_1 = P \to R_0, \\ &\ldots \\ &R_{n+1} = P \to R_n,\ \ n \geq 0 \qquad (4.67) \end{aligned}$$

where ⊗ denotes skip, or doing nothing but exit.

Based on Definitions 4.58, the big-R notation can be introduced below.

Definition 4.59 The *big-R notation,* R, is a generic mathematical operator that is used to denote: (a) a finite set of *repetitive* behaviors, or (b) a finite set of recurring architectural constructs in computing, in the following forms, respectively:

$$\text{(a)}\ \underset{\text{exp}\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{\mathrm{R}}}\ P \qquad (4.68)$$

$$\text{(b)}\ \underset{\text{i}\mathbf{N}=1}{\overset{n}{\mathrm{R}}}\ P(i\mathbf{N}) \qquad (4.69)$$

where **BL** and **N** are the type suffixes of Boolean and natural numbers, respectively; **T** and **F** are the Boolean constants true and false, respectively.

Further description of the type system and a summary of all type suffixes of RTPA will be presented in Section 4.6.3. Formal type theory will be provided in Section 5.3 on data object modeling and manipulation.

The mechanism of the big-R notation can be in analogy with the mathematical notations $\sum$ and $\prod$, or programming notations of while-loop and for-loop as shown in the following examples.

Example 4.18 The *big*-Σ notation $\sum\limits_{i=1}^{n} x_i$ is a widely used calculus for denoting repetitive additions. Assuming that the operation of addition is represented by *sum*(x), the mechanism of big-Σ can be expressed more generally by the big-R notation, i.e.:

$$\sum_{i=1}^{n} x_i = \mathop{\mathbf{R}}\limits_{\mathrm{i}=1}^{n} sum(x_i) \tag{4.70}$$

According to Definition 4.59, the big-R notation can be used to denote not only repetitive operational behaviors in computing, but also recurring constructs of architectures and data objects as shown below.

Example 4.19 The architecture of a two-dimensional array with $n \times m$ integer elements, A_{nm}, can be denoted by the big-R notation as follows:

$$A_{nm} = \mathop{\mathbf{R}}\limits_{\mathrm{i}=0}^{n\text{-}1} \mathop{\mathbf{R}}\limits_{\mathrm{j}=0}^{m\text{-}1} A[i, j]\mathbf{N} \tag{4.71}$$

Because the big-R notation provides a powerful and expressive means for denoting iterative and recursive behaviors and architectures of systems or human beings, it is a universal mathematical means for system modeling in terms of repetitive behaviors and structures or architectures, respectively [Wang, 2006f]. From this point of view, $\sum$ and $\prod$ are special cases of the big-R for repetitively doing additions and multiplications, respectively.

Definition 4.60 *An infinitive iteration* can be denoted by the big-R notation as:

$$\mathbf{R}\, P \triangleq \gamma \bullet P \curvearrowright \gamma \tag{4.72}$$

where γ is a label that denotes the rewinding point of a loop, and $\curvearrowright$ denotes a jump.

The infinitive iteration may be used to formally describe an everlasting behavior of systems.

Example 4.20 A simple everlasting clock, *CLOCK,* which does nothing but tick as C.A.R. Hoare proposed [Hoare, 1985], i.e.:

$$CLOCK \triangleq tick \to tick \to tick \to \ldots \tag{4.73}$$

can be efficiently denoted by the big-R notation as simply as follows:

$$CLOCK \triangleq \mathrm{R}\ tick \tag{4.74}$$

A more generic and useful iterative construct is the conditional iteration.

Definition 4.61 *A conditional iteration* can be denoted by the big-R notation as:

$$\begin{array}{l} \mathop{\mathrm{R}}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}} P \triangleq \gamma \bullet (\ \text{�}\ exp\mathbf{BL} = \mathbf{T} \\ \qquad\qquad\quad \to P \\ \qquad\qquad\quad \curvearrowright \gamma \\ \qquad\qquad\ \mid \text{�}\ \sim \\ \qquad\qquad\quad \to \otimes \\ \qquad\qquad\ ) \end{array} \tag{4.75}$$

where ⊗ denotes a skip.

The conditional iteration is frequently used to formally describe repetitive behaviors on given conditions. Eq. 4.75 expresses that the iterative execution of *P* will go on as long as the evaluation � of the conditional expression is true ($exp\mathbf{BL} = \mathbf{T}$), until $exp\mathbf{BL} = \mathbf{F}$ abbreviated by '~'.

The big-R notation captures and models a fundamental and widely applied mathematical concept in computing and human behavior description. More applications of the big-R notation in modeling fundamental computing behaviors and architectures will be provided in Section 5.4.2, which demonstrate that a convenient mathematical notation may dramatically reduce the difficulty and complexity in expressing a frequently used and highly recurring concept and notion in computing. The big-R notation has been adapted and implemented in RTPA and its support tools.

4.6 Real-Time Process Algebra (RTPA)

Software engineering is a unique discipline in which the objects of their studies require new forms of mathematics known as denotational mathematics in the treatment, modeling, description, specification, development, implementation, and maintenance of software systems. RTPA is developed as a coherent notation system and a formal engineering methodology for addressing the 3-D problem in software system specification, refinement, and implementation for both real-time and nonreal-time systems [Wang, 2002a/02b/03c/06a/07a].

This section presents the process metaphor of software systems and the structure of RTPA. The type system, process notations, process relations, and process composing rules of RTPA are described. The system specification and refinement methodology of RTPA and case studies on real-world problems that demonstrate the descriptive power of RTAP as a powerful software engineering notation system will be provided in the following sections. More rigorous treatment of RTPA type rules and formal semantics will be explored in Chapters 5 and 6, respectively.

4.6.1 THE PROCESS METAPHOR OF SOFTWARE SYSTEMS

An important finding in formal methods is that a software system can be perceived and described as the *composition* of a set of *processes*. The *process metaphor* of software systems has evolved from concurrent processes to real-time process systems in the area of operating system research and formal methods [Hoare, 1978/85; Milner, 1980/89].

Definition 4.62 *Formal methods* are mathematics-based techniques for software system design, description, specification, and modeling in software engineering.

Formal methods are developed to deal with human errors resulted by empirical technologies in software engineering. It has then been found that formal methods are also useful in training and in the exploration of theoretical foundations of software engineering. If software engineers were trained rigorously and experienced some mental challenges at a higher abstract level, it would help them to develop correct, reliable, and high quality software. Current formal methods can be classified into three categories known as *logic*-based (e.g., Z, Object-Z), *algebra*-based (e.g., CSP, RTPA), and *diagram*-based (e.g., SDL, SMC, and UML) [Wang, 2002h].

Conventional formal methods were based on logic and set theories [Spivey, 1988/92; Woodcock and Davies, 1996; Derrick and Boiten, 2001], which were perceived to be suitable for describing static behaviors of software systems. For describing system dynamic behaviors, a variety of algebra-based technologies were proposed since late 1970s [Hoare, 1978/85; Milner, 1980/89; Baeten and Bergstra, 1991; Gerber et al., 1992; Klusener, 1992; Cerone, 2000; Dierks, 2000; Fecher, 2001].

4.6.1.1 Process Algebra

Algebra is a form of mathematics that simplifies difficult problems by using symbols to represent variables, calculus, and their relations. Algebra enables complicated problems to be expressed and investigated in a formal and rigorous process.

Process algebra is a major branch of formal methods that provides an algebraic treatment for software systems as a set of interacting processes. According to Definition 4.26, a process algebra is an abstract algebraic system in which the entities of algebraic operation are computational processes. *Process algebra* defines a set of formal notations and rules for describing algebraic manipulations of software processes.

Hoare [Hoare, 1978/85], Milner [Milner, 1980/89], and others [Corsetti et al., 1991; Nicollin and Sifakis, 1991; Jeffrey, 1992; Vereijken, 1995] developed algebraic ways to represent communicating and concurrent systems known as process algebra. A typical process algebra is known as Communicating Sequential Processes (CSP) developed by C.A.R. Hoare [Hoare, 1985; Brookes et al., 1984]. CSP provides a notation system for the specification of process systems and proof of the implemented processes satisfies their specifications. In CSP, a system is modeled as a set of processes, composed sequentially or in parallel. Each process is described in terms of all of its possible behaviors. Processes may communicate by exchanging data via abstract channels. CSP uses 14 notations to denote processes of software systems as shown in Fig. 4.1.

An important finding in CSP and related work is that a software system may be modeled by a set of interacting (communicating) processes. A process in CSP is perceived as follows.

Definition 4.63 A *process* is an abstract model of a unit of meaningful software behavior that represents a transition procedure of the system from one state to another by changing values of its inputs $\{I\}$, outputs $\{O\}$, and/or internal variables $\{V\}$.

Process	Notation
(1) Sequential	$P ; Q$
(2) Event	$a \rightarrow P$
(3) Branch	$P \lhd c \rhd Q$
(4) Repeat	$* P$
(6) Parallel	$P \parallel Q$
(6) Deterministic choice	P □ Q
(7) Non-deterministic choice	$P \sqcap Q$
(8) Pipeline	P » Q
(9) Interleave	$P \vert\vert\vert Q$
(10) Interrupt	P ^ Q
(11) Assignment	$x := v$
(12) Channel input	$c ? v$
(13) Channel output	$c ! v$
(14) System termination	STOP

Figure 4.1 The CSP notations

It is noteworthy that the concept of process in CSP is hybrid. That is, it does not distinguish the differences between the fundamental meta processes and relational process operations. The CSP notations model a major part of elementary software behaviors that may be used in system specification and description. However, it lacks many useful processes that are perceived essential in system modeling, such as addressing, memory manipulation, timing, and system dispatch. CSP models all input and output (I/O) as abstract channel operations that are not expressive enough to denote complex system interactions, particularly for those of real-time systems.

Wang found that the existing work on process algebra and their timed variations [Reed and Roscoe, 1986; Boucher and Gerth, 1987; Schneider, 1991] can be extended to a new form of expressive mathematics: *Real-time process algebra* [Wang, 2002a/02b/03c/07a]. Real-time process algebra can be used to formally and precisely describe and specify architectures and behaviors of software systems on the basis of algebraic process notations and rules.

4.6.1.2 Real-Time Process Algebra (RTPA)

As indicated in Theorem 3.10, a generic computing problem is a 3-D problem that requires a formal method addressing the requirements in all dimensions, particularly the time dimension. The key metaphor in system modeling, specification, and description is that a software system can be perceived as the *composition* of a set of interacting *processes*, which are constructed on the basis of algebraic operations. A formal description of a process is given below.

Definition 4.64 A *process P* is a composed component of n meta statements s_i and s_j, $1 \leq i < n, j = i + 1,$ according to certain composing relations r_{ij}, i.e.:

$$P = (...(((s_1)\; r_{12}\; s_2)\; r_{23}\; s_3)\; ...\; r_{n-1,n}\; s_n) \tag{4.76}$$

where $r_{ij} \in \mathfrak{R}$, and $\mathfrak{R}$ is a set of algebraic process relations that will be described in Section 4.6.5.

Contrasting Definitions 4.63 and 4.64, readers may find the differences between conventional and mathematical concepts of processes. On the basis of Definition 4.64, the cumulative relational model of processes can be derived in the following theorem.

The 12th Law of Software Engineering

Theorem 4.3 The *Cumulative Relational Model* (CRM) *of processes* states that a *process P* is the basic unit of an applied computational behavior that is composed by a set of statements s_i, $1 \leq i \leq n$-1, with left-associated cumulative relations, i.e.:

$$\begin{aligned} P &= \mathop{R}_{i=1}^{n-1} (s_i\; r_{ij}\; \mathrm{s}_j), j = i+1 \\ &= (...(((s_1)\; r_{12}\; \mathrm{s}_2)\; r_{23}\; \mathrm{s}_3)\; ...\; r_{n-1,n}\; \mathrm{s}_n) \end{aligned} \tag{4.77}$$

where $s_i \in \mathfrak{P}$ and $r_{ij} \in \mathfrak{R}$.

In Theorem 4.3, $\mathfrak{P}$ and $\mathfrak{R}$ represents a set of meta processes and a set of process relations (operations), which will be defined in Eqs. 4.85 and 4.106, respectively. Theorem 4.3 reveals the nature of programs or software systems as a set of processes.

Definition 4.65 *Real-Time Process Algebra* (RTPA) is a set of formal notations and rules for describing algebraic and real-time relations of software processes.

RTPA is designed as a coherent algebraic software engineering notation system and a formal engineering methodology for addressing the 3-D problems in software system specification, refinement, and implementation, particularly for real-time and embedded systems.

RTPA can be used to describe both logical and physical models of a system. Therefore, logic views of the architecture of a software system and its operational platform can be described using the same set of RTAP notations as that for system behaviors. When the system architecture is formally specified, the static and dynamic behaviors that perform on the system architectural models can be rigorously described by a three-level refinement scheme at the system, class, and object levels in a top-down approach.

4.6.2 THE STRUCTURE OF RTPA

In software engineering, basic requirements for describing and specifying a software system can be considered in two categories: *architectural* components and *operational* components. Corresponding to this classification, system models can be described in three subsystems such as the architecture, static behaviors, and dynamic behaviors. A process can be a single meta process or a complex process that is built upon meta processes by using a set of algebraic process combination rules – the process relations.

Definition 4.66 The *structure of RTPA* is an algebraic software engineering notation system encompassing six subsystems as follows:

$$\begin{aligned} \text{RTPA} \triangleq\ & \text{Meta processes} \\ & \|\ \text{Process relations} \\ & \|\ \text{System architecture models} \\ & \|\ \text{Primary types} \\ & \|\ \text{Abstract dada types} \\ & \|\ \text{Specification refinement scheme} \end{aligned} \tag{4.78}$$

As shown in Eq. 4.78, RTPA is a set of coherent mathematical notations and a formal methodology for modeling software system architectures, static and dynamic behaviors. For modeling system behaviors, RTPA introduces 17 meta processes and 17 process relations, where a meta process is an elementary and primary process that serves as a common and basic building block for a software system, while a complex processes can be derived from meta processes by a set of process relations that serves as process combinatory rules. For modeling system architectures, RTPA provides 17 primitive types and a set of predefined Abstract Data Types (ADTs). Beyond the RTPA notation system, a stepwise system specification and refinement methodology is presented in RTPA that states the architectures and behaviors of any software system can be modeled and specified by a three-step refinement scheme using the same set of RTPA notations.

4.6.3 THE TYPE SYSTEM OF RTPA

The RTPA notation is strongly typed. That is, every operand, variable, constant, object, or architecture in RTPA is assigned with a type labeled as a bold suffix. Every identifier in RTPA is bounded by a type and constrained by additional user-defined domains of values as subsets of the mathematical domains of the predefined RTPA primitive types.

4.6.3.1 Primitive Types and the Type-Suffix Convention

Definition 4.67 A *variable* x with an arbitrary type $\mathbb{T}$ or a *constant* c with an arbitrary type $\mathbb{T}^*$ is an identifier that is first declared and then invoked in the following forms:

(a) Variable declaration: $<x : \mathbb{T} \mid \text{constraints} >$

(b) Constant declaration: $<c : \mathbb{T}^* \mid c\mathbb{T}^* = \text{an instant value}>$

(c) Invocation and reference: $x\mathbb{T}$ or $c\mathbb{T}^*$ (4.79)

where the constraints for variables are usually user-defined scopes of values, and the constraints for constants are specifically bounded values.

The *type-suffix convention* of RTPA provide great convenience for facilitating complicated large-scale system specifications where the type of variables and constants (and their scopes and allowable operations) are always explicitly denoted no matter how far away from where they were declared. The suffix convention of RTPA also dramatically reduces the complexity in syntaxes analysis and code generation.

4.6.3.2 Definitions of the Primitive Types of RTPA

The RTPA *type system* $\mathfrak{T}$ encompasses 17 primitive types elicited from fundamental computing needs, where the names, syntaxes, and mathematical domains of these primitive types are given in Table 4.8.

The 11th Principle of Software Engineering

Theorem 4.4 *Primary types of computational objects* state that the *RTPA type system* $\mathfrak{T}$ encompasses 17 primitive types elicited from fundamental computing needs, i.e.:

$$\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, \\ @e\mathbf{S}, @t\mathbf{TM}, @int\odot, ⓢs\mathbf{BL}\} \tag{4.80}$$

As shown in Table 4.8, the RTPA primitive types #1 through #10 are basic data types. The primitive type *date/time* (#11) is a special type for continuous systems, such as databases and real-time systems where long-range timing manipulation is needed. The *event* type is used to model a *system event* @e**S** (#14) as a string type; a *timing event* **@***t***TM** (#15) as a time type where **TM** is a collective timing type denoting **TM** ∈ {**TI**, **D**, **DT**}; or an *interrupt event* @e⊙ (#16) as an interrupt-point type ⊙. The status type is designated to model *system status* ⓢs**BL** (#17) as a Boolean type.

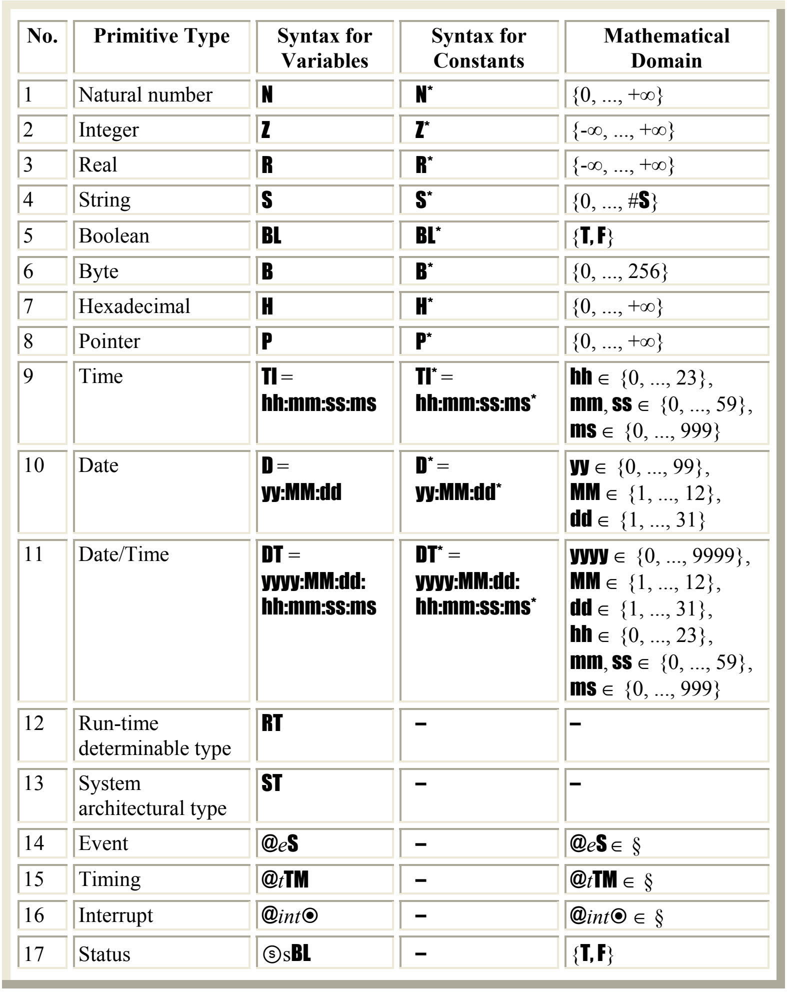
The *run-time determinable type* **RT** (#12) is a subset of all the rest of the primitive types defined in Table 4.8, which is designed to support flexible type specification that is unknown at compile-time, but will be instantiated at run-time. The *system architectural type* **ST** (#13) is a novel and important data type in RTPA that models system architectural components and is going to be further described in Section 5.3.1.

It is noteworthy in Table 4.8 that the primitive types of RTPA in system specification and description can be classified into two categories known as the *variable* and *constant types*. The former are well known in programming, but the latter are equivalently important in system modeling.

The *mathematical domain* given for each type is the scope of values of variables or constants in the type. Formal treatment of RTPA types will be discussed in Section 5.3 on type theories.

Complex types can be derived from the 17 meta types based on a set of architecture composition rules known as the *Component Logical Models* (CLMs) in RTPA [Wang, 2002a/02b/03c/07a]. Details on CLMs will be provided in Sections 4.7.2 and 5.3.1. A set of 11 frequently used complex types in RTPA, such as arrays, stacks, queues, tress, graphs, and files, has been modeled as ADTs [Guttag, 1975/77/02 Guttag, 1975/77/02], which will be discussed in Section 5.3.4.

Table 4.8
Primitive Types of RTPA



No.	Primitive Type	Syntax for Variables	Syntax for Constants	Mathematical Domain
1	Natural number	**N**	$\mathbf{N}^*$	$\{0, ..., +\infty\}$
2	Integer	**Z**	$\mathbf{Z}^*$	$\{-\infty, ..., +\infty\}$
3	Real	**R**	$\mathbf{R}^*$	$\{-\infty, ..., +\infty\}$
4	String	**S**	$\mathbf{S}^*$	$\{0, ..., \#\mathbf{S}\}$
5	Boolean	**BL**	$\mathbf{BL}^*$	$\{\mathbf{T}, \mathbf{F}\}$
6	Byte	**B**	$\mathbf{B}^*$	$\{0, ..., 256\}$
7	Hexadecimal	**H**	$\mathbf{H}^*$	$\{0, ..., +\infty\}$
8	Pointer	**P**	$\mathbf{P}^*$	$\{0, ..., +\infty\}$
9	Time	**TI** = **hh:mm:ss:ms**	$\mathbf{TI}^*$ = $\mathbf{hh{:}mm{:}ss{:}ms}^*$	$\mathbf{hh} \in \{0, ..., 23\}$, $\mathbf{mm}, \mathbf{ss} \in \{0, ..., 59\}$, $\mathbf{ms} \in \{0, ..., 999\}$
10	Date	**D** = **yy:MM:dd**	$\mathbf{D}^*$ = $\mathbf{yy{:}MM{:}dd}^*$	$\mathbf{yy} \in \{0, ..., 99\}$, $\mathbf{MM} \in \{1, ..., 12\}$, $\mathbf{dd} \in \{1, ..., 31\}$
11	Date/Time	**DT** = **yyyy:MM:dd:** **hh:mm:ss:ms**	$\mathbf{DT}^*$ = **yyyy:MM:dd:** $\mathbf{hh{:}mm{:}ss{:}ms}^*$	$\mathbf{yyyy} \in \{0, ..., 9999\}$, $\mathbf{MM} \in \{1, ..., 12\}$, $\mathbf{dd} \in \{1, ..., 31\}$, $\mathbf{hh} \in \{0, ..., 23\}$, $\mathbf{mm}, \mathbf{ss} \in \{0, ..., 59\}$, $\mathbf{ms} \in \{0, ..., 999\}$
12	Run-time determinable type	**RT**	–	–
13	System architectural type	**ST**	–	–
14	Event	$@e\mathbf{S}$	–	$@e\mathbf{S} \in \S$
15	Timing	$@t\mathbf{TM}$	–	$@t\mathbf{TM} \in \S$
16	Interrupt	$@int⦿$	–	$@int⦿ \in \S$
17	Status	ⓢs**BL**	–	$\{\mathbf{T}, \mathbf{F}\}$

4.6.3.3 Equivalence between Primitive Types

Definition 4.68 Type *equivalence* $\simeq$ between two arbitrary primitive types $\mathbb{T}_1$ and $\mathbb{T}_2$ is the property that variables x and y in these types are compatible in all allowable operations, i.e.:

$$\mathbb{T}_1(x) \simeq \mathbb{T}_2(y) \tag{4.81}$$

The 12th Principles of Software Engineering

Theorem 4.5 *Type equivalence* states that two types $\mathbb{T}_1$ and $\mathbb{T}_2$ are *equivalent, iff* the domain of type $\mathbb{T}_1$ is either identical to or a subset of that of $\mathbb{T}_2$, i.e.:

$$\mathbb{T}_1(x) = \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y) \tag{4.82a}$$

or

$$\mathbb{T}_1(x) \subseteq \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y) \tag{4.82b}$$

Examining Table 4.8 according to Theorem 4.5 and Definition 4.68, the following equivalent type categories among the RTPA primitive types can be identified:

$$\mathbf{B} \subseteq \mathbf{P} \subseteq \mathbf{H} \subseteq \mathbf{N} \subseteq \mathbf{Z} \subseteq \mathbf{R} \tag{4.83a}$$

$$\mathbf{TI}, \mathbf{D} \subseteq \mathbf{DT} \tag{4.83b}$$

Generally, let $\mathbb{T}$ be an arbitrary type, and let $\mathfrak{T}$ be the *universal type* or the super type that encompasses all primitive types as defined in Table 4.8, the following type relations can be obtained:

$$\begin{aligned}&\mathbf{RT} \subseteq \mathbb{T} \subseteq \mathfrak{T}\\&= \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @e\mathbf{S}, @t\mathbf{TM}, @int\odot, \textcircled{s}s\mathbf{BL}\}\end{aligned} \tag{4.84}$$

Equivalent type relations can be used to facilitate system specifications and modeling, as well as formal syntax and semantics analyses.

4.6.4 META PROCESSES OF RTPA

Computational operations in conventional process algebra, such as CSP [Hoare, 1985], Timed-CSP [Reed and Roscoe, 1986; Boucher and Gerth, 1987; Schneider, 1991], and other proposals, are treated as a set of processes at the same level. This approach results in an exhaustive listing of processes. Whenever a new operation is identified or required in computing, the existing process system must be extended.

RTPA adopts the foundationalism in order to find the most primitive computational processes known as the *meta processes*. In this approach, complex processes are treated as derived processes from these meta processes based on a set of algebraic process composition rules known as the *process relations*. This subsection describes the set of 17 meta processes of

RTPA. The 17 process relations of RTPA will be presented in the next subsection.

Definition 4.69 A *meta process* in RTPA is a primitive computational operation that cannot be broken down to further individual actions or behaviors.

Meta processes are elementary processes that serve as a basic building block for modeling software behaviors, based on them complex processes can be composed by algebraic operations.

4.6.4.1 Structure of the RTPA Meta Processes

In RTPA, a set of 17 meta processes has been elicited from essential and primary computational operations commonly identified in existing formal methods and modern programming languages [Higman, 1977; Hoare et al., 1987; Wilson and Clark, 1988; Louden, 1993]. Syntaxes and usages of the meta processes are formally described in the following subsections, while semantics of the meta processes will be formally presented in Section 6.6 [Wang, 2006a].

The 13th Principle of Software Engineering

Theorem 4.6 The *meta software processes* state that the *RTPA meta process system* $\mathfrak{P}$ encompasses 17 fundamental computational operations elicited from the most basic computing needs, i.e.:

$$\mathfrak{P} = \{:=, \text{�}, \Rightarrow, \Leftarrow, \not\Leftarrow, \gtrdot, \lessdot, |\gtrdot, |\lessdot, @, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\} \quad (4.85)$$

Names, notations, and syntaxes of the RTPA meta processes are described in Table 4.9. As shown in Table 4.9, each meta process is a basic operation on one or more operand such as variables, memory elements, or I/O ports. Structures of the operands and their allowable operations are constrained by their types as described in Section 4.6.3 where a generic type $\mathbb{T} \in \mathfrak{T}$.

It is noteworthy that not all generally important and fundamental computational operations, as shown in Table 4.9, have been explicitly identified in conventional formal methods such as the evaluation, addressing, memory allocation/release, timing/duration, and the system processes. However, all of them listed above are found necessary and essential in modeling both software system architectures and behaviors.

Table 4.9
RTPA Meta Processes

No.	Meta Process	Notation	Syntax
1	Assignment	$:=$	$y\mathbb{T} := x\mathbb{T}$
2	Evaluation	�	$�_{\mathbb{T}} exp\mathbb{T} \to \mathbb{T}$
3	Addressing	$\Rightarrow$	$id\mathbb{T} \Rightarrow \text{MEM}[ptr\mathbf{P}]\ \mathbb{T}$
4	Memory allocation	$\Leftarrow$	$id\mathbb{T} \Leftarrow \text{MEM}[ptr\mathbf{P}]\ \mathbb{T}$
5	Memory release	$\nLeftarrow$	$id\mathbb{T} \nLeftarrow \text{MEM}[\perp]\mathbb{T}$
6	Read	$\gtrdot$	$\text{MEM}[ptr\mathbf{P}]\mathbb{T} \gtrdot x\mathbb{T}$
7	Write	$\lessdot$	$x\mathbb{T} \lessdot \text{MEM}[ptr\mathbf{P}]\mathbb{T}$
8	Input	$\vert\gtrdot$	$\text{PORT}[ptr\mathbf{P}]\mathbb{T} \vert\gtrdot x\mathbb{T}$
9	Output	$\vert\lessdot$	$x\mathbb{T} \vert\lessdot \text{PORT}[ptr\mathbf{P}]\mathbb{T}$
10	Timing	$\underline{\underline{@}}$	$@t\mathbf{TM}\ \underline{\underline{@}}\ \S t\mathbf{TM}$ $\mathbf{TM} = \mathbf{yy{:}MM{:}dd}$ $\vert\ \mathbf{hh{:}mm{:}ss{:}ms}$ $\vert\ \mathbf{yy{:}MM{:}dd{:}hh{:}mm{:}ss{:}ms}$
11	Duration	$\underline{\underline{\Delta}}$	$@t_n\mathbf{TM}\ \underline{\underline{\Delta}}\ \S t_n\mathbf{TM} + \Delta n\mathbf{TM}$
12	Increase	$\uparrow$	$\uparrow(n\mathbb{T})$
13	Decrease	$\downarrow$	$\downarrow(n\mathbb{T})$
14	Exception detection	!	$!\ (@e\mathbf{S})$
15	Skip	$\otimes$	$\otimes$
16	Stop	$\boxtimes$	$\boxtimes$
17	System	§	$\S(SysID\mathbf{ST})$

4.6.4.2 Formal Description of the RTPA Meta Processes

This subsection provides a formal description of the syntaxes and usages of the 17 meta processes. Deductive semantics of the meta processes of RTPA [Wang, 2006a] will be provided in Sections 6.6.

4.6.4.2.1 Assignment

Definition 4.70 Let x: $\mathbb{T}$ and y: $\mathbb{T}$ be two declared variables with a arbitrary type $\mathbb{T}$**.** An *assignment*, denoted by **:=**, is a meta process that transfers the value of x, $\nu(x)$, to that of y, $\nu(y)$, if their types are identical $\mathbf{T}(x) = \mathbf{T}(y)$ or *equivalent* $\mathbf{T}(x) \simeq \mathbf{T}(y)$, i.e.:

$$y\mathbb{T} := x\mathbb{T} \tag{4.86}$$

where $\mathbb{T} \in \mathfrak{T}$ as defined in Table 4.7, and $x\mathbb{T}$ can be a constant, $x\mathbb{T}^*$, that matches $y\mathbb{T}$.

Note that in case $x\mathbb{T}$ is an expression $exp\mathbb{T}$ in Eq. 4.86, an evaluation of the expression should be carried out first as described by the following meta process.

4.6.4.2.2 Evaluation

Definition 4.71 An *evaluation*, denoted by $\text{�}_{\mathbb{T}}$, is a meta process that maps a given expression in type $\mathbb{T}$ into a value in the same type, i.e.:

$$\text{�}(exp\mathbf{BL})\mathbf{BL} = \text{�}_{\mathrm{BL}}\text{: } exp\mathbf{BL} \to \{\mathbf{T}, \mathbf{F}\} \tag{4.87a}$$

$$\text{�}(exp\mathbf{N})\mathbf{N} = \text{�}_{\mathrm{N}}\text{: } exp\mathbf{N} \to \mathbf{N} \tag{4.87b}$$

$$\text{�}(exp\mathbf{Z})\mathbf{Z} = \text{�}_{\mathrm{Z}}\text{: } exp\mathbf{Z} \to \mathbf{Z} \tag{4.87c}$$

$$\text{�}(exp\mathbf{R})\mathbf{R} = \text{�}_{\mathrm{R}}\text{: } exp\mathbf{R} \to \mathbf{R} \tag{4.87d}$$

$$\text{�}(exp\mathbf{B})\mathbf{B} = \text{�}_{\mathrm{B}}\text{: } exp\mathbf{B} \to \mathbf{B} \tag{4.87e}$$

where the types of the evaluations are called a *Boolean* ($\mathbb{T} = \{\mathbf{BL}\}$), *ordinal* ($\mathbb{T} = \{\mathbf{N}\}$), or *numerical* ($\mathbb{T} = \{\mathbf{Z}, \mathbf{R}, \mathbf{B}\}$) evaluation, respectively.

The preceding evaluations, except that of type Boolean, can be extended to a special type called *power set evaluation* such as Þ**N**, Þ**Z**, Þ**R**, and Þ**B**. For example, an ordinal evaluation on Þ**N** can be defined as follows:

$$\text{�}(exp\mathbf{N})\text{Þ}\mathbf{N} = \text{�}_{\text{ÞN}}\text{: } exp\mathbf{N} \to \text{Þ}\mathbf{N} \tag{4.88}$$

which is frequently used in a switch construct where a branch may be selected by a subset of the numbers in a given range.

Another special type of evaluation is *relational evaluations* that compare two variables or expressions by a binary relation R, $R \in \mathbb{R}$, in the following form:

$$\text{�}(exp_1\mathbb{T}, exp_2\mathbb{T})\mathbb{R} = \text{�}_{\mathbb{R}}\text{: } exp\mathbb{T} \times exp\mathbb{T} \to \mathbb{R} \tag{4.89}$$

where $\mathbb{R} = \{=, \neq, >, <, \geq, \leq\}$.

4.6.4.2.3 Addressing

An addressing function, π: $id\mathbb{T} \to ptr\text{Þ}\mathbf{P}$, has been introduced in Eq. 4.40a for mapping a given logical $id\mathbb{T}$ into the first byte of the physical memory block MEM[$ptr\mathbf{P}$, $ptr\mathbf{P}+n-1$]$\mathbb{T}$. The addressing process of RTPA is formally defined as follows.

Definition 4.72 *Addressing*, denoted by $\Rightarrow$, is a meta process that maps a given logical $id\mathbb{T}$ into a block of the physical memory $\text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T}$, denoted by $ptr\text{Þ}\mathbf{P}$ accommodating n bytes of memory for the variable in type $\mathbb{T}$, $\mathbb{T} \in \{\mathbf{P}, \mathbf{H}, \mathbf{N}, \mathbf{Z}\}$, i.e.:

$$\begin{array}{l} id\mathbb{T} \Rightarrow \text{MEM}[ptr\text{Þ}\mathbf{P}]\mathbb{T} \\ \quad \triangleq (\pi\text{: } id\mathbb{T} \rightarrow ptr\text{Þ}\mathbf{P} \\ \qquad \Leftrightarrow id\mathbb{T} = \text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T} \\ \qquad) \end{array} \tag{4.90}$$

where n is language implementation-dependent.

A formal description of the abstract memory model, $\text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T}$, will be provided in Section 5.3.1.4.

4.6.4.2.4 Memory Allocation

Definition 4.73 *Memory allocation*, denoted by $\Leftarrow$, is a meta process that collects a unique memory block logically named $id\mathbb{T}$ and physically located by $ptr\text{Þ}\mathbf{P}$ accommodating n bytes of memory for the variable in type $\mathbb{T}$, i.e.:

$$\begin{array}{l} id\mathbb{T} \Leftarrow \text{MEM}[ptr\text{Þ}\mathbf{P}]\mathbb{T} \\ \quad \triangleq (\pi^{-1}\text{: } ptr\text{Þ}\mathbf{P} \rightarrow id\mathbb{T} \\ \qquad \Leftrightarrow id\mathbb{T} = \text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T} \\ \qquad) \end{array} \tag{4.91}$$

where π^{-1}: $ptr\text{Þ}\mathbf{P} \rightarrow id\mathbb{T}$ is the memory allocation function that is an inverse function of addressing, which associates a physical memory block $\text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T}$ with the given logical $id\mathbb{T}$.

Memory allocation is a key meta process for dynamic memory manipulation implemented in the RTPA support system at a part of the operating system.

4.6.4.2.5 Memory Release

Definition 4.74 *Memory release*, denoted by $\nLeftarrow$, is a meta process that dissociates and frees a unique block of n continuous physical memory elements denoted by $ptr\text{Þ}\mathbf{P}$ from its logical identifier $id\mathbb{T}$, i.e.:

$$\begin{array}{l} id\mathbb{T} \nLeftarrow \text{MEM}[\perp]\mathbb{T} \\ \quad \triangleq (\pi\text{: } id\mathbb{T} \rightarrow ptr\text{Þ}\mathbf{P} \\ \qquad \Leftrightarrow (\text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T} := \perp \wedge ptr\mathbf{P} := \perp \wedge id\mathbb{T} := \perp) \\ \qquad) \end{array} \tag{4.92}$$

where $\perp$ denotes an undefined or unallocated value.

The released memory block that was logically identified by $id\mathbb{T}$ will then be returned to the system memory pool collected by the system memory management function provided by an operating system or the RTPA support system.

4.6.4.2.6 Read

Definition 4.75 *Read*, denoted by $\gtrdot$, is a meta process that gets data $x\mathbb{T}$ from a given memory location MEM[$ptr\mathbf{P}$], where $Ptr\mathbf{P}$ is a pointer that identifies the physical memory address, i.e.:

$$\text{MEM}[ptr\mathbf{P}]\mathbb{T} \gtrdot x\mathbb{T} \tag{4.93}$$

where $\mathbb{T} \in \mathfrak{T}$.

4.6.4.2.7 Write

Definition 4.76 *Write*, denoted by $\lessdot$, is a meta process that puts data $x\mathbb{T}$ to a given memory location MEM[$ptr\mathbf{P}$], where $ptr\mathbf{P}$ is a pointer that identifies the physical memory address, i.e.:

$$x\mathbb{T} \lessdot \text{MEM}[ptr\mathbf{P}]\mathbb{T} \tag{4.94}$$

where $\mathbb{T} \in \mathfrak{T}$.

4.6.4.2.8 Input

Definition 4.77 *Input*, denoted by $|\gtrdot$, is a meta process that receives data $x\mathbb{T}$ from a given system I/O port PORT[$ptr\mathbf{P}$]$\mathbb{T}$, where $ptr\mathbf{P}$ is a pointer that identifies the physical address of the port interface, i.e.:

$$\text{PORT}[ptr\mathbf{P}]\mathbb{T} \;|\gtrdot\; x\mathbb{T} \tag{4.95}$$

where $\mathbb{T} \in \{\mathbf{B}, \mathbf{S}\}$.

A formal description of the port model, PORT[$ptr\mathbf{P}$]$\mathbb{T}$, will be provided in Section 5.3.1.4.

4.6.4.2.9 Output

Definition 4.78 *Output*, denoted by $|\lessdot$, is a meta process that sends data $x\mathbb{T}$ to a given system I/O port PORT[$ptr\mathbf{P}$]$\mathbb{T}$, where $ptr\mathbf{P}$ is a pointer that identifies the physical address of the port interface, i.e.:

$$x\mathbb{T} \mid\!\lessdot \text{PORT}[ptr\mathbf{P}]\mathbb{T} \qquad (4.96)$$

where $\mathbb{T} \in \{\mathbf{B}, \mathbf{S}\}$.

4.6.4.2.10 Timing

Definition 4.79 *Timing*, denoted by $\underset{=}{@}$, is a meta process that sets the value of a timing variable @t as the absolute time of the current system clock §t, i.e.:

$$@\text{t}\mathbf{TM} \ \underset{=}{@} \ \S\text{t}\mathbf{TM} \qquad (4.97)$$

where $\mathbf{TM} \in \{\mathbf{TI}, \mathbf{D}, \mathbf{DT}\}$; each subdomain of time and/or date has been defined in Table 4.8, respectively.

A specific timing process may adopt one of the following expressions depending on the need for the range of time in a given system:

$$@\text{t}\mathbf{hh{:}mm{:}ss{:}ms} \ \underset{=}{@} \ \S\text{t}\mathbf{hh{:}mm{:}ss{:}ms} \qquad (4.98a)$$

$$@\text{t}\mathbf{yy{:}MM{:}dd} \ \underset{=}{@} \ \S\text{t}\mathbf{yy{:}MM{:}dd} \qquad (4.98b)$$

$$@\text{t}\mathbf{yyyy{:}MM{:}dd{:}hh{:}mm{:}ss{:}ms} \ \underset{=}{@} \ \S\text{t}\mathbf{yyyy{:}MM{:}dd{:}hh{:}mm{:}ss{:}ms} \qquad (4.98c)$$

4.6.4.2.11 Duration

Definition 4.80 *Duration*, denoted by $\triangleq$, is a meta process that sets a relative time $@\text{t}_\text{n}\mathbf{N}$ as an integer based on the relative system clock $\S\text{t}_\text{n}\mathbf{N}$ and the given period $\Delta\text{n}\mathbf{N}$, i.e.:

$$@\text{t}_\text{n}\mathbf{N} \triangleq \S\text{t}_\text{n}\mathbf{N} + \Delta\text{n}\mathbf{N} \qquad (4.99)$$

where the unit of all relative timing variables is $\mathbf{ms}$.

4.6.4.2.12 Increase

Definition 4.81 *Increase*, denoted by $\uparrow$, is a meta process that adds one to a given variable $n\mathbb{T}$, i.e.:

$$\uparrow\!(n\mathbb{T}) \qquad (4.100)$$

where $\mathbb{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}\}$.

4.6.4.2.13 Decrease

Definition 4.82 *Decrease*, denoted by ↓, is a meta process that subtracts one from a given variable $n\mathbb{T}$, i.e.:

$$\downarrow(n\mathbb{T}) \tag{4.101}$$

where $\mathbb{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}\}$.

4.6.4.2.14 Exception Detection

Definition 4.83 *Exception detection*, denoted by !, is a meta process that logs a detected exception event $@e\mathbf{S}$ at run-time, i.e.:

$$!\,(@e\mathbf{S}) \tag{4.102}$$

The RTPA exceptional detection mechanism is a fundamental process for safety and dependable system modeling, which enables system exception detection, handling, and postmortem analysis to be implemented.

4.6.4.2.15 Skip

Definition 4.84 *Skip*, denoted by ⊗, is a meta process that exits a current control structure, such as loop, branch, or switch and return to the immediate upper layer of the current control structure, i.e.:

$$\otimes \tag{4.103}$$

A skip process implements no externally observable behaviors, but it conducts important internal control operations. The semantics of skip is equivalent to the behaviors of *exit*, *break*, or unconditional jump to a predefined program address. Further explanation of the semantics of the skip process may be referred to Section 6.6.1.15.

4.6.4.2.16 Stop

Definition 4.85 *Stop*, denoted by ⊠, is a meta process that terminates a system's operation, i.e.:

$$\boxtimes \tag{4.104}$$

Note that the stop process is a system level process that shuts down the system physically.

4.6.4.2.17 System

Definition 4.86 *The system*, denoted by §, is a top-level meta process that acts at the highest level controller of a process system for dispatching and/or executing a specific process according to system timing or predefined events, i.e.:

$$\S(\mathit{SysID}\mathbf{S}) \tag{4.105}$$

where $\mathit{SysID}\mathbf{S}$ is an identity of the system in string type.

A formal description of an abstract system underlying the computing platform will be provided in Section 5.6.1.

4.6.5 PROCESS RELATIONS AND ALGEBRAIC OPERATIONS OF RTPA

The meta processes of RTPA developed in Section 4.6.4 identified a set of fundamental elements for modeling the most basic behaviors of a software system. It is interesting to realize that there is only a small set of 17 meta processes in software system modeling. However, via the combination of a number of the meta processes by certain algebraic process operations, any architecture and behavior of real-time or nonreal-time software systems can be sufficiently described [Higman, 1977; Hoare et al., 1987; Wang and King, 2000].

Definition 4.87 A *process relation* in RTPA is an algebraic operation and a compositional rule between two or more meta processes in order to construct a complex process.

A set of 17 fundamental process relations has been elicited for building and composing complex processes in the context of real-time software systems. Syntaxes and usages of the 17 RTPA meta processes are formally described in this subsection. Semantics of these process relations will be formally described in Section 6.6.

4.6.5.1 Structure of the RTPA Process Relations

A set of 17 process relations $\mathfrak{R}$ has been elicited from fundamental algebraic and relational operations in computing, where definitions and syntaxes of each of these process relations will be provided in the following subsections.

The 14th Principle of Software Engineering

Theorem 4.7 The software composing rules state that the RTPA *process relation system* $\mathfrak{R}$ encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs, i.e.:

$$\mathfrak{R} = \{\rightarrow, \curvearrowright, |, |\ldots|\ldots, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \|, \oint\!\!\oint, |||, \gg, \lightning, \hookrightarrow_{t}, \hookrightarrow_{e}, \hookrightarrow_{i}\} \tag{4.106}$$

Names, notations, and syntaxes of the 17 RTPA process relations are described in Table 4.10. The first 7 process relations in Table 4.10, *sequential* (#1), *jump* (#2), *branch* (#3), *switch* (#4), and *iterations* (#5 through #7), have long been identified as the Basic Control Structures (BCS's) of software architectures [Hoare et al., 1987; Wilson and Clark, 1988]. To represent the modern programming structural concepts, CSP [Hoare, 1985] identified the following 7 additional process relations such as *recursion* (#8), *function call* (#9), *parallel* (#10), *concurrency* (#11), *interleave* (#12), *pipeline* (#13), and *interrupt* (#14).

However, these process relations or operations were treated the same as the meta processes in existing formal methods. That is, the conventional notation systems are not an algebraic production system rather than an exhaustive instruction system, which do not distinguish the basic computational operations and their composing rules.

RTPA [Wang, 2002a/02b/03c/07a] extends the BCS's and process relations to *time-driven dispatch* (#15), *event-driven dispatch* (#16), *and interrupt-driven dispatch* (#17) in order to model the top-level system behaviors, particularly those of real-time systems. The 17 process relations (BCS's) are regarded as the foundation of programming and system architectural design, because any complex process can be implemented by the algebraic process composing operations onto the set of the 17 meta processes as defined in Table 4.9.

Table 4.10
RTPA Process Relations and Algebraic Operations

No.	Process Relation	Notation	Syntax
1	Sequence	$\rightarrow$	$P \rightarrow Q$
2	Jump	$\curvearrowright$	$P \curvearrowright Q$
3	Branch	$\mid$	�$exp\mathbf{BL} = \mathbf{T} \rightarrow P$ $\mid$ �$\sim \rightarrow Q$
4	Switch	$\mid$ $\ldots$ $\mid$	�$\mathrm{exp}\mathbb{T} = \ \mathrm{i} \rightarrow P_i$ $\mid \sim \rightarrow \oslash$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$
5	While-loop	R^{*}	$\mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}} P$
6	Repeat-loop	R^{+}	$P \rightarrow \mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}} P$
7	For-loop	R^{i}	$\mathop{R}\limits_{i\mathbf{N}=1}^{n\mathbf{N}} P(i\mathbf{M})$
8	Recursion	$\circlearrowleft$	$\mathop{R}\limits_{i\mathbf{N}=n\mathbf{N}}^{0} P^{i\mathbf{M}} \circlearrowleft P^{i\mathbf{M}-1}$
9	Function call	$\rightarrowtail$	$P \rightarrowtail F$
10	Parallel	$\|\|$	$P \mid\mid Q$
11	Concurrence	∯	P ∯ Q
12	Interleave	$\|\|\|$	$P \mid\mid\mid Q$
13	Pipeline	»	P » Q
14	Interrupt	↯	P ↯Q
15	Time-driven dispatch	$\hookrightarrow_t$	$@t_i\mathbf{TM} \hookrightarrow_t P_i$
16	Event-driven dispatch	$\hookrightarrow_e$	$@e_i\mathbf{S} \hookrightarrow_e P_i$
17	Interrupt-driven dispatch	$\hookrightarrow_i$	$@int_j\odot \hookrightarrow_i P_j$

The 13th Law of Software Engineering

Theorem 4.8 The *express power of algebraic modeling* states that the total number of the possible computational operations $\mathcal{N}$ is a set of combinations between two arbitrary meta processes $\mathbb{P}_1, \mathbb{P}_2 \in \mathfrak{P}$ composed by each of the process relations $\mathbb{R} \in \mathfrak{R}$ in RTPA, i.e.:

$$\begin{aligned}\mathcal{N} &= \#\mathfrak{R} \bullet C^{2}_{\#\mathfrak{P}} \\ &= 17 \bullet \frac{17!}{2!(17\text{-}2)!} \\ &= 17 \bullet 136 \\ &= 2{,}312\end{aligned} \tag{4.107}$$

Theorem 4.8 demonstrates the power of the algebraic structure towards computational behavior modeling and programming. It is noteworthy that an ordinary high level programming language may introduce about 150 to 300 individual instructions. However, the expressive power of RTPA is in a very high order in program composition, i.e., one order higher than any programming language, although it just adopts a small set of 17 meta processes and 17 process relations.

In Table 4.10, the big-R used in process relations #5 through #8 is a special notation recently created for denoting iterative and recursive behaviors of software systems [Wang, 2006f]. Formal models of the big-R notation have been given in Section 4.5.3, and its applications in iterative and recursive behavioral modeling will be provided in Section 5.4.2.

4.6.5.2 Formal Description of the RTPA Process Relations

This subsection defines and explains the 17 process relations of RTPA for manipulating process operations and combinational rules between meta processes. Deductive semantics of the process relations [Wang, 2006a] will be provided in Section 6.6.2.

4.6.5.2.1 Sequence

Definition 4.88 A *sequence*, denoted by →, is a process relation in which two meta processes *P* and *Q* are executed one by one, i.e.:

$$P \to Q \tag{4.108}$$

4.6.5.2.2 Jump

Definition 4.89 A *jump*, denoted by $\curvearrowright$, is a process relation in which, on the termination of a process *P*, the system exits the linear sequence of processes, and invokes a designated process *Q*, i.e.:

$$P \curvearrowright Q \tag{4.109}$$

where Q is usually identified as a label or a logical address.

4.6.5.2.3 Branch

Definition 4.90 A *branch*, denoted by |, is a process relation in which the selection of a process is determined by a conditional expression *exp*$\mathbf{BL}$, i.e.:

$$\begin{aligned} &\text{�}exp\mathbf{BL} = \mathbf{T} \to P \\ &|\ \text{�}\sim\ \to Q \end{aligned} \tag{4.110}$$

where '~' means '*exp*$\mathbf{BL}$ = $\mathbf{F}$,' or more general, 'otherwise.' When the *else* branch is optional, Eq. 4.110 is equivalent to:

$$\begin{aligned} &\text{�}exp\mathbf{BL} = \mathbf{T} \to P \\ &|\ \text{�}\sim\ \to \varnothing \end{aligned} \tag{4.111}$$

4.6.5.2.4 Switch

Definition 4.91 A *switch*, denoted by | … | …, is a process relation in which the branch is determined by a numerical expression *exp*$\mathbb{T}$, i.e.:

$$\begin{aligned} \text{�}exp\mathbb{T} = \ & 0 \to \mathrm{P}_0 \\ & |\ 1 \to \mathrm{P}_1 \\ & |\ \ldots \\ & |\ \mathrm{n\text{-}1} \to \mathrm{P}_{\mathrm{n-1}} \\ & |\ \sim\ \to \varnothing \end{aligned} \tag{4.112}$$

where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$.

4.6.5.2.5 While-Loop

Definition 4.92 A *while-loop*, denoted by R^*, is a process relation in which a meta process or a complex process, *P*, is executed repeatedly as long as the conditional expression *exp*$\mathbf{BL}$ is true, i.e.:

$$\mathrm{R}^{*} \triangleq \underset{exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{\mathrm{R}}} P \tag{4.113}$$

where the lower bound, $exp\mathbf{BL} = \mathbf{T}$, denotes that P may or may not be iterated at run-time if $exp\mathbf{BL} \neq \mathbf{T}$ at the beginning; the upper bound, $exp\mathbf{BL} = \mathbf{F}$, shows the condition to terminate the iteration.

4.6.5.2.6 Repeat-Loop

Definition 4.93 A *repeat-loop*, denoted by R^{+}, is a process relation in which a meta process or a complex process, P, is executed iteratively for at least one time until the conditional expression $exp\mathbf{BL} = \mathbf{F}$, i.e.:

$$\mathrm{R}^{+} \triangleq P \rightarrow \underset{exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{\mathrm{R}}} P \tag{4.114}$$

It can be seen that a repeat-loop is a sequential combination of P and a while-loop of P, i.e.:

$$\mathrm{R}^{+}P = P \rightarrow \mathrm{R}^{*}P \tag{4.115}$$

4.6.5.2.7 For-Loop

A for-loop is a special form of the while-loop, where the termination condition of iteration will be determined by a known constant or expression.

Definition 4.94 A *for-loop*, denoted by R^{i}, is a process relation in which a meta process or a complex process, $P(i)$, is executed repeatedly for n times controlling by an index i, $i \in \{1, \ldots, n\}$, i.e.:

$$\mathrm{R}^{i} \triangleq \underset{i\mathbf{N}=1}{\overset{n}{\mathrm{R}}} P(i) \tag{4.116}$$

Examples of the three forms of iterations as defined in Definitions 4.92 through 4.94 will be provided in Section 5.4 on modeling iterative behaviors of software systems.

4.6.5.2.8 Recursion

Definition 4.95 A *recursion*, denoted by $\circlearrowleft$, is a process relation in which a process *P* calls itself, i.e.:

$$\mathrm{P} \circlearrowleft \mathrm{P} \tag{4.117}$$

Recursion processes are frequently used in programming to simplify system structures and to specify neat and expressive system functions. It is particularly useful when an infinite or run-time determinable specification has to be clearly expressed. Examples of recursions will be provided in Chapter 6 on modeling recursive behaviors of software systems.

4.6.5.2.9 Function Call

Definition 4.96 A *function call*, denoted by $\rightarrowtail$, is a process relation in which a process *P* calls another process *Q* as a predefined sub-process, i.e.:

$$P \rightarrowtail Q \tag{4.118}$$

In Eq. 4.118, the called process *Q* can be regarded as an embedded part of process *P*.

4.6.5.2.10 Parallel

Definition 4.97 A *parallel*, denoted by ||, is a process relation in which two processes *P* and *Q* are executed simultaneously synchronizing by a common system clock, i.e.:

$$\mathrm{P} \parallel \mathrm{Q} \tag{4.119}$$

The parallel process relation is designed to model behaviors of a *Multi-Processor Single-Clock* (MPSC) system as shown in Table 4.10 (#10). The syntax of parallel processes may also be extended to denote relations between system architectural models that are functionally parallel or equivalent.

4.6.5.2.11 Concurrence

Definition 4.98 A *concurrence*, denoted by $\oiint$, is a process relation in which two processes *P* and *Q* are executed simultaneously and asynchronously according to separate system clocks, and each such process is executed as a complete task, i.e.:

$$P \oiint Q \tag{4.120}$$

The concurrent process relation is designed to model behaviors of a *Multi-Processor Multi-Clock* (MPMC) system.

4.6.5.2.12 Interleave

Definition 4.99 An *interleave*, denoted by |||, is a process relation in which two processes *P* and *Q* are executed simultaneously, synchronized by a common system clock, while the execution of each such process would be alternatively dispatched as a multi-threads system, i.e.:

$$P \; ||| \; Q \tag{4.121}$$

The interleave process relation is designed to model behaviors of a *Single-Processor Single-Clock* (SPSC) system.

4.6.5.2.13 Pipeline

Definition 4.100 A *pipeline*, denoted by », is a process relation in which two processes *P* and *Q* are interconnected to each other, and the succeeding process takes the output(s) of the previous process as its input(s), i.e.:

$$P \gg Q \tag{4.122}$$

4.6.5.2.14 Interrupt

Definition 4.101 An *interrupt*, denoted by ↯, is a process relation in which a running process *P* is temporarily held before completion by an interrupt event $@e\odot$ at the interrupt point $\odot$ sent by another process *Q* with a higher priority, and the interrupted process will be resumed when *Q* has been completed, i.e.:

$$P ↯ Q \triangleq P \,||\, (@int\odot \nearrow Q \searrow \odot) \tag{4.123}$$

where $\nearrow$ and $\searrow$ denote *interrupt service* and *interrupt return*, respectively.

The interrupt relation describes execution priority and control taking-over between processes.

4.6.5.2.15 Time-Driven Dispatch

Definition 4.102 A *time-driven dispatch*, denoted by $\hookrightarrow_{\mathrm{t}}$, is a process relation in which the *i*th process P_i is triggered by a predefined system time $@t_i\mathbf{TM}$, i.e.:

$$@t_i\mathbf{TM} \hookrightarrow_{\mathrm{t}} P_i\,,\ \ i \in \{1, ..., n\} \tag{4.124}$$

4.6.5.2.16 Event-Driven Dispatch

Definition 4.103 An *event-driven dispatch*, denoted by $\hookrightarrow_{\mathrm{e}}$, is a process relation in which the *i*th process P_i is triggered by a predefined system event $@e_i\mathbf{S}$, i.e.:

$$@e_i\mathbf{S} \hookrightarrow_{\mathrm{e}} P_i\,,\ \ i \in \{1, \ldots, n\} \tag{4.125}$$

4.6.5.2.17 Interrupt-Driven Dispatch

Definition 4.104 An *interrupt-driven dispatch*, denoted by $\hookrightarrow_{\mathrm{i}}$, is a process relation in which the *i*th process P_i is triggered by a predefined system interrupt $@int_i\odot$, i.e.:

$$@int_j\odot \hookrightarrow_{\mathrm{i}} P_j\,,\ \ j \in \{1, \ldots, n\} \tag{4.126}$$

All types of events, including the operational events, timing events, and interrupt events, are captured by the system in order to dispatch a designated process. Detailed models and mechanisms of system event capture and dispatch will be discussed in Section 5.4.3.

4.7 The RTPA Methodology for Software System Modeling and Refinement

In common engineering practice, a complicated system should be specified via a number of systematic refinements in a top-down approach by using a set of coherent notations. On the basis of the RTPA notation system developed in Section 4.6, this section describes the RTPA modeling, specification, and refinement methodology for software system architectures, static, and dynamic behaviors via three-level refinements.

4.7.1 THE RTPA METHODOLOGY

In RTPA three fundamental facets of software systems are recognized as the architecture, static behaviors, and dynamic behaviors. The top-level specification of a software system can be denoted by a coherent set of mathematical notations of RTPA as follows.

Definition 4.105 A *software system model* in RTPA, §(*SysID***ST**), encompasses the following three subsystems, i.e.:

$$\begin{aligned}\S(\mathit{SysID}\mathbf{ST}) \triangleq\ & \text{SysID}\mathbf{ST}.\text{Architecture} \\ & \parallel \text{SysID}\mathbf{ST}.\text{StaticBehaviors} \\ & \parallel \text{SysID}\mathbf{ST}.\text{DynamicBehaviors} \qquad (4.127)\end{aligned}$$

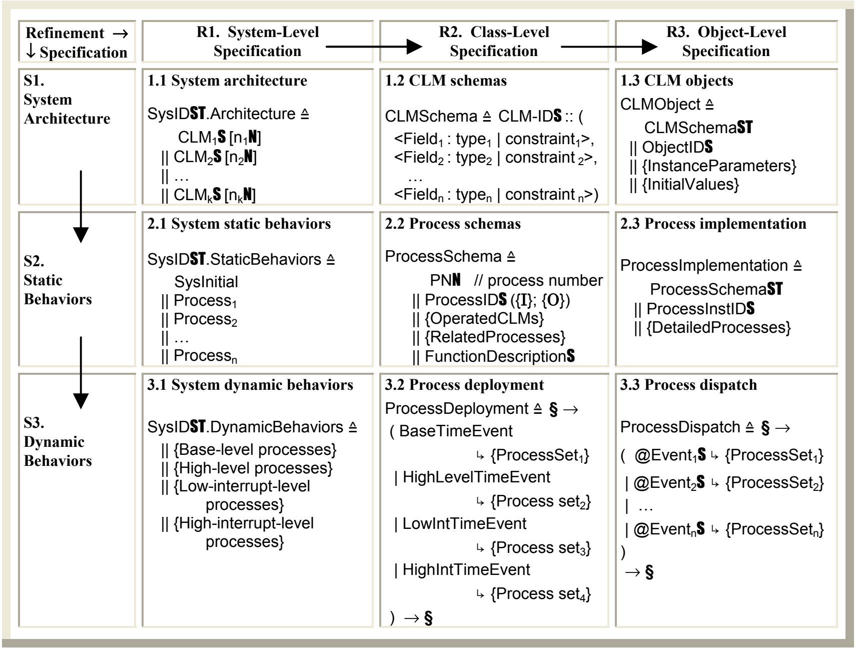
where **ST** is the system type suffix.

The RTPA specification and refinement methodology can be described as a 3 × 3 matrix as shown in Fig. 4.2. Each of the subsystems described in Eq. 4.127 can be systematically extended by a three-level refinement process at the *system, class,* and *object* levels. Fig. 4.2 shows the method and expected work products of each specification subsystem at a different level of system refinement.

In the RTPA specification and refinement scheme for software systems, two key modeling methods, the *component logical model* and *process*, are introduced to model software system architectures and behaviors. The mathematical model of the latter has been described in Definitions 4.64 and 4.69. The definition of the former is given below.

Definition 4.106 A *Component Logical Model* (CLM) is an abstract model of a system architectural component that represents a hardware interface, an internal logical model, a data structure, and/or a common control structure of a system.

According to Fig. 4.2, the three refinement steps for system architecture specification (S1) are: 1.1 *system architecture*, 1.2 *CLM schemas*, and 1.3 *CLM objects*. Similarly, the refinement strategy for system static behavior specification (S2) is: 2.1 *system static behaviors*, 2.2 *process schemas*, and 2.3 *process implementations*. System dynamic behaviors (S3) can be specified by: 3.1 *system dynamic behaviors*, 3.2 *process deployment*, and 3.3 *process dispatch*, in a three-level refinement. Detailed explanations and illustrations of the RTPA scheme for software system specification and refinement will be given in Sections 4.7.2 through 4.7.4.



Refinement → **↓ Specification**	**R1. System-Level Specification**	**R2. Class-Level Specification**	**R3. Object-Level Specification**
S1. **System Architecture**	**1.1 System architecture** SysID**ST**.Architecture ≙ CLM$_1$**S** [n$_1$**N**] || CLM$_2$**S** [n$_2$**N**] || … || CLM$_k$**S** [n$_k$**N**]	**1.2 CLM schemas** CLMSchema ≙ CLM-ID**S** :: (<Field$_1$: type$_1$ | constraint$_1$>, <Field$_2$: type$_2$ | constraint $_2$>, … <Field$_n$: type$_n$ | constraint $_n$>)	**1.3 CLM objects** CLMObject ≙ CLMSchema**ST** || ObjectID**S** || {InstanceParameters} || {InitialValues}
S2. **Static Behaviors**	**2.1 System static behaviors** SysID**ST**.StaticBehaviors ≙ SysInitial || Process$_1$ || Process$_2$ || … || Process$_n$	**2.2 Process schemas** ProcessSchema ≙ PN**N** // process number || ProcessID**S** ({I}; {O}) || {OperatedCLMs} || {RelatedProcesses} || FunctionDescription**S**	**2.3 Process implementation** ProcessImplementation ≙ ProcessSchema**ST** || ProcessInstID**S** || {DetailedProcesses}

S3.
Dynamic Behaviors
	3.1 System dynamic behaviors
	SysID**ST**.DynamicBehaviors ≙
	|| {Base-level processes}
	|| {High-level processes}
	|| {Low-interrupt-level processes}
	|| {High-interrupt-level processes}
		3.2 Process deployment
		ProcessDeployment ≙ § →
		(BaseTimeEvent
		↳ {ProcessSet$_1$}
		| HighLevelTimeEvent
		↳ {Process set$_2$}
		| LowIntTimeEvent
		↳ {Process set$_3$}
		| HighIntTimeEvent
		↳ {Process set$_4$}
		) → §
			3.3 Process dispatch
			ProcessDispatch ≙ § →
			(@Event$_1$**S** ↳ {ProcessSet$_1$}
			| @Event$_2$**S** ↳ {ProcessSet$_2$}
			| …
			| @Event$_n$**S** ↳ {ProcessSet$_n$}
			)
			→ §

Figure 4.2 The scheme of system modeling and refinement in RTPA

There are four basic types of system architectures known as: *parallel*, *serial*, *pipeline*, and *nested* as shown in Fig. 4.3. Any complicated system architecture can be represented by a combination of these four basic architectures between components. It is interesting to find that each of the basic architectures corresponds to a key RTPA process relation as defined in Table 4.10. Therefore, for the first time, not only system behaviors (operations), but also system architectures can be expressed by the same set of algebraic notations in RTPA.

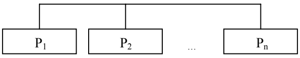
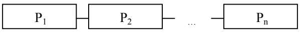
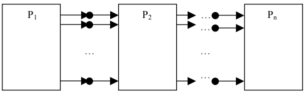
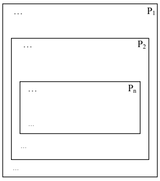
No.	Type of Architecture	Syntax	Example
1	Parallel	$P \| Q$	§(ParallelSys**ST**) $\triangleq P_1 \| P_2 \| \ldots \| P_n$ 
2	Serial	$P \rightarrow Q$	§(SerialSys**ST**) $\triangleq P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$ 
3	Pipeline	$P \gg Q$	§(PipelineSys**ST**) $\triangleq P_1 \gg P_2 \gg \ldots \gg P_n$ 
4	Nested	$P \rightarrowtail Q$	§(NestedSys**ST**) $\triangleq P_1 \rightarrowtail P_2 \rightarrowtail \ldots \rightarrowtail P_n$ 

Figure 4.3 RTPA meta architectures

Example 4.21 The architecture of a sample system §(SysA**ST**) is given in the left-hand side of Fig. 4.4, which consists of serial, parallel, and nested architectural relations among system components. Applying the RTPA methodology, the architecture of §(SysA**ST**) can be formally specified as shown in the right-hand side of Fig. 4.4.

The formal architectural specification of real-world systems with hardware and software subsystems will be demonstrated in Section 4.8.

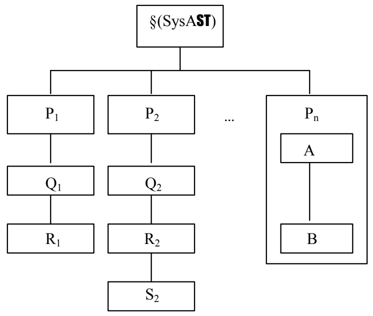
System Architecture	RTPA Expression
 §(SysA**ST**) P1 P2 ... Pn A B Q1 Q2 R1 R2 S2	$\S(\text{SysA}\mathbf{ST}) \triangleq \S \rightarrow$ $(\ (P_1 \rightarrow Q_1 \rightarrow R_1)$ $\| \ (P_2 \rightarrow Q_2 \rightarrow R_2 \rightarrow S_2)$ $\| \ \ldots$ $\| \ (P_n \rightarrowtail (A \rightarrow B))$ $)$

Figure 4.4 The architecture of a sample system

4.7.2 SYSTEM ARCHITECTURE MODELING AND REFINEMENT IN RTPA

In RTPA, the *architectural components* of a system are modeled by CLMs, which are an abstract model of the system hardware interface, an internal logic model of hardware, and/or a common control structure of a system [Wang, 2002a]. The *operational components* of a system are modeled by processes, which will be described in Section 4.8.

4.7.2.1 The System Architecture

System architecture, at the top level, specifies a list of names of CLMs and their relations. A CLM can be regarded as a predefined class of system hardware or internal control models, which can be inherited or implemented by corresponding CLM objects as specific instances in the succeeding system architectural refinement procedures.

Definition 4.107 The *architecture* of a software system can be described as a set of *k* parallel CLMs in RTPA, i.e.:

$$\begin{aligned}\text{SysID}\mathbf{ST}.\text{Architecture}\mathbf{ST} \triangleq \ & \text{CLM}_1\mathbf{ST}\,[n_1\mathbf{N}] \\ & \| \ \text{CLM}_2\,\mathbf{ST}\,[n_2\,\mathbf{N}] \\ & \| \ \ldots \\ & \| \ \text{CLM}_k\mathbf{ST}\,[n_k\,\mathbf{N}] \end{aligned} \qquad (4.128)$$

where **ST** is the suffix of the system architectural type that denotes a CLM in the type of system architectures, **N** is a type suffix of natural numbers, and n_i**N** is the given number of the *i*th CLM configured in the system.

It can be seen that types play an important role in modeling system architectural entities such as CLMs and data objects. A complete definition of the RTPA type system may refer to Section 4.6.3.

Eq. 4.128 provides the first-step refinement of the architectural specification of the given system represented by SysID**ST**.Architecture**ST**. As a result, the system's *architectural components* and their relationships are clearly specified.

4.7.2.2 The CLM Schema

Definition 4.108 A *CLM schema* is an abstract logical structure of a component logical model in the form of a record-like abstract data structure that specifies the configuration of a CLM by a set of *n* fields, types, and their constraints, i.e.:

$$\begin{aligned}\text{CLMSchema}\mathbf{ST} \triangleq\ & \text{CLM\_ID}\mathbf{S}\text{::} \\ & (\ \text{Field}_1 : \text{type}_1 \mid \text{constraint}_1>, \\ & \ \ \text{Field}_2 : \text{type}_2 \mid \text{constraint}_2>, \\ & \ \ \ldots \\ & \ \ \text{Field}_n : \text{type}_n \mid \text{constraint}_n> \\ &) \end{aligned} \tag{4.129}$$

A CLM schema can be treated as the architectural specification of a class, which will be used as a blueprint in further refinement of the CLM objects as an instance in implementing the CLM classes in the next step.

4.7.2.3 The CLM Objects

The instantiation of a CLM schema is called a *CLM object* in RTPA. The CLM object is the result of the final refinement of the specification in order to obtain the architectural model of a given software system.

Definition 4.109 *A CLM object* in RTPA is a derived instance of a CLM schema and its detailed implementation, i.e.:

$$\begin{aligned}\text{CLMObject}\mathbf{ST} \triangleq\ & \text{CLMSchema}\mathbf{ST} \\ & \parallel \text{ObjectID}\mathbf{S} \\ & \parallel \{\text{InstanceParameters}\} \\ & \parallel \{\text{InitialValues}\} \end{aligned} \tag{4.130}$$

After the three-step refinement known as system architecture, CLM schemas, and CLM objects, all architectural components, their relations and implementations can be obtained systematically.

Example 4.22 In RTPA, similar architectural schemas, as well as repetitive actions or processes, can be specified by using the *big-R* notation. The RTPA specification of the architectures of request buttons of an elevator dispatching system, Buttons**ST**, is formally specified in RTPA as shown in Fig. 4.5. The specification indicates that there are 30 similar request buttons in the system denoted by Key(1) through Key (30) that share the same architectural model of ‘Buttons**ST**’.

$$
\begin{array}{l}
\text{Buttons}\mathbf{ST} \triangleq \mathop{\mathit{R}}\limits_{i=1}^{30} (\ \text{Key}\ (i\mathbf{N}): \\
\qquad\qquad <\text{PortAddress} : \mathbf{H} \mid \text{FF00H}\mathbf{H} \le \text{PortAddress}\mathbf{H} \le \text{FF09H}\mathbf{H}>, \\
\qquad\qquad <\text{KeyInput} :\ \mathbf{B} \mid \text{KeyInput}\mathbf{B} = <\text{xxxx xkkk}\mathbf{B}>, \\
\qquad\qquad <\text{Direction} : \mathbf{BL} \mid \mathbf{T} = \text{Up} \wedge \mathbf{F} = \text{Down}>, \\
\qquad\qquad <\text{KeyPosition} : \mathbf{N} \mid 1 \le \text{KeyPosition}\mathbf{N} \le 6> \\
\qquad\quad)
\end{array}
$$

Figure 4.5 The component architectural schema of buttons in an elevator system

As shown in Fig. 4.5, for each Buttons**ST**, it consists of the following four fields: (a) A PortAddress**N** with specific values; (b) A key input information KeyInput**B** with the last three bites effective; (c) A direction**BL** indicating moving direction up or down; and (d) A KeyPosition**N** showing the floor level of the buttons.

The architectural specifications developed in this subsection provide a set of abstract object models and clear interfaces between system hardware and software. After reaching this point, the co-design of a software system, particularly a real-time system, can be separately carried out by hardware and software teams in parallel.

It is recognized that system *architecture specification* by the means of CLMs is the most fundamental and the most difficult part in software design and modeling. However, conventional formal methods hardly provide any support for this purpose. RTPA provides a set of expressive notations for specifying system architectural structures and control models including hardware, software, and their interactions.

On the basis of the system architecture specification and with the work products of system architectural components (CLMs), the operational components of the given system and their behaviors can be specified easily by experienced analysts or programmers as discussed in the following subsections.

4.7.3 SYSTEM STATIC BEHAVIOR MODELING AND REFINEMENT

Static behaviors of software systems are those determinable at compile-time, which can be modeled by a set of processes and their relations. This subsection describes how RTPA is used to formulate detailed process specifications based on the CLM architectures specified in Section 4.7.2.

4.7.3.1 System Static Behaviors

Definition 4.110 The specification of *system static behaviors* is the high-level configuration of processes of a system and their relations. Its general form at the top-level can be represented by a set of parallel processes, i.e.:

$$
\begin{aligned}
\text{SysID}\mathbf{ST}.\text{StaticBehaviors} \triangleq\ & \text{SysInitial} \\
& \|\ \text{Process}_1 \\
& \|\ \text{Process}_2 \\
& \|\ \ldots \\
& \|\ \text{Process}_n
\end{aligned}
\qquad (4.131)
$$

4.7.3.2 Process Schemas

As a result of the first-step refinement in the previous subsection, system static behaviors have been described as a set of process names and their relations. The second-step refinement of system static behaviors in RTPA is to specify the schemas of these identified processes as defined in Fig. 4.2.

Definition 4.111 *A process schema* is the structure of a process that identifies the process by a process number PN**N** and a process name ProcessID**S,** lists operated CLMs and relations with other processes, and describes brief functions of the process, as follows:

$$
\begin{aligned}
\text{ProcessSchema}\mathbf{ST} \triangleq\ & \text{PN}\mathbf{N} \\
& \|\ \text{ProcessID}\mathbf{S}\ (\{\mathbf{I}\};\ \{\mathbf{O}\}) \\
& \|\ \text{FunctionDescription}\mathbf{S} \\
& \|\ \{\text{OperatedCLMs}\} \\
& \|\ \{\text{RelatedProcesses}\} \\
& \|\ \text{FunctionDescription}\mathbf{S}
\end{aligned}
\qquad (4.132)
$$

where FunctionDescription**S** is a brief description of major functions of a process, which will be used to guide further refinement of the process.

The process schema provides further detailed information on each process' functionality, I/O, and its relationships with system architectural components (CLMs) and other processes.

4.7.3.3 Process Implementation

The final refinement step of component static behaviors is to extend the process schemas as specified in Section 4.7.3.2 into detailed processes. This level of specification for system static behaviors is called process implementation.

Definition 4.112 *Process implementation* is the final-step refinement of static behaviors of a system that extends a process schema to a detailed process by using meta processes, process relations, and related CLMs provided in RTPA, i.e.:

$$\begin{aligned} \text{ProcessImplementation}\mathbf{ST} \triangleq\ & \text{ProcessSchemas}\mathbf{ST} \\ & \| \ \text{ProcessInstID}\mathbf{S}\text{S} \\ & \| \ \{\text{DetailedProcesses}\} \end{aligned} \qquad (4.133)$$

Based on the refined specifications, code can be derived seamlessly and rigorously, and tests of the code can be generated prior to the coding phase.

4.7.4 SYSTEM DYNAMIC BEHAVIOR MODELING AND REFINEMENT

Dynamic behaviors of software systems are processes determinable at run-time. According to the RTPA system specification and refinement scheme as shown in Fig. 4.2, the work products developed in Section 4.7.3, the specifications of system static behaviors in term of a set of processes, are only static functional components of the system. To run the components as a live and interacting system, its dynamic behaviors, in terms of *process deployment* and *process dispatch* of all predefined static processes, are yet to be specified in the following subsections.

4.7.4.1 System Dynamic Behaviors

Definition 4.113 The specification of *system dynamic behaviors* at the top level can be generically modeled by the allocation of timing relationships of all static processes contained in the preceding phase, i.e.:

$$
\begin{array}{rl}
\text{SysID}\mathbf{ST}\text{.DynamicBehaviors} \triangleq & \{\text{Base-level processes}\} \\
& \| \; \{\text{High-level processes}\} \\
& \| \; \{\text{Low-interrupt-level processes}\} \\
& \| \; \{\text{High-interrupt-level processes}\} \quad (4.134)
\end{array}
$$

where one or more priority levels may be added or omitted for a specific system.

The four typical priority levels of processes in dynamic behavior specification for a real-time system can be defined as shown in Table 4.11 in an increased priority. For a nonreal-time system, such as a transaction processing and database system, only base level processes may be modeled.

Table 4.11
Priority Levels of Processes in Dynamic Behavior Specification

No.	Priority level	Definition	Execution Priority
1	*Base*	A process that has no strict execution priority at run-time.	All base-level processes of a system are dispatched in the lowest priority when there are no higher level processes scheduled or interrupt events occurred.
2	*High*	A process that has some timing requirements for execution priority at run-time.	A high-level process may take over the run-time resources of a base-level process in system dispatching.
3	*Low interrupt*	An interrupt-event-driven process that has strict execution priority at run-time.	A low-interrupt-level process may take over the run-time resources of an ordinary base-level or high-level process in system dispatching.
4	*High interrupt*	An interrupt-event-driven process that has extremely strict execution priority at run-time.	A high-interrupt-level process may take over the run-time resources of all other type processes in system dispatching.

4.7.4.2 Dynamic Behaviors Deployment

Definition 4.114 *Process deployment* is a set of detailed dynamic process relations at run-time, which refines system dynamic behaviors by specifying precise *time-driven relations* between the system clock, system interrupt sources, and processes at different priority levels as follows:

$$
\begin{array}{ll}
\text{ProcessDeployment} \triangleq \S \rightarrow & \\
\quad (\;\; @\text{BaseTimeEvent}\mathbf{S} & \hookrightarrow \{\text{ProcessSet}_1\} \\
\quad \;\; \mid @\text{HighLevelTimeEvent}\mathbf{S} & \hookrightarrow \{\text{ProcessSet}_2\} \\
\quad \;\; \mid @\text{LowIntTimeEvent}\mathbf{S} & \hookrightarrow \{\text{ProcessSet}_3\} \\
\quad \;\; \mid @\text{HighIntTimeEvent}\mathbf{S} & \hookrightarrow \{\text{ProcessSet}_4\} \\
\quad) & \\
\quad \rightarrow \S &
\end{array}
\tag{4.135}
$$

where § denotes the system.

Process deployment specifies *time-driven* process relations at run-time, where precise timing relationships between different priority levels are specified. Process deployment may be refined next by process dispatching structures.

4.7.4.3 Dynamic Behaviors Dispatch

Dynamic behavior dispatch is the most refined dynamic process relations of a system at run-time.

Definition 4.115 *Process dispatch* is detailed dynamic process relations at run-time, which refines system dynamic behaviors by specifying *event-driven relations* as follows:

$$
\begin{array}{l}
\text{ProcessDispatch} \triangleq \S \rightarrow \\
\quad (\;\; @\text{Event}_1\mathbf{S} \hookrightarrow \{\text{ProcessSet}_1\} \\
\quad \;\; \mid @\text{Event}_2\mathbf{S} \hookrightarrow \{\text{ProcessSet}_2\} \\
\quad \;\; \mid \; \ldots \\
\quad \;\; \mid @\text{Event}_n\mathbf{S} \hookrightarrow \{\text{ProcessSet}_n\} \\
\quad) \\
\quad \rightarrow \S
\end{array}
\tag{4.136}
$$

Process dispatch specifies *event-driven* process relations at run-time, where precise process dispatching strategies are specified for each external or internal event.

The 14th Law of Software Engineering

Theorem 4.9 The *essential facets of software system modeling* state that software systems can be formally specified by its *architectures, static behaviors,* and *dynamic behaviors* with multiple-level refinements.

RTPA adopts only 17 meta processes and 17 process relations to describe software system behaviors in a stepwise refinement approach. Experimental case studies demonstrate that both human and software behaviors can be sufficiently described by RTPA [Wang, 2003c/07h/07i; Wang and Gafurov, 2003; Wang and Ngolah, 2002/03; Wang and Zhang, 2003; Wang and Huang, 2005; Wang and Ruhe, 2007; Tan and Wang, 2003; Adewumi and Wang, 2004; Vu and Wang, 2004; Chiew and Wang, 2004]. Specification and modeling of actions and behaviors are a core part of computing requirement that can be explicitly and precisely described by RTPA, which has been developed as an expressive, easy-to-comprehend, and language-independent notation system, and a specification and refinement methodology for software system modeling and specifications.

This section has demonstrated that a software system, including its architecture, static behaviors, and dynamic behaviors, can be formally described and seamlessly refined by RTPA. Because of the equivalence between software and human behaviors, RTPA can also be used for describing human dynamic behaviors as a series of actions and cognitive processes [Wang, 2007h/07i; Wang and Gafurov, 2003; Wang and Ruhe, 2007].

4.8 RTPA: Notations for Software Engineering

As explained in Sections 4.5 through 4.7, RTPA is a neat and powerful denotational mathematics structure, which is capable to be used as a generic software engineering notation system. RTPA will be adopted throughout the remainder of this book as a descriptive and expressive means for system architectures and behaviors description and specification.

The preceding section has presented the generic methodology of RTPA for software system description and refinement. This section describes the usage of RTPA as software engineering notations based on case studies on real-world software engineering problems at both component and system levels.

4.8.1 MODELING COMPONENT-LEVEL PROBLEMS USING RTPA

ADTs are perfect software architectures at component level that can be used to explain the modeling methodology of component architectures and behaviors in RTPA. An ADT is a logical model of a complex and/or user defined data type with a set of predefined operations. A queue as a typical ADT is presented in this subsection to demonstrate how the RTPA notation and methodology are used to model and specify the architecture, static behaviors, and dynamic behaviors of software components.

4.8.1.1 Existing Approaches to ADT Specification

There are a number of approaches to the specification of ADTs. Mathematically, the main approaches are logical and algebraic, as well as their combinations. Although each of these approaches has its advantages, there are gaps when applying them to solve real-time specification problems.

The logic approach is good at specifying the properties of ADT operations, usually in forms of the preconditions and post-conditions of operations. Due to the nature of logic, the logical approach is the easiest one to model the behaviors of ADTs, particularly their static behaviors. In contrary, the algebraic approach is good at describing dynamic and run-time behaviors of ADTs in an abstract, elegant, and dynamic manner.

Example 4.23 A specification of the ADT, *Queue*, in the logic-based approach is shown in Fig. 4.6 [Stubbs and Webre, 1985].

Elements: e, f, g, … of type stdelement.
Structure: Queue Q = {<e, t_e>, <f, t_f>, <g, t_g>, …},
where t_x is the time of insertion of
<x, t_x> into Q (<x, t_x>, <y, t_y> ∈ S | x <> y → t_x <> t_y)
Domain: 0 <= Cardinality(Q) <= maxsize.
Operations:
enqueue(e: stdelement)
pre – ∃Q ∧ Cardinality(Q) <> maxsize.
post – Q = Q’ ∪ {<e, t_e>}.
serve(var e: stdelement)
pre – ∃Q ∧ Q <> {}.
post – Q = Q’ - {<e, t_e>}|(∀<x, t_x> ∈ Q, t_e < t_x).
empty: boolean
pre – ∃Q.
post – empty = (Q = {}).
full: boolean
pre – ∃Q.
post – full = Cardinality(Q) = maxsize.
clear
pre – ∃Q.
post – Q = {}.
create
pre – true.
post – ∃Q ∧ Q = {}.

Figure 4.6 Specification of an ADT model of Queue in predicate logic

Note in the above example that the data structures are usually informally described in the logic-based approach. Also, there is actually nothing that has been specified for how the Queue is created, and what are the operations between the pre- and post-conditions.

A queue in RTPA is modeled as an algebraic entity, which has predefined operations on the architectural model of the Queue. Unlike the conventional approaches to ADT specifications that treat ADTs as static data types, ADTs in RTPA are treated as dynamic finite state machines to serve as both structural and operational components in system design and modeling [Tan and Wang, 2003].

4.8.1.2 Architectural Specification in RTPA

At the top level, an RTPA specification of the queue, *Queue*$\mathbf{ST}$, has three parallel facets, which are the Queue's architecture, static behaviors, and dynamic behaviors as shown below.

$$\begin{aligned} \text{Queue}\mathbf{ST} \triangleq\ & \text{Queue}\mathbf{ST}.\text{Architecture} \\ & \| \ \text{Queue}\mathbf{ST}.\text{StaticBehaviors} \\ & \| \ \text{Queue}\mathbf{ST}.\text{DynamicBehaviors} \end{aligned} \tag{4.137}$$

Then, Queue$\mathbf{ST}$ can be broken up and be further refined by detailed specifications according to the RTPA specification and refinement method.

Example 4.24 The architecture of the Queue$\mathbf{ST}$ is specified by RTPA as shown in Fig. 4.7, where both the architectural CLM and an access model are provided for the Queue.

$$\begin{aligned} \textbf{Queue}\,\mathbf{ST}\textbf{.Architecture} \triangleq\ & \text{CLM} : \mathbf{ST} \\ & \| \ \text{AccessModel} : \mathbf{ST} \\ & \| \ \text{Events} : \mathbf{S} \\ & \| \ \text{Status} : \mathbf{BL} \end{aligned}$$

$$\begin{aligned} & \textbf{Queue.}\,\mathbf{ST}\ \textbf{Architecture.CLM} \triangleq \text{QueueID}\mathbf{S} :: \\ & \qquad (\ <\text{Size} : \mathbf{N} \mid \text{Size}\mathbf{N} \geq 0>, \\ & \qquad\ \ <\text{Element} : \mathbf{RT}>, \\ & \qquad\ \ <\text{CurrentPos} : \mathbf{P} \mid 0 \leq \ \text{CurrentPos}\mathbf{P} \leq \ \text{Size}\mathbf{N}\text{-}1> \\ & \qquad) \end{aligned}$$

$$\begin{aligned} & \textbf{Queue}\,\mathbf{ST}\textbf{.Architecture.AccessModel} \triangleq \\ & \qquad\qquad \text{QueueID}\mathbf{S}(\text{CurrentPos}\mathbf{P})\mathbf{RT} \end{aligned}$$

Figure 4.7 The architectural model of the Queue specified in RTPA

In Fig. 4.7 the *access model* of Queue$\mathbf{ST}$ is a logic model for supporting external invocation of the Queue in operations, such as *enqueue* and *service*. The other parts of the model are designed for internal manipulation of the Queue, such as creation, memory allocation, and release.

4.8.1.3 Static Behavior Specification in RTPA

Component static behaviors in RTPA are valid operations of system that can be determined at compile-time, which describe the configuration of processes of the component and their relations. The schemas of a set of static behaviors of Queue**ST**, known as *create, release, enqueue, serve, clear, empty test, full test,* are *modeled* as given in the following example.

Example 4.25 The detailed specification of one of the Queue's static behaviors, Queue**ST**.serve, is given below.

```
QueueST.Serve (<I ::  QueueInstS>;
                          <O :: ⓢQueueID.ServedBL, ElementRT>)  ≙
{
  QueueIDS := QueueInstS
  → ( � ⓢQueueExistBL = T  ∧ CurrentPosP  > 0
          → (QueueID(1))RT >  Element RT
          → QueueID(i))RT >  QueueID(i-1)RT
          → ↓ (QueueID.CurrentPosP)
          → ⓢQueueID.ServedBL := T
      | � ~
          → ⓢQueueID.ServedBL := F
          → ! (@'QueueIDExistBL = F  ∨  QueueEmptyBL = T')
     )
}
```

Figure 4.8 The static behavioral model of the Queue specified in RTPA

Contrasting the static behavior model of Queue**ST**.serve in RTPA as shown in Fig. 4.8 and that of predicate logic as shown in Fig. 4.6, advances of the RTPA method and notations are well demonstrated. Among them, the most important advantage is that a system model in RTPA can be seamlessly refined into code in a programming language in the succeeding phases of software engineering.

4.8.1.4 Dynamic Behavior Specification in RTPA

Component dynamic behaviors in RTPA are process relations that may be determined at run-time. According to the RTPA system specification and refinement scheme, the specifications of system static behaviors are only functional components of the system. To incorporate the components into a live and interacting system, the dynamic behaviors of the system in terms of process deployment and dispatch are yet to be specified.

Example 4.26 The dynamic behaviors of Queue**ST** are specified in RTPA as shown in Fig. 4.9, where the process dispatch mechanisms of the Queue specifies detailed dynamic process relations at run-time by a set of *event-driven* relations.

```
QueueST.DynamicBehaviors ≙ { § →
(  @CreateQueueS  ↳  Queue.Create (<I:: QueueInstS, ElementInstRT, SizeInstN>;
                     <O:: ⓢQueueID.AllocatedBL, ⓢQueueID.ExistBL>)
 | @ReleaseQueueS ↳  Queue.Release (<I:: QueueInstS>;
                                   <O:: ⓢQueueID.ReleasedBL>)
 | @EnqueueS      ↳  Queue.Enqueue  (<I:: QueueInstS, ElementInstRT>;
                                     <O:: ⓢQueueID.EnqueuedBL>)
 | @ServeS        ↳  Queue.Serve  (<I:: QueueInstS>;
                                  <O:: ⓢQueueID.ServedBL, ElementRT>)
 | @ClearS        ↳  Queue.Clear  (<I:: QueueInstS>;
                                  <O:: ⓢQueueID.ClearedBL>)
 | @QueueEmptyS   ↳  Queue.EmptyTest  (<I:: QueueInstS>;
                                      <O:: ⓢQueueID.FullBL>)
 | @QueueFullS    ↳  Queue.FullTest (<I:: QueueInstS>;
                                   <O:: ⓢQueueID.FullBL>)
 ) → §
}
```

Figure 4.9 The dynamic behavioral model of the Queue specified in RTPA

Figs. 4.7 through 4.9 model an ADT, *Queue***ST**, in a coherent system from three perspectives. With the RTPA specification and refinement method and the expressive power of RTPA notation system, the features of ADTs as both static data types and dynamic system components can be specified rigorously and precisely.

4.8.2 MODELING SYSTEM-LEVEL PROBLEMS USING RTPA

The same methodology and notations of RTPA for component level specification can be applied in system level modeling and refinement. An Automated Teller Machine (ATM) is taken as a well-known example of safety-critical and real-time systems for demonstrating the methodology of RTPA. This subsection describes how the architecture, static and dynamic behaviors of the ATM system can be modeled rigorously, precisely, and consistently using RTPA [Wang and Zhang, 2003].

The conceptual model of the ATM system can be described by an FSM, and the formal model of the ATM is specified by RTPA. The formal model of the ATM enables implementation of system models independent of programming languages and operating platforms. It also improves the controllability, reliability, maintainability, and quality of the design and implementation in real-time software engineering.

4.8.2.1 The Conceptual Model of the ATM

The conceptual model of the ATM architecture and behaviors are given in Figs. 4.10 and 4.11, respectively. Fig. 4.10 describes the configuration and logical relationships among components of the ATM.

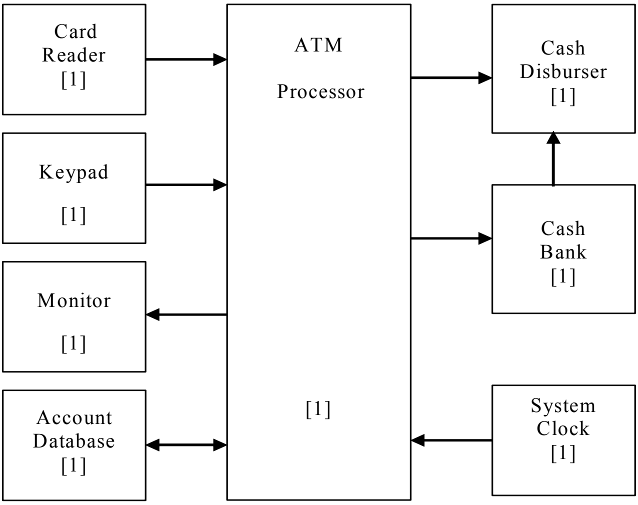


Figure 4.10 The conceptual model of the ATM architecture

A state transition diagram is adopted in Fig. 4.11 to describe the basis behaviors of the ATM system as an FSM.

The following subsections develop a formal specification of the ATM system using RTPA notations and methodology.

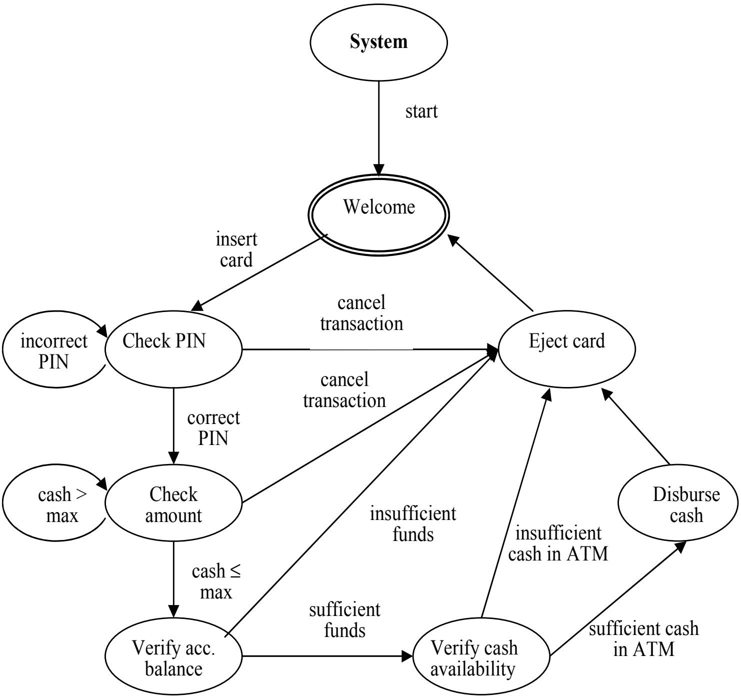


Figure 4.11 The conceptual model of the ATM behaviors

4.8.2.2 Formal Description of the ATM Architectures

According to the RTPA scheme for system specification and refinement, the top-level specification of the ATM system can be described as follows:

$$\begin{aligned} \textbf{§(ATM}\mathbf{ST}\textbf{)} \triangleq\ & \text{ATM}\mathbf{ST}\text{.Architecture} \\ & \|\ \text{ATM}\mathbf{ST}\text{.StaticBehaviors} \\ & \|\ \text{ATM}\mathbf{ST}\text{.DynamicBehaviors} \end{aligned} \tag{4.138}$$

The high-level specification of the architecture of ATM as a set of CLMs is shown in Fig.4.12, where the number in the angular brackets, [n**N**], specifies the required number of instance objects for a CLM in the system architectural configuration.

```
ATMST.Architecture ≙ <ATMProcessor : ST | [1]>
                     || <SystemClock : ST | [1]>
                     || <CardReader : ST | [1]>
                     || <Keypad : ST | [1]>
                     || <Monitor : ST | [1]>
                     || <AccountDatabase : ST | [1]>
                     || <CashBank : ST | [1]>
                     || <CashDisburser : ST | [1]>
                     || <Events : S>
                     || <Status : BL>
```

Figure 4.12 The architecture of the ATM system

For example, the RTPA specification of the architectural model of the *CardReader***ST** in the ATM is further refined as given in Fig. 4.13, where the statuses, ports of interfaces, and the data format of the card reader are formally specified. A complete description of the ATM architectural models may be referred to Appendix K.

```
CardReaderST ≙ CardReaderS ::
               (<Data : N | 0 ≤ DataN ≤ 1000000>,
                <Status : BL | T = CardInserted ∧ F = NoCard>,
                <CardEjectStatus : BL | T = Ejected ∧ F = NoAction>,
                <CardReaderPort : B | CardReaderPortB = FFF1H >
               )
```

Figure 4.13 The architecture of the ATM CardReader**ST**

4.8.2.3 Formal Description of the ATM Static Behaviors

System static behaviors model the high-level configuration of all processes of a system and their relations. This subsection describes how RTPA can be used to formulate detailed process specifications of the ATM system based on the CLM architectures obtained in the architectural modeling.

The ATM system encompasses eight static behaviors, such as the system, welcome, check PIN, check amount, verify account balance, verify cash availability, disburse cash, and eject card. Each of these static behaviors can be described as a process in the ATM. For instance, the *welcome* processes can be specified as shown in Fig. 4.14. A complete description of the ATM process models is provided in Appendix K.

Based on the detailed specifications of system components as a set of processes, program code can be derived easily and rigorously based on them, and test cases for the code can also be generated prior to the program being implemented.

```
Welcome (<I::( )>; <O::( )>) ≙
{// State 1: Welcome
 // Operated CLMs :: {stdCardSlotST }
 // Related processes :: {CheckPIN}
          T
          R          ( � Monitor.StatusBL = T
 CardInsertedBL= F
              → 'Welcome!' |<  PORT(MonitorPortP)S
              → 'Please insert your card.' |< PORT(MonitorPortP)S
              → CardReaderST.StatusBL |> CardInsertedBL
          | � ~
              → ! (ⓢMonitorFaultBL = T)
              → StateN := 8                   // System fault
           )
CardReaderST.DataN  |> AccountNumN
→ ( � SysDatabaseST(AccountNumN).StatusBL := T
        → PINEnterTimesN := 10000       // 10s
        → ProcessStateN := 2
    | � ~
        → 'Invalid Card.' |< PORT(MonitorPortP^)S
        → EjectCardBL := T
        → EjectCardBL  |> CardReaderST.CardEjectStatusBL
        → ProcessStateN := 1
}
```

Figure 4.14 The Welcome process of the ATM system

4.8.2.4 Formal Description of the ATM Dynamic Behaviors

Generally, system dynamic behaviors are the timing relationships between the static processes of a system. The dynamic behaviors of the ATM system can be specified by a number of execution priority levels of processes based on their real-time timing requirements, which are the base-level and high-interrupt-level processes.

Process deployment is defined as dynamic process relations at run-time, which refines system dynamic behaviors by specifying precise and explicit *time-driven relations* between system clock, system interrupt sources, and processes at different priority levels. For example, the ATM dynamic behaviors are shown in Fig. 4.15, where precise timing relationships between different priority levels are specified.

```
**ATM.ProcessDeployment** ≜
{ // basic level processes
 @ System ↳ ( SysInitial
              → $\overset{\mathbf{T}}{\underset{SysShutDown\mathbf{BL}=\mathbf{F}}{R}}$ ATMProcessDispatching
              → ⊠
            )
|| // High-interrupt level processes
 ⊙ @SysClock1msInt
     ↗ ( SysClock
         ↳ CardScanning
       )
     ↘ ⊙
}
```

Figure 4.15 The ATM process deployment process

Process dispatch describes another aspect of system dynamic behaviors at run-time by specifying the *event-driven relationships* of the given system. For instance, the specification of ATM process dispatch is shown in Fig. 4.16.

```
**ATMProcessDispatch** ≜
{( �ⓢATMID.Running**BL** = **T**
    → ( @ATM.State**N** = 0  // Idle
           → ∅
        | @ATM.State**N** = 1
           ↳ Welcome (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 2
           ↳ CheckPIN (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 3
           ↳ CheckAmount (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 4
           ↳ VerifyAccountBalance (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 5
           ↳ VerifyCashAvailability (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 6
           ↳ DisburseCash (<**I::** (PN**N**)>; <**O::** ( )>)
        | @ATM.State**N** = 7
           ↳ EjectCard (<**I::** (PN**N**)>; <**O::** ( )>))
}
```

Figure 4.16 The ATM process dispatch process

This section has demonstrated that the ATM system, including its architecture, and static and dynamic behaviors, can be essentially and sufficiently described by RTPA. The case studies on real-world problems have shown that the formal specification and modeling of the ATM system are helpful for improving safety operations and quality services of the system. Other related case studies on RTPA for formally modeling real-time systems may be referred to [Wang, 2003c; Wang and Ngolah, 2002/03; Wang and Huang, 2005; Adewumi and Wang, 2004; Vu and Wang, 2004]. A complete specification of the ATM system is provided in Appendix K of this book.

RTPA is not only useful as a generic notation and methodology for software engineering, but also good at modeling human cognitive processes. The applications of RTPA in modeling cognitive processes of the brain and natural intelligence may be referred to [Wang, 2007h/07i; Wang and Gafurov, 2003; Wang and Ruhe, 2007; Chiew and Wang, 2004].

RTPA has been developed as an algebra-based, expressive, easy-to-comprehend, and language-independent notation system, and a practical specification and refinement methodology for software engineering. RTPA is capable to support top-down software system design and implementation by algebraic modeling and seamless refinement methodologies. The RTPA methodology covers the entire software engineering processes from system modeling to code generation in a coherent algebraic notation.

A number of case studies on large-scale software system modeling and specifications have been carried out, such as the Telephone Switching System (TSS) [2002a], the Lift Dispatching System (LDS) [Wang and Ngolah, 2002], the Automated Teller Machine (ATM) [Wang and Zhang, 2003], and a set of ADTs [Tan and Wang, 2003]. RTPA has also been used to specify algorithms and software process models such as CMM.

Experiences show that the RTPA notation system and methodology have the following advantages:

- Easy to learn and acquisition
- Easy to comprehend
- Suitable for specifying the 3-D real-time system behaviors
- Suitable for specifying both architectural and operational components in a system
- Expressive for both system architectures and behaviors
- Expressive for real-time events and timing manipulations
- Strongly and strictly typed with a type suffix system
- Built-in exceptional detection mechanisms for safety-critical applications

A set of support tools for RTPA has been developed [Tan and Wang, 2006; Tan, Wang, and Ngolah, 2004a/04b/05/06; Ngolah, Wang, and Tan, 2005b/06], which encompasses the RTPA parser, type checker, and code generator in C++ and Java. The RTPA code generator enables system specifications in RTPA to be automatically translated into fully executable code. The RTPA tools will support system architects, analysts, and practitioners for developing consistent and correct specifications and architectural models of large-scale real-time and distributed systems, and the automatic generation of code based on the rigorous specifications in the descriptive mathematical notations.

RTPA is characterized as the least complete set of algebraic notations and well structured stepwise method for software system specification and refinement. The application results encouragingly demonstrated that RTPA is a powerful and practical software engineering notation system and methodology for both academics and practitioners in software engineering.

4.9 Summary

Mathematics deals with statements about abstract *objects* and *relations* between them. The entire **theory of software engineering** is about mathematical models and formal treatment of software architectures, behaviors, and software engineering processes, which are centered by denotational mathematics and formal inference means.

This chapter has explored essential mathematical means for modeling software architectures and behaviors. Existing mathematical means, such as sets*, functions, relations, mathematical logic*, and their applications in software engineering, has been reviewed. The investigation has been continued on what are the **essential elements of mathematical needs** for modeling software systems and software engineering processes, which leads to the findings of the inadequacy of conventional **analytic mathematics**, and the requirement for a **denotational mathematics** in software engineering. **Real-Time Process Algebra** (RTPA) has thus been introduced as an expressive mathematics and practical notation system for a **rigorous treatment** of software system architectures, static behaviours, and dynamic behaviours. The methodology of RTPA for software system description and refinement has been presented, and applications of RTPA as a powerful and generic notation system for software engineering have been described. As a result, the **mathematical foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Mathematical Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge architecture as summarized below.

Chapter 4. Mathematical Foundations of SE

■ Fundamental Mathematics

- Set theory
- Relations
- Functions
- Propositional logic
- Predicate logic
- Algebraic systems

■ Denotational Mathematics for Software Engineering

- Fundamental elements in modeling software systems
- The need for denotational mathematics in software engineering
- The big-R notation

■ Real-Time Process Algebra (RTPA)

- The process metaphor of software systems
- The structure of RTPA
- The type system of RTPA
- Meta processes of RTPA
- Process relations of RTPA

■ The RTPA Methodology for Software System Modeling and Refinement

- The RTPA methodology
- System architecture modeling and refinement
  - The system architecture
  - The CLM schema
  - The CLM objects
- System static behavior modeling and refinement
  - System static behaviors
  - Process schemas
  - Process implementation
- System dynamic behavior modeling and refinement
  - System dynamic behaviors

- Dynamic behaviors deployment
- Dynamic behaviors dispatch

■ RTPA: Notations for Software Engineering

- Modeling Component-Level Problems using RTPA
  - Existing approaches to ADT specification
  - Architectural specification in RTPA
  - Static behavior specification in RTPA
  - Dynamic behavior specification in RTPA
- Modeling System-Level Problems using RTPA
  - The conceptual model of the ATM system
  - Formal description of the ATM architectures
  - Formal description of the ATM static behaviors
  - Formal description of the ATM dynamic behaviors

SIGNIFICANT FINDINGS OF THIS CHAPTER

• Mathematics, as well as philosophy, is the **top level abstraction means** and therefore the **most general human knowledge**.

• The **utility of mathematics** in software engineering states that denotational mathematics is the means and rules to rigorously and explicitly express design notions and conceptual models on abstract architectures and complex interactive behaviors at the highest level of abstraction and in the largest scope of systems.

• Conventional **analytic mathematics** and propositional logic are inadequate in dealing with software engineering problems. This finding reveals a profoundly overlooked problem in software engineering, i.e., the formal means and tools were inadequate, and there was a lack of a **denotational mathematics** for software engineering.

• A fundamental view towards the description and modeling of human and system behaviors is that there are essentially three categories of **descriptivity**: to *be*, to *have*, and to *do*. All mathematical means and forms, in general, are an abstract description of these three categories of human and system behaviors. That is, mathematical logic is the abstract means for describing “to be,” set theory for describing “to have,” and algebras for describing “to do.”

• The **behavior space** Ω of software is 3-dimensional, which can be described by a Cartesian product of operations OP, time T, and memory space S, i.e.: $\Omega = OP \times T \times S$. Mathematics, theories, methodologies, and tools for software engineering must be designed to adequately deal with such 3-D problems.

• The **big-R notation** models and unifies a fundamental and widely applied mathematical concept in computing and human behavior description, i.e., iterations and recursions. It demonstrates that a convenient mathematical notation may dramatically reduce the difficulty and complexity in expressing a frequently used and highly recurring concept and notion in computing.

• **Software system behaviors** can be described as the composition of a list of interacting **processes**. The **top-level behaviors** of software systems are **process dispatches** known as the *event-, time-,* and *interrupt-driven* dispatching mechanisms.

• **Software system architectures** can be described as a set of **Component Logical Models** (CLMs).

• The **evaluation** (or quantification) $\text{�}_{\mathbb{T}}$ is a fundamental computing operation that maps a given expression in type $\mathbb{T}$ into a value in the same type. When the type $\mathbb{T} = \{\mathbf{BL}, \mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{B}\}$, the evaluations are called a *Boolean*, *ordinal*, or *numerical* evaluation (for types $\mathbf{Z}, \mathbf{R}, \mathbf{B}$), respectively, i.e.:

$$\text{�}_{\mathrm{BL}}(exp\mathbf{BL})\mathbf{BL} = \text{�}_{\mathrm{BL}}: exp\mathbf{BL} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

$$\text{�}_{\mathrm{N}}(exp\mathbf{N})\mathbf{N} = \text{�}_{\mathrm{N}}: exp\mathbf{N} \rightarrow \mathbf{N}$$

$$\text{�}_{\mathrm{Z}}(exp\mathbf{Z})\mathbf{Z} = \text{�}_{\mathrm{Z}}: exp\mathbf{Z} \rightarrow \mathbf{Z}$$

$$\text{�}_{\mathrm{R}}(exp\mathbf{R})\mathbf{R} = \text{�}_{\mathrm{R}}: exp\mathbf{R} \rightarrow \mathbf{R}$$

$$\text{�}_{\mathrm{B}}(exp\mathbf{B})\mathbf{B} = \text{�}_{\mathrm{B}}: exp\mathbf{B} \rightarrow \mathbf{B}$$

• The **addressing** $\Rightarrow$ is a fundamental computing operation that maps a given logical $id\mathbb{T}$ into a block of the physical memory denoted by $ptr\text{Þ}\mathbf{P}$ accommodating n bytes of memory for the variable in type $\mathbb{T}$, i.e.:

$$\begin{aligned} id\mathbb{T} &\Rightarrow \mathrm{MEM}[ptr\text{Þ}\mathbf{P}]\mathbb{T} \\ &\quad \Leftrightarrow (\pi: id\mathbb{T} \rightarrow ptr\text{Þ}\mathbf{P} \\ &\qquad\quad \rightarrow id\mathbb{T} = \mathrm{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T} \\ &\qquad\ ) \end{aligned}$$

- **Memory allocation** $\Leftarrow$ is a fundamental computing operation that collects a unique memory block logically named $id\mathbb{T}$ and physically located by $ptr\text{Þ}\mathbf{P}$ accommodating n bytes of memory for the variable in type $\mathbb{T}$, i.e.:

$$\begin{aligned}
id\mathbb{T} &\Leftarrow \text{MEM}[ptr\text{Þ}\mathbf{P}]\mathbb{T} \\
&\Leftrightarrow (\pi^{-1}\text{: } ptr\text{Þ}\mathbf{P} \rightarrow id\mathbb{T} \\
&\qquad \rightarrow id\mathbb{T} = \text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n\text{-}1]\mathbb{T} \\
&\quad)
\end{aligned}$$

- **Memory release** $\nLeftarrow$ is a fundamental computing operation that dissociates and frees a unique block of n continuous physical memory elements denoted by $ptr\text{Þ}\mathbf{P}$ from its logical identifier $id\mathbb{T}$, i.e.:

$$\begin{aligned}
&id\mathbb{T} \nLeftarrow \text{MEM}[\perp]\mathbb{T} \\
\Leftrightarrow &(\pi\text{: } id\mathbb{T} \rightarrow ptr\text{Þ}\mathbf{P} \\
&\quad \rightarrow \text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n\text{-}1]\mathbb{T} := \perp \\
&\quad \rightarrow ptr\mathbf{P} := \perp \\
&\quad \rightarrow id\mathbb{T} := \perp \\
&\quad)
\end{aligned}$$

- The **mathematical model of a process** P is a composed component of n meta statements p_i and p_j, $1 \leq i < n$, $j = i + 1$, according to certain composing relations r_{ij}, i.e.:

$$\begin{aligned}
P &= \mathop{R}_{i=1}^{n-1} (p_i \ r_{ij} \ p_j), j = i+1 \\
&= (...(((p_1) \ r_{12} \ p_2) \ r_{23} \ p_3) \ ... \ r_{n-1,n} \ p_n)
\end{aligned}$$

where r_{ij} is one of the 17 **cumulated relations** or composing rules identified in RTPA.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Fundamental Mathematics

- Abstraction and categorization of objects are basic human cognitive processes. Set is the mathematical model of these cognitive processes.

- A **set** is a collection of elements with a common property. Set is a fundamental and powerful mathematical concept for abstracting and eliciting objects that share certain common properties. **Abstraction** is an elicitation of common properties of elements from a given set.

- The **basic set operations** are *union, intersection, difference,* and cardinal size. **Derived set operations** are *complement, symmetric difference, Cartesian product*, and *partition*.

- Since the position, or the sequential order, of an element in a set has no meaning, the mathematical entities of **pair**, **tuple**, **sequence**, **list**, and **ordered set** are introduced for dealing with the order information of elements in sets.

- **Relation** is the most important concept in programming theories, because a program can be modeled as a finite list of relations between individual statements. Relations also play an important role in explaining internal knowledge representation and the natural intelligence. Relations as mathematical entities can be treated and composed based on algebraic laws.

- A **program** can be treated as a composition of a list of statements by predefined relational or composing rules. The relations between statements are special type relations known as **cumulative relations ®**, where a given relation is related to all previous relations.

- The **composing rules** in programming can be classified into **17 process relations** known as *sequential, branch, switch, iterations* (3), *procedure call, recursion, parallel, concurrence, interleave, pipeline, interrupt, jump,* and *system dispatches* (3).

- **Algebra** is a branch of mathematics in which variables and their relations are represented by abstract symbols and formulae. Using algebra, generic relations between variables and quantities may be formally, precisely, and efficiently described. Rigorous reasoning can then be conducted based on established algebraic rules and properties.

- Function is another important mathematical concept developed in algebra. A **function** is a mapping relation between two sets in a generic signature. Almost all discrete or continuous relations between sets can be described as functions.

- By extending the objects under study beyond sets in algebra, a number of **advanced algebraic systems** are developed, such as *abstract algebra, process algebra, concept algebra,* and *system algebra.*

Denotational Mathematics for Software Engineering

- **Denotational mathematics** is a set of contemporary mathematical structures for dealing with the unique mathematical entities, abstract objects, relations, and formal manipulations in abstract system modeling, which encompasses **concept algebra, system algebra,** and **RTPA**.

- New problems require new forms of mathematics. Conventional science and engineering disciplines have been mainly using **analytic mathematics** in theory development and problem solving. Software engineering needs a **denotational mathematics**, e.g., **RTPA**, which can be used to describe software systems rigorously, explicitly, and expressively. A coherent notation system for software engineering is derived on the basis of RTPA.

- Denotational mathematics deal with the **3-D behavior space** Ω of software system behaviors, i.e.: $\Omega = OP \times T \times S$.

- A generic and fundamental operation in system and human behavioral modeling is the formal description of *repetitive actions* and/or *recurring architectures*. The **big-R notation** is introduced to denote this fundamental requirement in computing and software engineering.

Real-Time Process Algebra (RTPA)

- **Real-Time Process Algebra** (RTPA) is a set of formal notations and algebraic rules for modelling and describing real-time process architectures and behaviours of software systems.

The **structure of RTPA** can be defined as follows:

$$\begin{aligned} \text{RTPA} \triangleq\ & \text{Meta processes} \\ & \|\ \text{Process relations} \\ & \|\ \text{System architectures} \\ & \|\ \text{Primary types} \\ & \|\ \text{Abstract dada types} \\ & \|\ \text{Specification refinement scheme} \end{aligned}$$

- The **RTPA type system** $\mathfrak{T}$ encompasses 17 primitive types as follows:

$$\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @e\mathbf{S}, @t\mathbf{TM}, @int⦿, ⓢs\mathbf{BL}\}$$

RTPA adopts the **type-suffix convention** in which every identifier of variables, constants, and expressions is attached with a type in bold in the format of $id\mathbb{T}$, $\mathbb{T} \in \mathfrak{T}$.

• The set of **RTPA meta processes** $\mathfrak{P}$ encompasses 17 fundamental primitive operations in computing as follows:

$$\mathfrak{P} = \{:=, \text{�}, \Rightarrow, \Leftarrow, \not\Leftarrow, \gtrdot, \lessdot, |\gtrdot, |\lessdot, \underline{\underline{@}}, \triangleq, \uparrow, \downarrow, !, \oslash, \boxtimes, \S\}$$

• The set of **RTPA process relations** $\mathfrak{R}$ encompasses 17 fundamental primitive operations in computing as follows:

$$\mathfrak{R} = \{\rightarrow, \curvearrowright, |, |\ldots|, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \|, \oiint, |||, \gg, \lightning, \hookrightarrow_{t}, \hookrightarrow_{e}, \hookrightarrow_{i}\}$$

The RTPA Methodology for Software System Modeling and Refinement

• RTPA deals with complicated system modeling and specifications via a number of **systematic refinements** in a top-down approach by using a coherent set of notations.

• In RTPA three **fundamental aspects of software systems** can be modeled and specified, i.e.:

$$\begin{aligned} \S(\text{SysID}\mathbf{S}) \triangleq\ & \text{SysID}\mathbf{S}.\text{Architecture} \\ & \|\ \text{SysID}\mathbf{S}.\text{StaticBehaviors} \\ & \|\ \text{SysID}\mathbf{S}.\text{DynamicBehaviors} \end{aligned}$$

• The specification of each of the above subsystems can be implemented by a three-level refinement process at the system, class, and object levels.

• In the RTPA specification and refinement scheme, two key concepts, *CLM* and the *process*, are introduced to model software system architectures and behaviors, respectively. A **CLM** is an abstract model of a system **architectural component** that represents a hardware interface, an internal logical model, and/or a common control structure of a system. A **process** is an abstract model of a unit of software **system behaviors** that represents a transition procedure of the system from one state to another by changing values of its inputs {I}, outputs {O}, and/or internal variables {V}.

- RTPA is not only a mathematical inference means, but also a generic **software engineering notation system**. RTPA applications in system architectural and behavioral modeling and specifications are presented in a number of real-world case studies, which may be used as testing benchmarks for comparatively evaluating the express power of existing formal methods.

Questions and Research Opportunities

4.1 Why is denotational mathematics needed for software engineering determined by the nature of software as well as Theorem 1.3, Theorem 1.4, and the HAMSD model?

4.2 What are the differences between *denotational* and *analytic* mathematics in means and purposes?

4.3 Set is a fundamental and powerful mathematical concept for describing objects that share certain common properties. According to Lemma 4.1, abstraction is an elicitation of common properties of elements from a given set.

According to the HAMSD model of knowledge abstraction as presented in Theorem 1.4, explain why mathematics is the top level abstraction means in system modeling.

4.4 Tuple is a powerful modeling means in mathematics and software engineering for denoting a coherent encapsulation or composition of multiple objects. Try to denote the automobile you've modeled in Ex.1.13 with a tuple and extend the categories of functions defined in it with detailed attributes or properties.

4.5 A *cumulative relation* ® as given in Definition 4.21 is an ordered list of embedded relations. Try to provide an instance of a cumulative relation.

4.6 Try to prove the following logical equivalences using truth tables:

a) $X \Rightarrow Y \Leftrightarrow \neg X \vee Y$
b) $(X \Leftrightarrow Y) \Leftrightarrow (X \wedge Y) \vee (\neg X \wedge \neg Y)$

4.7 According to Definition 4.33, a generic computational *operation* can be defined as an abstract function *op* on the set of operands or data objects *O*. Provide a mathematical definition for the above concept.

4.8 Create a form to compare and contrast the definitions and usages of inference methods in predicate logic, such as *universal instantiation, universal generalization, existential instantiation,* and *existential generalization* as described in Section 4.4.2.3.

4.9 Describe the corresponding relationships between the basic expressiveness of natural languages and the mathematical means in conventional and denotational mathematics.

4.10 Use the *big-R notation* of RTPA to denote the following computational operations:

a) A *while loop* for process *P*
b) A *repeat loop* for process *Q*

4.11 Use the *big-R notation* of RTPA to denote the following architectural models of data objects and system structures:

a) A one dimensional array, *Array***ST**, with 100 integer elements *A*[*i* **N**]**Z**.

b) Ten buttons on an equipment, *Buttons***ST**, that share the same structure, i.e., *Button* (0) ... *Button* (9).

4.12 Use the *MaxFinder* algorithm as given below to explain Theorem 4.3, and identify all statements and relations in the process.

```
$\text{MaxFinder}\ (\{\text{I}{::}\ \text{X}[0]\mathbf{N}, \text{X}[1]\mathbf{N}, \ldots, \text{X}[\text{n}-1]\mathbf{N}\}; \{\text{O}{::}\ \text{max}\mathbf{N}\}) \triangleq$
{
  $\text{Xmax}\mathbf{N} := 0$
  $\to \mathop{R}\limits_{\text{i}\mathbf{N}=0}^{\text{n}\mathbf{N}-1}$ ( � $\text{X}[\text{i}\,\mathbf{N}]\mathbf{N} > \text{Xmax}\mathbf{N}$
              $\to \text{Xmax}\mathbf{N} := \text{X}[\text{i}\,\mathbf{N}]\mathbf{N}$
          )
  $\to \text{max}\mathbf{N} := \text{Xmax}\mathbf{N}$
}
```

4.13 What is the architecture of RTPA as a coherent notation system?

4.14 Briefly describe the RTPA type system $\mathfrak{T}$.

4.15 According to Theorem 4.5, partition the 17 RTPA primitive types into equivalence sets.

4.16 RTPA extends the type rules from variables to constants. What are the advantages of this extension over conventional programming practice in software engineering?

4.17 Briefly describe the set of RTPA meta processes $\mathfrak{P}$.

4.18 Why is *evaluation* (�$_{\mathbb{T}}$) modeled as a fundamental computing process in RTPA?

4.19 Why is *addressing* ($\Rightarrow$, or π: *id*$\mathbb{T}$ → *ptr*Þ**P**) modeled as a fundamental computing process in RTPA?

4.20 Why is *memory allocation* ($\not\Leftarrow$) modeled as a fundamental computing process in RTPA?

4.21 What is the syntactic and semantic differences between *skip* ($\otimes$) as modeled in RTPA and an operation that does nothing?

4.22 Briefly describe the algebraic process operations of RTPA known as the process relations $\mathfrak{R}$.

4.23 Why can computing operations at the system level be modeled as three process dispatching operations known as the *event-, time-,* and *interrupt-driven* process dispatches?

4.24 On the basis of Theorem 4.8, explain why the expressive power of RTPA is much higher than existing programming languages, though it only adopts 17 meta processes and 17 algebraic process operations.

4.25 What are the three subsystems that are generically modelled at the top layer of software systems in RTPA? What are the sequences for specifying and refining these three subsystems in RTPA?

4.26 Briefly summarize the methodology of RTPA in software system modelling and refinement.

4.27 Why has architectural modelling been recognized as the most creative, important, and difficult aspects in system design and implementation?

4.28 Why should architectural models be designed and specified first before the behavioural models of a given system is carried out according to RTPA methodology? What would be wasted in software engineering if a project team goes directly into programming?

4.29 A *Component Logical Model* (CLM) is a powerful modeling technique of RTPA for system architectural component modeling. Explain how CLM may be used to model the following data objects in system architectures:

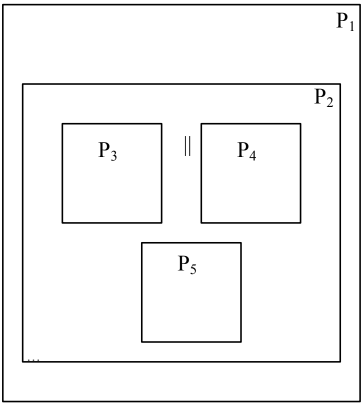
a) A hardware interface

b) An internal logical model

c) A data structure

d) A top-level control structure of a system

4.30 *Serial, parallel*, and *nested* architectures are the basic architectures of software systems. Try to describe the architecture of the following system, §, using proper RTPA notations.



4.31 Referring to Sections 4.7.4 and 4.8.2, describe what is the generic top-level structure of a real-time systems modelled by the event-dispatching-based dynamic behaviours.

4.32 Read the following classic article in software engineering:

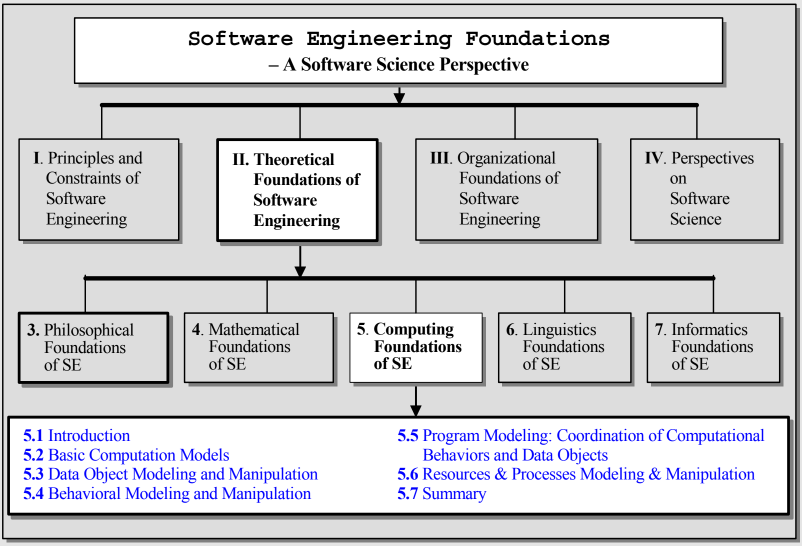
Juris Hartmanis (1994), On Computational Complexity and the Nature of Computer Science, The 1993 Turing Award Lecture, *Communications of the ACM*, 37(10), pp.37-43.

Discuss the following topics in a group:

- About the author.
- What was the nature of computer science according to the author in the 1990s?
- What is the role of mathematics in computer science?
- What conclusions of the article interested you? Why?
- Your argument(s) or counter-points on any of the conclusions derived in this article.

Chapter 5

COMPUTING FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective

I. Principles and Constraints of Software Engineering

II. Theoretical Foundations of Software Engineering

III. Organizational Foundations of Software Engineering

IV. Perspectives on Software Science

3. Philosophical Foundations of SE

4. Mathematical Foundations of SE

5. Computing Foundations of SE

6. Linguistics Foundations of SE

7. Informatics Foundations of SE

5. Computing Foundations of Software Engineering

Knowledge Structure

- Basic computation models
  - Basic operations in computing
  - Turing machines
  - Cognitive machines
  - Automata
  - von Neumann machines
- Data object modeling and manipulation
  - Types and data structures
  - Formal type theory
  - Basic data modeling techniques
  - Abstract data types (ADTs)
- Behavioral modeling and manipulation
  - Internal behaviors modeling
  - Iterative and recursive behaviors modeling
  - External and Interactive behaviors modeling
- Program modeling: coordination of computational behaviors and data objects
  - The unified mathematical model of programs
  - Programs modeling at component level
  - Programs modeling at system level - Frameworks
- Resources and processes modeling and manipulation
  - Abstract model of computing systems
  - Architectures of operating systems
  - Computing resources manipulation
  - Real-time/embedded resources and processes manipulation

Learning Objectives

- To understand the *basic computing models* such as automata, Turing machines, von Neumann machines.
- To know the techniques for *data object* and *architecture* modeling and manipulation in software engineering.
- To know the techniques for *system behavioral* modeling and manipulation in software engineering.
- To be familiar with the techniques for *program* modeling and manipulation in software engineering.
- To be familiar with the techniques for software engineering *resources* and *process* modeling and manipulation in software engineering.
- To understand the unified program model.
- To understand the generic abstract model of computing systems.

"The entire computing theory is about mathematical models of computers and algorithms."

Lewis and Papadimitriou (1998)

"There are three levels of problems. There is the level of solving a very specific instance ... . That is the level closest to the practitioners. Then there is the level of studying the problem in general, with emphasis on methodology for solving it ... .That is one level up because you are not interested just in a specific instance. Then there is a metatheoretic level where you study the whole structure of a class of problems. This is the point of view that we have inherited from logic and computability theory."

Richard Karp (1985)

5.1 Introduction

Computer science is an inquiry of computational methods, generic computer architectures and implementations, computing objects and their abstract representations, as well as programs and programming methodologies that embody a generic computer for specific applications.

Computing theory is one of the most important and direct foundations of software engineering, because software engineering as an engineering discipline grew out of computer science when the complexity and costs for producing software had become increasingly greater than that of hardware since the 1970s.

As early as in the 1830s, there were creations of machines that attempted to realize many of the principles of modern computers such as Charles Babbage's (1791-1871) *difference engine* and *analytical engine*. However, since these machines were too complicated to be implemented by the 19th century technologies, the leap from theory to practice had to wait until the inventions of electronics, particularly transistors and integrated circuits, a century later.

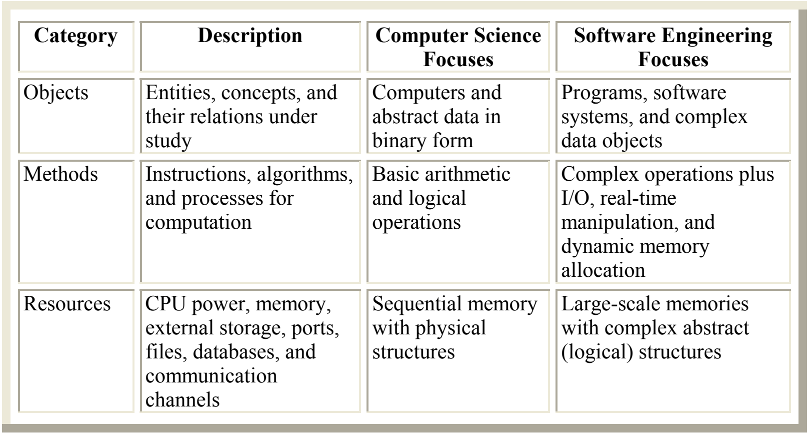
Logical inferences and mathematical induction played a central rule in the ultimate philosophy of computing theory contributed by Charles Babbage, Alan M. Turing (1936, 1950), and John von Neumann (1946, 1958, 1963, 1966). The fundamental *objects* of computation are abstracted by binary digits (bits). Any real-world data object is seen to be able to be reduced to *bits* – the most fundamental and general form of representation of real-world objects and data. As a consequence on the basis of this profound axiom, computation *methods* in general are perceived to be based on the basic arithmetical and logical operations on bits known as Boolean algebra. Any other complex operations must be reduced to these kinds of basic forms of operations in computing. In addition, computing *resources* are dramatically simplified to the form of finite or infinite sequential memory of bits and characters.

That is why the hardware technologies of computing were matured so quickly, because all problems can be unified by the basic operations on the basic objects based on bits. Therefore, hardware devices such as processor and memory chips may be repetitively designed and massively produced. Once a design of a computer is correct, all products based on it must be correct all the time.

However, in software engineering, the development of software is recognized as a one-off activity. To the maximum extent, software design and implementation can only be reduced to known languages components or common design patterns. Although the method and process for software development may be reusable, the objects under study and the resources required in software engineering are far more complicated than those of basic computing hardware techniques.

Table 5.1 summarizes and contrasts the differences of computational objects, methods, and resources in computer science and software engineering. As identified in Table 5.1, computer science only provides basic computing theories and programming methodologies to software engineering. However, areas now thought critical in software engineering – the nature of software, cognitive foundations, denotational mathematics, architectural and behavioral laws, system theories, coordinative work organization theories, and management infrastructures – have not been fully covered by computer science.

Table 5.1
Objects under Study in Computer Science and Software Engineering



Category	Description	Computer Science Focuses	Software Engineering Focuses
Objects	Entities, concepts, and their relations under study	Computers and abstract data in binary form	Programs, software systems, and complex data objects
Methods	Instructions, algorithms, and processes for computation	Basic arithmetic and logical operations	Complex operations plus I/O, real-time manipulation, and dynamic memory allocation
Resources	CPU power, memory, external storage, ports, files, databases, and communication channels	Sequential memory with physical structures	Large-scale memories with complex abstract (logical) structures

This chapter explores the computing foundations of software engineering, and examines what computer science may provide for software engineering as well as what it may not. A new treatment of computing theories for software engineering is taken, which focuses on the needs for modeling and manipulating complicated data objects, behaviors, and resources in software engineering beyond bits.

In the remainder of this chapter, basic computation models, such as automata, Turing machines, von Neumann machines, and cognitive machines, are reinterpreted in the context of software engineering in Section 5.2. Section 5.3 presents data objects modeling with the focuses on type theory and architectural modeling of software systems. Section 5.4 describes behavioral modeling, particularly the Basic Control Structures (BCS's), and highlights their fundamental roles in computing. Section 5.5 models programs as the coordination and interaction between computational behaviors and data objects on the basis of Sections 5.3 and 5.4. Section 5.6 discusses computing resources modeling and manipulation, as well as process coordination, by focusing on generic and real-time operating system models.

Closely related to this chapter, the mathematical models and algebraic treatment of software have been described in Chapter 4. The language aspect of programming and software engineering will be discussed in Chapter 6 with comparative studies between natural and programming languages, syntaxes and semantics, as well as linguistics and formal language theories in computing and software engineering.

5.2 Basic Computational Models

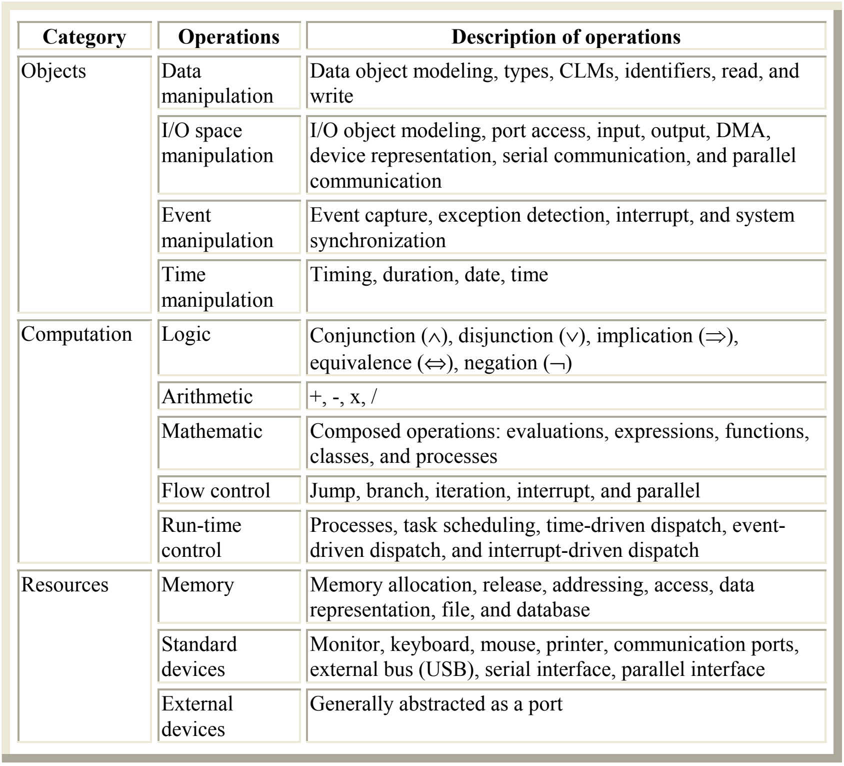
This section first elicits the fundamental needs in computation. Then, it describes the approaches to implement computing machines such as automata, Turing machines, von Neumann machines, and cognitive computers.

5.2.1 BASIC OPERATIONS IN COMPUTING

The philosophy on when computation and software are needed has been discussed in Section 3.4.1, where Theorem 3.9 states that the necessary and sufficient conditions for computing are the repeatability, flexibility, and run-time determinability. As recognized in Section 3.5.2, the problem space of computing is infinite. Further, the solution space for each given application problem can be extremely large because of the combination of possible design and implementation technologies in each software engineering process.

The fundamental operations in computation can be classified into three categories: *computational operations, object manipulations*, and *resource manipulation* as shown in Table 5.2. Although there are various computational operations such as logical, arithmetical, mathematical, flow control, and run-time control, all of them can be reduced to three fundamental Boolean logic operations known as ∧, ∨, and ¬. Objects in computing, such as data, I/O, events, time, and their addresses can be reduced to bits. Then, all resources of computing such as memory, standard devices, and external devices can be reduced to memory locations or port spaces identified by binary addresses.

Table 5.2
Basic Operations in Computing



Category	Operations	Description of operations
Objects	Data manipulation	Data object modeling, types, CLMs, identifiers, read, and write
	I/O space manipulation	I/O object modeling, port access, input, output, DMA, device representation, serial communication, and parallel communication
	Event manipulation	Event capture, exception detection, interrupt, and system synchronization
	Time manipulation	Timing, duration, date, time
Computation	Logic	Conjunction (∧), disjunction (∨), implication (⇒), equivalence (⇔), negation (¬)
	Arithmetic	+, -, x, /
	Mathematic	Composed operations: evaluations, expressions, functions, classes, and processes
	Flow control	Jump, branch, iteration, interrupt, and parallel
	Run-time control	Processes, task scheduling, time-driven dispatch, event-driven dispatch, and interrupt-driven dispatch
Resources	Memory	Memory allocation, release, addressing, access, data representation, file, and database
	Standard devices	Monitor, keyboard, mouse, printer, communication ports, external bus (USB), serial interface, parallel interface
	External devices	Generally abstracted as a port

In a summary, the most fundamental computing needs are only *binary data, basic Boolean operations,* and *a linear memory/port space*. Any complex application can be implemented on the basis of these three essences of computation by certain composition rules. Therefore, all digital computers and computational operations are based on Boolean algebra. This leads to the following theorem on the common root of computer science and information science.

The 15th Law of Software Engineering

Theorem 5.1 The *root of computing and information science* states that the *most fundamental data object model* shared in both computing and information science is *binary digits* (bits).

Theorem 5.1 reveals that the most fundamental data object in computing, informatics, and software engineering is bits. All other data objects in computing are derived objects of bits. Similarly, the following theorem recognizes the most fundamental operations in computing and software engineering.

The 15th Principle of Software Engineering

Theorem 5.2 The *primitive computational behaviors* state that the *most fundamental computational operations* are *logical, arithmetic,* and *memory access operations* on *bits*.

Theorem 5.2 indicates that the most fundamental computing operations and behaviors are bit-based logical, arithmetic, and memory operations. All other operations and behaviors in computing are derived behaviors of them.

Theorems 5.1 and 5.2 are the profound conditions of the whole architecture of modern computing theories, methodologies, and technologies. However, it is noteworthy that some of programming languages, such as Java, puts limit on the expressive power to address all these essential programming requirements, particularly for real-time applications, in order to gain high portability. Some operating systems may also restrict direct access and manipulation of system resources and data objects such as absolute memory addresses, I/O ports, and interrupt events in programming.

The mathematical models of all basic computing operations as shown in Table 5.2 have been described in RTPA, where these operations are categorized into the meta processes and the process relations. The latter are corresponding mainly to the flow and run-time control operations, which enable the algebraic composition of a large set of complex computational operations on the basis of a certain instruction set of a programming language.

5.2.2 AUTOMATA

Automata are one of the earliest digital computing models that are still widely used in computing applications to model and describe system behaviors [von Neumann, 1946/58/63/66; Wiener, 1948; Shannon, 1956; Krohn and Rhodes, 1963; Arbib and Michael 1966; Arbib, 1969; Hopcroft and Ullman, 1979]. An *automaton* is an abstract model of computers or robots that respond to external events or stimulates by predesigned instructions. An automaton transits between a finite set of functional states driven by external events and current internal states. Therefore, it is also known as a finite state machine synonymously [Arbib et al., 1968].

In computing, automata are modeled and used mainly as finite state machines for language reorganization. However, in software engineering, automata are treated as fundamental modeling techniques of software behaviors and interactions with external environments. Therefore, an automaton can be perceived as an event-driven finite state machine. This subsection discusses the definition, formal descriptions, and applications of automata in software engineering, and their usage and limitations.

5.2.2.1 Automata and Finite State Machines (FSMs)

An automaton is a finite state machine based on the mechanism of event-driven state transitions that can be formally defined as follows.

Definition 5.1 A *Finite State Machine* (FSM) is a 5-tuple, i.e.:

$$FSM \triangleq (\Sigma, S, s, T, \delta) \tag{5.1}$$

where

(i) Σ is a finite set of *alphabet*, inputs, or events;
(ii) S is a finite set of internal *states*;
(iii) s is the *initial state*, $s \in S$;
(iv) T is the set of *final states*, $T \subseteq S$; and
(v) δ is the *state transition function*, which is defined as:

$$\delta\colon S \times \Sigma \rightarrow S \tag{5.2}$$

The state transition function δ as defined in Eq. 5.2 can be described equivalently by a *state transition table* with the following schema as shown in Table 5.3.

Table 5.3
The Schema of a State Transition Table of FSMs

Current state	**Current event**	**Next State**
$s_{i,}\ s_i \in \mathrm{S}$	$e_{i,}\ e_i \in \Sigma$	$s_{i+1} = \delta(s_{i,}\ e_i),\ s_{i+1} \in \mathrm{S}$

A practical question before all transitional instances are derived for Table 5.3 is how many possible transitions may be expected for a given layout of an FSM. Because the transition function δ is defined as a Cartesian product in Eq. 5.2, an outstanding advantage of it is that the number of transitions in δ can be exactly predicated by the following definition.

Definition 5.2 The *size of state space* of an FSM, $S_\Omega(FSM)$, is all possible transitions that can be determined by the product of both sizes of the state set S and alphabet set ∑., i.e.:

$$S_\Omega(FSM) = \#S \bullet \#\Sigma \tag{5.3}$$

It is noteworthy that S_Ω (FSM) determined by Eq. 5.3 predicates all possible transitions of an FSM, which encompasses both *legal* and *illegal transitions* that represent the entire behaviors of the FSM. The former are the defined transitions in δ as defined in Eq. 5.2, and the latter are exceptional transitions outside δ denoted by $\overline{\delta}$. It may be expected that $\overline{\delta}$ could be much larger than δ. Therefore, one of the important tasks of professional system architects and analysts is to identify the whole state space of an FSM and to rule out all possible exceptional transitions.

Based on the above discussion, Eq. 5.3 may be extended to the following in order to consider the two important portions of transitional behaviors of a given FSM, i.e.:

$$\begin{aligned} S_\Omega(FSM) &= \#S \bullet \#\Sigma \\ &= \#\delta + \#\overline{\delta} \end{aligned} \tag{5.4}$$

Therefore, it can be perceived in software engineering that, to a certain extent, *requirement engineering* elicits δ from users' needs for a given system; while *system specification* identifies the entire space of all possible behaviors of the required system, S_Ω $(FSM) = \#\delta + \#\overline{\delta}$, in order to prevent the FSM from going into any of the illegal transitions.

The 16th Principle of Software Engineering

Theorem 5.3 The *nature of requirements and specifications* states that *requirement elicitation* focuses on desired functions of a system δ, while *system specification* focuses on the entire behavioral space of the system Ω, including both δ and the undesired but potential system transitions represented by $\overline{\delta}$ in the behavioral space, i.e.:

$$\begin{aligned} S_\Omega &= \#\delta + \#\overline{\delta} \\ &= \#S \bullet \#\Sigma \end{aligned} \tag{5.5}$$

It is noteworthy, according to Theorem 5.3, that most software systems may go wrong not because they are incorrect on normally required functions, but because there are wrong or not prepared for implied and nonspecified exceptions.

Corollary 5.1 For a software system, particular a complex system, the size of undesired behavior space is far more greater than that of the desired one, i.e.:

$$\#\overline{\delta} \gg \#\delta \tag{5.6}$$

According to Corollary 5.1, system design and specification should focus on the entire S_Ω, or should emphasize on both δ and $\overline{\delta}$. This is a major indicator that distinguishes professionals and amateurs in software engineering, where the latter focus only on required behaviors (δ) and rush to implement them, while the former thoughtfully model the whole behaviors of a given system ($\overline{\delta}$ + δ) and eliminate the possibility for the system under design crashes into any undesired exceptional states.

5.2.2.2 Approaches to Describe FSMs

On the basis of discussions in Sections 5.2.2.1 and 4.7, four approaches to describe FSMs can be summarized below:

a) To define the 5-tuple for a given FSM according to Definitions 5.1 and 5.2;

b) To define the state transition table for the FSM using the template as given in Table 5.3;

c) To draw a labeled digraph known as the *state diagram* as shown in Fig. 5.1;

d) To specify an event-driven RTPA process as discussed in Section 4.7 and illustrated in Example 5.2.

The following examples demonstrate the methods for specifying FSMs according to the above four approaches.

Example 5.1 According to Definition 5.1, an automaton FSM_1 can be described as follows:

$$FSM_1 = (\Sigma, S, s, T, \delta)$$

where $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, $s = s_0$, and $T = \{s_2\}$.

The transition function $\delta: S \times \Sigma \rightarrow S$ is given in Table 5.4. According to Eq. 5.3, the state space includes six legal transitions and two illegal ones, because once FSM_1 reaches the final state s_2, it will no longer be able to change its state.

Table 5.4
The State Transition Table of FSM_1

Current state s_i	**Current event** e_i	**Next state** $s_{i+1} = \delta(s_{i,}\ e_i)$	**Category**
s_0	A	s_1	δ
s_0	B	s_2	
s_1	A	s_1	
s_1	b	s_2	
s_2	a	-	$\overline{\delta}$
s_2	b	-	

An equivalent state diagram of FSM_1 can be derived on the basis of Table 5.4 as shown in Fig. 5.1, where a square block denotes the initial state, and a double circle denotes the final state.

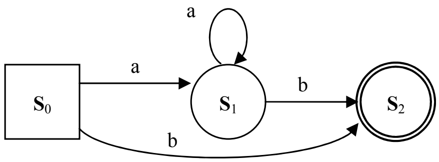


Figure 5.1 The state diagram of FSM_1

Example 5.2 The fourth method for formally describing an FSM is by using RTPA. The RTPA specification of *FSA₁* is given below.

$$
\begin{array}{l}
FSA_1 \triangleq \{ s_0 \rightarrow \\
\qquad\qquad\qquad (\;\; @a\mathbf{S} \;↳\; s_1 \\
\qquad\qquad\qquad\quad \rightarrow (\;\; @a\mathbf{S} \;↳\; s_1 \\
\qquad\qquad\qquad\qquad\;\; \mid @b\mathbf{S} \;↳\; s_2 \\
\qquad\qquad\qquad\qquad\;) \\
\qquad\qquad\qquad \mid @b\mathbf{S} \;↳\; s_2 \\
\qquad\qquad\qquad) \\
\qquad\qquad \}
\end{array}
\qquad (5.7)
$$

Further details may be referred to the RTPA notations and the system specification methodology described in Sections 4.6 and 4.7.

Due to the well-defined theories and methods, FSMs have found a wide range of applications in computing and software engineering. This is why the state diagrams have been adopted in UML as an important part of its diagram-based notations for modeling software systems.

5.2.2.3 Description of Software Behaviors by FSMs

The preceding section shows that automata theories are expressive and rigorous methods that can be applied to describe and specify software system behaviors. Especially, it provides a powerful means to predict the possible state space or domain of behaviors of a given system requirement or specification according to Theorem 5.3.

This subsection takes the ATM as previously described in Section 4.8.2 and Fig. 4.11 as a real-world example to demonstrate the application of FSM technologies in system modeling and specification [Wang and Zhang, 2003].

Example 5.3 The ATM as shown in Fig. 4.11 can be abstracted as an FSM as described in Fig. 5.2. Therefore, the ATM can be formally described as follows:

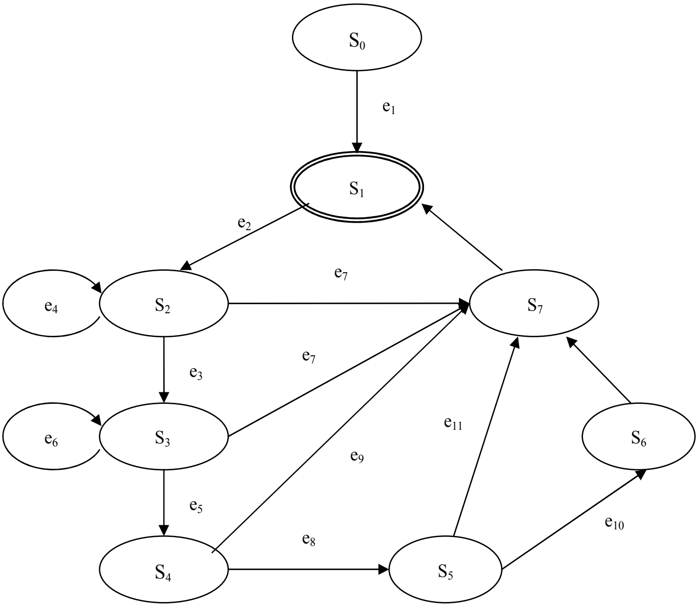


Figure 5.2 The abstract FSM model of the ATM

$$ATM \triangleq (\Sigma, S, s, T, \delta) \qquad (5.8a)$$

Corresponding to the given ATM in Fig. 5.2, the components of the 5-tuple as given in Eq. 5.8a can be formally defined below:

- $\Sigma = \{e_1, e_2, ..., e_{11}\}$, where:

e_1 - Start
e_2 - Insert card
e_3 - Correct PIN
e_4 - Incorrect PIN
e_5 - Cash $\leq$ max
e_6 - Cash > max
e_7 - Cancel transaction
e_8 - Sufficient funds
e_9 - Insufficient funds
e_{10} - Sufficient cash in ATM
e_{11} - Insufficient cash in ATM (5.8b)

- $S = \{s_0, s_1, \ldots, s_7\}$, where:

  s_0 - System
  s_1 - Welcome
  s_2 - Check PIN
  s_3 - Check amount
  s_4 - Verify account balance
  s_5 - Verify cash availability
  s_6 - Disburse cash
  s_7 - Eject card (5.8c)

- $s = s_1$ // welcome (5.8d)

- $T = \{s_1\}$ // welcome (5.8e)

- $\delta = f$: $S \times \sum \rightarrow S$ is given in Table 5.5 corresponding to the state diagram model of the ATM system as shown in Fig. 5.2.

Table 5.5
The State Transition Table of the ATM

s_i	e_i	$s_{i+1} = \delta(s_i, e_i)$
s_0	e_1	s_1
s_1	e_2	s_2
s_2	e_3	s_3
s_2	e_4	s_2
s_2	e_7	s_7
s_3	e_5	s_4
s_3	e_6	s_3
s_3	e_7	s_7
s_4	e_8	s_5
s_4	e_9	s_7
s_5	e_{10}	s_6
s_5	e_{11}	s_7
s_6	-	s_7
s_7	-	s_1
…	…	$\bar{\delta}$

Example 5.4 According to Theorem 5.3, the size of behavior space of the ATM can be predicated as:

$$\begin{aligned} S_\Omega(\text{ATM}) &= \#\delta + \#\bar{\delta} \\ &= \#S \bullet \#\sum \\ &= 8 \bullet 11 \\ &= 88 \end{aligned}$$

where $\#\delta = 12 + 2 \bullet 11 = 34$, and $\#\bar{\delta} = S_\Omega - \#\delta = 88 - 34 = 54$.

Special care should be taken in the design of this ATM to prevent all the 54 illegal transitions from happening. If any of these transitions occurs, the system should be ready to handle it as an exception.

A complete specification of the ATM system in RTPA, in terms of its architecture and behaviors, is provided in Section 4.8.2 and Appendix K.

5.2.2.4 FSM Composition and Refinement

In an FSM, a state can be extended to a sub-FSM. This provides a powerful approach to refine a system design and specification, hierarchically.

Example 5.5 A given automaton FSM$_0$ can be refined by three more detailed automata FSM$_1$, FSM$_2$, and FSM$_3$, as shown in Fig. 5.3, where the event-driven relations between the sub-FSMs are also provided. A formal description of the refinement of FSM$_0$ in RTPA is provided in Eq. 5.9.

$$\begin{aligned}
\text{FSM}_0 \triangleq \{ \; & \text{FSM}_1 \\
& \quad \to (\;\; e_1 \hookrightarrow \text{FSM}_2 \\
& \qquad\;\; \mid e_2 \hookrightarrow \text{FSM}_3 \\
& \qquad) \\
\} \; &
\end{aligned} \tag{5.9}$$

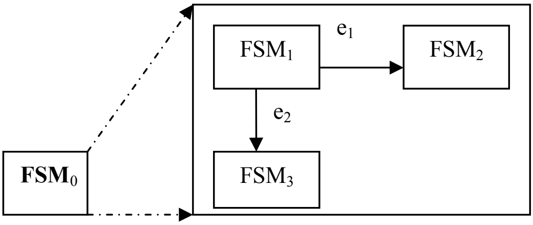


Figure 5.3 Refinement of an FSM by sub-FSMs

If the *refinement* of FSM$_0$ described above is perceived as decomposition of an FSM into sub-FSMs, the *composition* of a number of FSMs into a coherent system can be explained as an inverse operation of FSM refinement. Both FSM refinement and composition provide a foundation for software system modeling, specification, and integration.

5.2.2.5 Deterministic and Nondeterministic Automata

There are *deterministic* and *nondeterministic* automata. The former represent the automata that their transition functions are known for each given event in the context of the current state; the latter represent the automata that their transition functions are not unique, or the possible next states are multiple on a given event and the current state. Therefore, the behaviors of a nondeterministic FSM are inpredictable and should require additional information or be arbitrarily instantiated at run-time.

The deterministic FSMs have been defined in Definition 5.1, This subsection introduces the definition of nondeterministic FSMs and their relationships with the deterministic counterparts.

Definition 5.3 A *nondeterministic FSM*, *FSM'*, is defined by a 5-tuple, i.e.:

$$FSM' \triangleq (\Sigma', S, s, T, \delta') \tag{5.10}$$

where

(i) Σ' is a finite set of *alphabet* plus a generic freely transitional event known as the empty event ϕ;
(ii) *S* is a finite set of internal *states*;
(iii) *s* is the *initial state*, $s \in S$;
(iv) *T* is the set of *final states*, $T \subseteq S$;
(v) *δ'* is the *state transition function*, which is defined as:

$$\delta': S \times \Sigma' \to S \tag{5.11}$$

Comparing Definitions 5.1 and 5.3, it can be seen that the only differences between a deterministic and nondeterministic FSM is the extension of Σ by ϕ, and the replacement of the transition function δ by δ' that is no longer unique and allows multiple arbitrary next states. In other words, a nondeterministic FSM is a special case of the deterministic one. This leads to the following Corollary.

Corollary 5.2 The *deterministic and nondeterministic FSMs* are *equivalent.* That is, for any given nondeterministic FSM, there is an equivalent deterministic FSM.

A proof of Corollary 5.2 may be referred to [Lewis and Papadimitriou, 1998].

According to Corollary 5.2, a nondeterministic FSM can always be converted into an equivalent deterministic counterpart. Conversion between nondeterministic and deterministic FSMs can be enabled by providing additional rules in the transition function or simply allowing arbitrary transitions among defined (legal) multiple next states on the basis of a given event and the current state of the system.

5.2.2.6 Usage of Automata

Automata are found useful for describing software behaviors based on event-driven mechanisms at framework level. However, their expressive power is limited when there is a series of complicated actions or algorithms responding to a specific event or request, which is modeled as a process in RTPA in Chapter 4. In addition, automata lack the capability to model the architectures and data objects of software systems as RTPA does, which are an equally, if not a more, important part of software system modeling and specification. These are the reasons why automata and FSMs have not been adopted as the main software engineering notation system and modeling techniques.

The 17th Principle of Software Engineering

Theorem 5.4 The *weaknesses of automata* state that automata and FSMs as a system composition and modeling method built on event-driven mechanisms are inadequate to model the complete basic computational requirements, particularly the lack of the descriptive power for:

a) System architectures and data objects modeling;

b) Nonevent-driven transitional process modeling;

c) Detailed behavioral descriptions;

d) Mathematical operations and processing of complicated languages.

Revealed by Theorem 5.4, the features, descriptive power, and suitability for large-scale systems modeling by FSMs and RTPA can be observed by comparing Examples 5.1, 5.2, 5.3, and the models presented in Section 4.8.2. In dealing with large-scale system specifications, it is found that the algebraic description of an automaton in RTPA is more convenient and rigorous. Another advantage of the RTPA methodology is that its work products for system modeling are much closer to the form of programs, the naturally succeeding phase in software system development.

5.2.3 TURING MACHINES

Turing machines are the most fundamental mathematical model of computation that are perceived, in theory, to imitate logical human thought. A Turing machine provides a generic abstract model for digital computers, and reveals the basic computability of problems and their implementation by the simplest computing machines.

5.2.3.1 The Abstract Model of Computing

The development of Turing machines is a great application of fundamentalism as described in Section 3.2.3 in computing, in order to seek the most fundamental needs and mechanisms for computing. According to Theorems 5.1 and 5.2, the most fundamental data object modeling technique is bits. Based on it the most fundamental computational operations are logical and architectural operations on bits.

In his classic paper *On Computable Numbers, with an Application to the Entscheidungs Problem*, Turing (1936) described that: "All detailed sets of instructions that can be carried out by a *human calculator* can also be carried out by a suitably defined *simple machine.*" It is known then as the Turing machine.

According to Theorems 5.1 and 5.2, a Turing machine is the simplest model of computing and machine intelligence. Any complicated computing machine can be reduced onto a number of basic Turing machines. This provides a practical approach to build large and complicated computing systems based on simple ones. Any computational task that cannot be processed by Turing machines is a non-determinable problem.

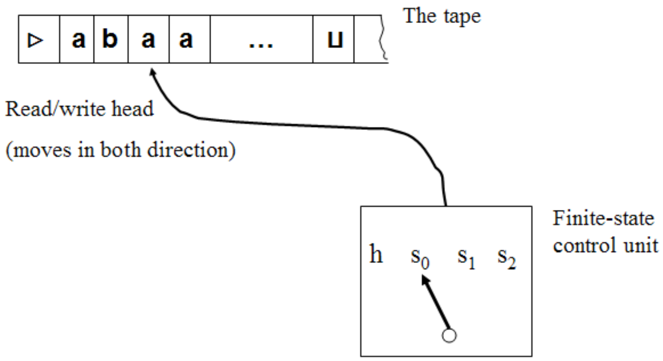


Figure 5.4 A Turing machine

A typical Turing machine can be illustrated as shown in Fig. 5.4. The Turing machine encompasses three basic components: (a) the finite-state *control unit*, (b) the *tape* (memory) with finite or infinite cells, and (c) the *read/write head*. In Fig. 5.4, the symbol ⊔ represents a blank space, and ▷ represents the left end or the beginning of the tape. Among the finite states of the control unit, *h* is the halting or termination state of the machine.

5.2.3.2 Formal Description of Turing Machines

A formal model of a Turing machine can be described as follows.

Definition 5.4 A *Turing machine* (*TM*) is defined by a 6-tuple, i.e.:

$$TM \triangleq (\Sigma, S, s, H, M, \delta) \tag{5.12}$$

where

(i) Σ is the finite set of *alphabet*, $\Sigma = \{R, W, \sqcup, \triangleright\}$, where R represents a finite set of symbols read from the tape, W a finite set of symbols written to the tape, ⊔ the *blank space*, and ▷ the beginning of the tape;

(ii) S is the finite set of *states*;

(iii) s is the *initial state*, $s \in S$;

(iv) H is the set of *halting states*, $H \subseteq S$;

(v) M is the set of *head movements*, $M = \{\leftarrow, \rightarrow, \Diamond\}$ denotes move to left, right, or no move, respectively.

(vi) δ is the *state transition function* that is defined by:

$$\delta: (S \setminus H) \times \Sigma \rightarrow S \times \Sigma \times M \tag{5.13}$$

where the state transition function $\delta(s_\mathrm{i}, r_\mathrm{i}) = (s_{\mathrm{i+1}}, w_\mathrm{i}, m_\mathrm{i})$ denotes that when the TM is in state s_i, $s_i \in S\backslash H$, and scanning symbol r_i, $r_i \in \mathrm{R} \subseteq \Sigma$, it will carry out three actions: (a) write symbol w_i, $w_i \in W \subseteq \Sigma$, onto the tape at the current position; (b) conduct a movement of the head m_i, $m_i \in M = \{\leftarrow, \rightarrow, \Diamond\}$; and (c) transfer to the next state s_{i+1}, such that:

(1) $w_i \neq \triangleright$; and
(2) If $r_i = \triangleright$, then $w_i = \phi$ and $m_i = \rightarrow$;

According to Definition 5.4, the transition function δ of the TM is not defined on any states in *H*. In other words, when the TM reaches a halting state its operation will be totally terminated. In the above definition, Condition (1) on δ means that the TM will never write additional ▷ on the tape, so that ▷ is the unmistakable sign of the left end of the tape; and Condition (2) means when the TM sees the left end of the tape ▷, it must move right in order to maintain that the leftmost $\triangleright$ is never erased in order to prevent the TM from falling off the left end of the tape.

Example 5.6 A Turing machine TM_1 is designed to simulate 3 + 5 = 8, where the initial and final tape contents are $[\triangleright,0,0,1,1,1,0,\sqcup,0,1,1,1,1,1,0,0,0, \ldots]$ and $[\triangleright,1,1,1,1,1,1,1,1,0,0,0, \ldots]$, respectively. Let a finite set of states be given below:

s_0: Scans rightwards through the first group of 1's until the blank separator $\sqcup$ is read;

s_1: Scans rightwards through the second group of 1's until it finds a 0;

s_2: Erases the 1 scanned from the current position;

s_3: Writes the current scanned 1 at the tail of the first group of 1's.

The above given TM_1 can be formally defined as follows:

$$TM_1 \triangleq (\Sigma, S, s, H, M, \delta) \tag{5.14}$$

where

- $\Sigma = \{1, 0, \sqcup, \triangleright\}$
- $S = \{s_0, s_1, s_2, s_3, h\}$
- $s = s_0$
- $H = \{h\}$
- $M = \{\leftarrow, \rightarrow, \lozenge\}$
- δ: $(S \setminus H) \times \Sigma \rightarrow S \times \Sigma \times M$ is as given in Table 5.6 below:

Table 5.6
The State Transition Table of TM$_1$

s_i	r_i	$(s_{i+1}, w_i, m_i) = \delta(s_i, r_i)$
s_0	0	$(s_1, 0, \rightarrow)$
s_0	1	$(s_0, 1, \rightarrow)$
s_1	0	$(s_2, 0, \leftarrow)$
s_1	1	$(s_1, 1, \rightarrow)$
s_2	0	$(s_2, 0, \leftarrow)$
s_2	1	$(s_3, 0, \leftarrow)$
s_3	0	$(h, 0, \lozenge)$
s_3	1	$(s_3, 1, \leftarrow)$

Comparing the definitions of FSM and TM, it is noteworthy that the extension of descriptive power in TMs over FSMs is the introduction of both the output operation by writing a symbol w_i onto the tape, and the head action m_i associated to a state transition s_{i+1}. In the TM model, the output actions can be either to rewrite the currently scanned symbol r_{i-1} onto the tape in the current place or another designated symbol in the alphabet. The head movement actions can be any one in $M = \{\leftarrow, \rightarrow, \lozenge\}$. However, if considering that a state of FSM is a process that may be able to carry out more powerful operations, then FSM is equivalent to TM.

Although there are a variety of TMs, it can be proven that all TMs are equivalent [Gersting, 1982; Lewis and Papadimitriou, 1998; McDermid, 1991]. With the extension of TMs over FSMs on both the I/O capacity and the infinite memory tape, virtually any complex computing activities may be sufficiently modeled and implemented. This can be proven by Theorems 5.1 and 5.2.

5.2.3.3 The Nature of Computing

Definition 5.5 A *computation* of a TM is a finite sequence of state transitions, i.e.:

$$\mathop{R}\limits_{i\mathbf{N}=1}^{k\mathbf{N}} ((s_i, r_{i-1}, m_{i-1}) \rightarrow (s_{i+1}, w_i, m_i)) \qquad (5.15a)$$

where $(s_0, -, -)$ is the initial state, (s_k, a_{k-1}, h) is the halting state, and the trace of I/Os or the processes of an TM computation is:

$$(r_0, w_1) \rightarrow (r_1, w_2) \rightarrow \ldots \rightarrow (r_{k-1}, w_k) \qquad (5.15b)$$

According to Definition 5.4, a certain TM is a fixed and unprogrammable computing machine, specialized at solving a particular problem with instructions (δ) that are hard-wired. An FSM can be perceived as a restricted TM where the head is read-only and shift only from left to right.

Turing's contribution is the identification of the basic mechanism for computing and machine intelligence. Turing's theory shows that the basic elements needed for computing are: $TM = (\Sigma, S, s, H, M, \delta)$. Any complex computing system may be decomposed into a number of simple TMs. In this view, Turing machines are the most *fundamental* computing structures rather than the most *powerful* machines.

According to Turing's theory as well as Theorems 5.1 and 5.2, the basic functionality for computing can be summarized in the following theorem.

The 18th Principle of Software Engineering

Theorem 5.5 The *fundamental computational capabilities* state that the essential capabilities for computation are as follows:

- A memory for storing bit information;
- A simple addressing capability for accessing information in the memory;
- Read/write operations for retrieving or updating the memory;
- A conditional and quantitative evaluation capability for interpreting the inputted information;
- A stored-information-driven mechanism for determining the next step.

The findings in Theorem 5.5 are significant because the theorem reveals that intelligence is *memory-based* [Wang and Wang, 2006]. Further, it indicates that computing, a highly abstract machine intelligence, can be reduced to a sequence of *simple memory manipulations*, such as addressing, reading, and writing, as well as quantitative evaluations.

A TM is capable to process memory-stored information as a closed system. However, it cannot process interacting I/O events from/to the external world, which is seen as a basic requirement for a modern computing system. It also lacks the descriptive power for: a) System architectures and data objects modeling; b) Nontransitional behavioral modeling; and c) High-level data objects and behavioral modeling.

The meta processes of RTPA as presented in Chapter 4 modeled all the basic computational operations as well as their algorithmic composing rules. Therefore, RTPA is a high-level and convenient mathematical means for formally denoting computing needs and computational behaviors on the basis of modern computers.

5.2.4 VON NEUMANN MACHINES

Sections 5.2.2 and 5.2.3 described the abstract models of low-level computers in terms of FSMs and TMs. Although these models reveal the necessary functions of computation, they possess insufficient functions to computing. This may be analogized to those of using machine languages vs. high-level languages in programming.

In order to develop more powerful high-level computers, the weaknesses of the low-level computational models, such as TM and automata, have to be enhanced. This subsection introduces the stored-program concept in computing and a high-level computing model known as the von Neumann machines.

5.2.4.1 The Stored-Program Concept

Stored-program computers and systems are a remarkable technical advance in the evaluation of computer architectures and implementation technologies. Before the introduction of the stored-program concept, computing machines are designed and controlled by wired logic or electronic circuits, in which changes of instructions in applications require rewiring of the physical machine.

Definition 5.6 *Stored-program techniques* are a computer organization technology that treats instructions of a computer as the same of data objects, and then the abstract representation of the instructions – the program – can be interpreted and executed by the Central Processing Unit (CPU).

The stored-program concept can be traced back to the times of Babbage in mid 1940s. John von Neumann formalized the technology of stored-program computers [von Neumann, 1946], and it is still the most widely used architecture of modern computers. Based on the successful development of the first electronic digital computer ENIAC (Electrical Numerical Integrator and Calculator) in 1946, von Neumann developed an abstract model of computers with a universal structure, and yet be able to execute any kind of computation by means of programmable control without the need for changing the physical units of the computer. The above concept, usually referred to as the stored-program technology, became essential for modern digital computers.

The main purpose for introducing stored-program control is to provide flexible computation where updates of desired functions may be introduced primarily through program modification rather than through changes of hardware. Therefore, stored-program may be considered as the central concept of the von Neumann architecture of modern digital computers. It is also the foundation of theoretical and functional equivalence between hardware and software in computation.

5.2.4.2 The von Neumann Architecture of Computers

The key requirements for implementing a stored-program controlled computer are: a) The generalization of common computing architectures; and b) The generic computer is able to interpret the data loaded in memory as computing instructions. These are the essences of stored-program computers with *von Neumann architecture* [von Neumann, 1946/58; Stallings, 1987]. von Neumann elicited the five fundamental and essential components to implement general-purpose programmable digital computers in order to embody the concept of stored-program-controlled computers.

Definition 5.7 A *von Neumann Architecture* (VNA) of computers is a 5-tuple that consists of five components: (a) the *Arithmetic-Logic Unit* (ALU), (b) the *Control Unit* (CU) with a *Program Counter* (PC), (c) a *memory* (M), (d) a set of *Input/Output* (*I/O*) *devices*, and (e) a *bus* (B) that provides the data path between these components, i.e.:

$$VNA \triangleq (ALU,\ CU,\ M,\ I/O,\ B) \tag{5.16}$$

Definition 5.8 *von Neumann Machines* (VNMs) are VNA-based computers aiming at *stored-program-controlled* data processing based on mathematical logic and Boolean algebra.

A VNM can be illustrated in Fig. 5.5, which is centric by the bus and characterized by the all-purpose memory for both data and instructions. Comparing Figs. 5.5 and 5.4, it can be seen that a VNM is an enhanced Turing machine (TM), where the power and functionality of all components of TM including the control unit (with wired instructions), the tape (memory), and the head of I/O are greatly enhanced and extended with more powerful instructions and I/O capacity.

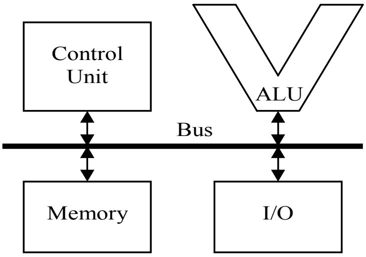


Figure 5.5 The von Neumann architecture of computers

Definition 5.9 An *abstract model of a VNM* performs computation in the following iterative steps:

$$
\begin{array}{l}
\S(\text{VNAMachine}) \triangleq \\
\quad \{ \\
\quad\quad \text{PC}\mathbf{N} := 0 \\
\quad\quad \to \mathop{R}\limits_{\text{EOF}\mathbf{BL}=\mathbf{F}}^{\mathbf{T}} (\ \text{MEM}[\text{PC}\mathbf{N}]\mathbf{B} \Rightarrow \text{Inst}\mathbf{B} \\
\quad\quad\quad\quad\quad\quad \to \uparrow(\text{PC}\mathbf{N}) \\
\quad\quad\quad\quad\quad\quad \to \text{�}\text{Inst}\mathbf{B} = (\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \begin{array}{ll}
0: & \rightarrowtail \text{Instr}_0 \\
1: & \rightarrowtail \text{Instr}_1 \\
\ldots & \\
\text{i}: & \rightarrowtail \text{Read} \\
\text{i+1}: & \rightarrowtail \text{Write} \\
\text{i+2}: & \rightarrowtail \text{Input} \\
\text{i+3}: & \rightarrowtail \text{Output} \\
\text{i+4}: & \rightarrowtail \text{PC}\mathbf{N} := k\mathbf{N},\ k\mathbf{N} \le \text{n}\mathbf{N} \\
\ldots & \\
\text{m}: & \rightarrowtail \boxtimes\ )
\end{array} \\
\quad\quad\quad\quad\quad\quad\quad\quad) \\
\quad\quad\quad\quad\quad\quad) \\
\quad \}
\end{array}
\tag{5.17}
$$

where $n\mathbf{N}$ is the length of the program, $Instr_i$ denotes a specific instruction in the instruction set of a given computer [Wang, 2006h].

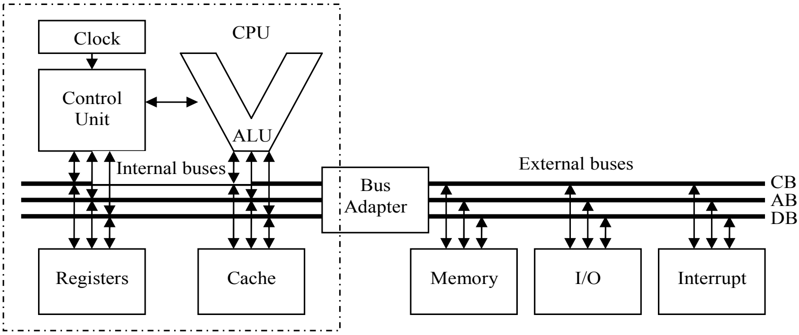


Figure 5.6 Typical modern computer architecture based on VNA

A modern VNM can be illustrated as shown in Fig. 5.6. Although the architecture of computers has changed little since von Neumann's times and most modern computers are still based on VNA, new extensions and enhancements have been developed and implemented in almost all aspects of VNMs as follows [Stallings, 1987; Hennessy and Patterson, 1996]:

- More powerful instruction set, register set, and on chip cache have been introduced into CPU. Pipeline techniques are adopted for instruction decoding and executions.

- The bus has been separated into control, address, and data buses, as well as internal and external buses.

- The memory has been extended from the 1-D sequential memory to 2-D segmented memory and virtual memory.

- The pure internal stored-program computing has been extended to be able to process external events and interrupts for advanced I/Os and multi-threads.

Trends in advanced computer architectures beyond VNM are *parallel, networking,* and *cognitive* computers. Parallel and networking computers may be implemented with a set of homogeneous/heterogeneous VNMs or non-VNMs. The theory of cognitive computers may result in new architectures of computers as discussed in the following subsection.

5.2.5 COGNITIVE MACHINES

The theory and philosophy behind the next generation computers and computing technologies are cognitive informatics (Chapter 9) [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06] and denotational mathematics [Wang, 2002a/05a/06d/06e/06f/06j/07a]. It is commonly believed that the future-generation cognitive computers will adopt *non-von Neumann architectures*.

5.2.5.1 The Wang Architecture of Computers

Definition 5.10 A *Wang Architecture* (WA) of computers, known as a *Cognitive Machine*, is a parallel structure encompassing an *Inference Engine* (IE) and a *Perception Engine* (PE), i.e. [Wang, 2006b]:

$$
\begin{aligned}
WA \triangleq\ & (\text{IE} \parallel \text{PE}) \\
=\ & (\quad \text{KMU} && \text{// The } \textit{Knowledge} \text{ Manipulation Unit} \\
& \quad \parallel \text{BMU} && \text{// The } \textit{Behavior} \text{ Manipulation Unit} \\
& \quad \parallel \text{EMU} && \text{// The } \textit{Experience} \text{ Manipulation Unit} \\
& \quad \parallel \text{SMU} && \text{// The } \textit{Skill} \text{ Manipulation Unit} \\
& \quad) \\
& \parallel (\quad \text{BPU} && \text{// The } \textit{Behavior} \text{ Perception Unit} \\
& \quad \parallel \text{EPU} && \text{// The } \textit{Experience} \text{ Perception Unit} \\
& \quad)
\end{aligned}
\qquad (5.18)
$$

As illustrated in Fig. 5.7, WA computers as defined in Eq. 5.18 are not centered by a CPU for data manipulation as the VNA computers do. The WA computers are centered by the concurrent IE and PE for cognitive learning and autonomic perception based on abstract concept inferences and empirical stimulus perception. The IE is designed for concept/knowledge manipulation according to concept algebra [Wang, 2006b], particularly the nine concept operations for knowledge acquisition, creation, and manipulation. The PE is designed for perception processing according to RTPA and the formally described cognitive process models of the perception layers as defined in the LRMB model in Section 9.3.1.

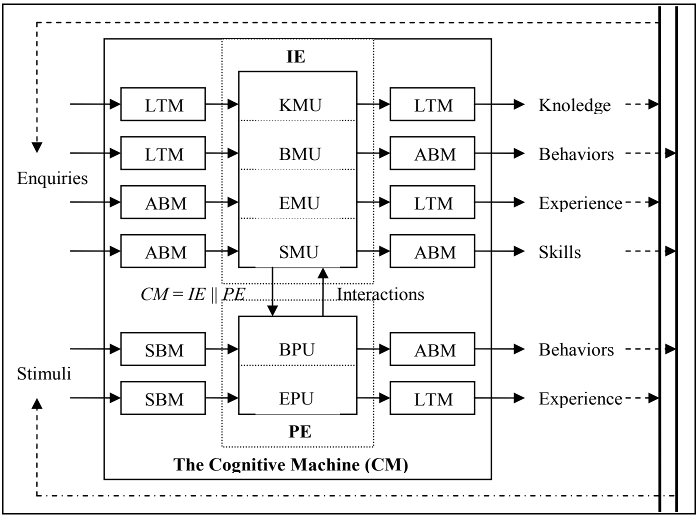


Figure 5.7 The architecture of a cognitive machine

5.2.5.2 Cognitive Computers

Definition 5.11. *Cognitive computers* with WA are aimed at cognitive and perceptive concept/knowledge processing based on contemporary denotational mathematics, i.e., *Concept Algebra*, RTPA, and *System Algebra*.

As that of mathematical logic and Boolean algebra are the mathematical foundations of VNA computers. The mathematical foundations of WA computers are based on contemporary denotational mathematics. According to the LRMB reference model [Wang et al., 2006], all the 39 fundamental cognitive processes of human brains can be formally described in denotational mathematics, particularly concept algebra and RTPA [Wang, 2006e/02a], which can be implemented and simulated by WA-based cognitive computers.

One paradigm of cognitive computers is the *autonomic computers* [IBM, 2001/06; Pescovitz, 2002; Kephart and Chess, 2003; Murch, 2004; Wang, 2003d/04a/07e/04f], which is a nonimperative computer based on non-VNA, e.g., WA, that autonomously carries out robotistic and interactive applications based on goal- and inference-driven mechanisms on the basis of nonlinear and content sensitive memory architectures.

The history towards autonomic computing may be traced back to the work on automata by Norbert Wiener, John von Neumann, Alan Turing, and Claude E. Shannon as early as in the 1940s [Wiener, 1948; von Neumann, 1946/58/63/66; Turing, 1950; Shannon, 1956; Rabin and Scott, 1959]. In the same period, Warren McCulloch proposed the term of artificial intelligence (AI) [McCulloch, 1943/65/93], and S.C. Kleene analyzed the relations of automata and nerve nets [Kleene, 1956]. Then, Bernard Widrow developed the term of artificial neural networks in the 1950s [Widrow and Lehr, 1990; Harvey, 1994]. The concepts of robotics [Brooks, 1970] and expert systems [Giarrantans and Riley, 1989] were developed in the 1970s and 1980s, respectively. Then, intelligent systems [Meystel and Albus, 2002] and software agents [Negreponte, 1995; Chorafas, 1998; Jennings, 2000] emerged in the 1990s. These events and developments led to the formation of the concept of autonomic computing. This subject will be further discussed in Section 15.4.

5.3 Data Object Modeling and Manipulation

As highlighted in the previous sections, a major thread of this chapter on computing foundations of software engineering is that computational operations can be classified into the categories of data object, behavior, and resources modeling and manipulations. Based on this view, programs are perceived as the coordination of data objects and behaviors in computing.

Definition 5.12 *Data object modeling* in computing is a process to creatively extract and abstractly represent a real-world problem by computing objects based on the constraints of given computing resources.

Relations between the data objects and resources form the architectural model of an application system. The behaviors of the application system are then the computational operations embodied onto the data objects. The data object modeling process is recognized as much more important and difficult than behaviors modeling, because the former is an open and creative process and it involves both real-world entities and their abstract representation on computing resources and their constraints. This section focuses on how data objects of a system are elicited, modeled, and constructed. Behavioral modeling and manipulation will be presented in Section 5.4.

5.3.1 TYPES AND DATA STRUCTURES

Using types to model the natural world can be traced back to the mathematical thought of Bertrand Russell and Godel [Schilpp, 1946; van Heijenoort, 1997]. Types are an important logical property shared by data objects in programming. Languages where variables should be declared in types are called *typed* languages. Most modern programming languages are typed and their compilers are capable to do static type checks to maintain consistency between declared and applied variables and data objects [Martin-Lof, 1975; Grune et al., 2000].

In computing, data in their most primitive form are a string of bits. Therefore, types are not an innate property of data at the level of physical representation and implementation. However, types are found expressively convenient for data representation at the logical level in programming and software engineering.

To some extent, type theory is developed for modeling and manipulating data objects for programmers, language processors, and tool developers. Therefore, types are the most fundamental techniques for modeling data objects in software engineering. Another purpose of type theory is to prevent computational operations on incompatible operands. The knowledge of type theory can help software engineers to avoid both obvious and not so obvious pitfalls, and it can also improve regularity and orthogonality in language design.

5.3.1.1 Type Systems of Programming Languages

The maximum range of values that a variable can assume is a type, and a type is associated with a set of predefined or allowable operations. Methodologies of types and their properties have been defined in RTPA in Table 4.8 and refined in Table 5.7, where 17 primitive types in computing and software engineering have been elicited.

Definition 5.13 A *data type*, shortly a *type*, is a set in which all member data objects share a common logical property or attribute.

The usages of types are as follows:

- To determine allowable operations, e.g., Boolean variables can not be operated by arithmetic operations such as addition and subtraction. Integers can not be operated by logical operations such as union or intersection;
- To help elicit common properties of data;
- To classify data into basic categories;
- To interpret semantics of values of data;
- To direct physical representation and implementations of data objects in a computer;
- To separate logical models of data objects from their physical model's implementation details.

A type can be classified as *primitive* and *derived* (complex) types. The former are the most elemental types that cannot further divided into more simple ones; the latter are a compound form of multiple primitive types based on given rules, which will be discussed in Section 5.3.1.3. Most primitive types are provided by programming languages; while most user defined types are derived ones.

Definition 5.14 A *type system* specifies data object modeling and manipulation rules of a programming language, as that of a grammar system which specifies the program composing rules (grammar) of the language.

Some typical structures of type systems are modeled in Figs. 5.8 through 5.10 with the Pascal [Jensen, 1978; Louden, 1993], Java [Wiener and Pinson, 2000], and IDL [OMG, 2002] type structures, respectively. IDL stands for the Interface Description Language defined by the Object Management Group (OMG).

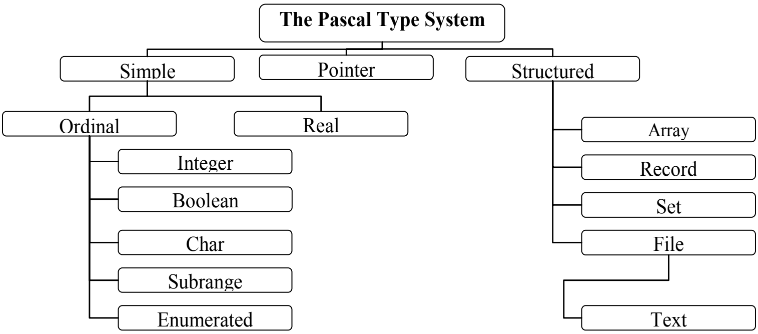


Figure 5.8 The type system of Pascal

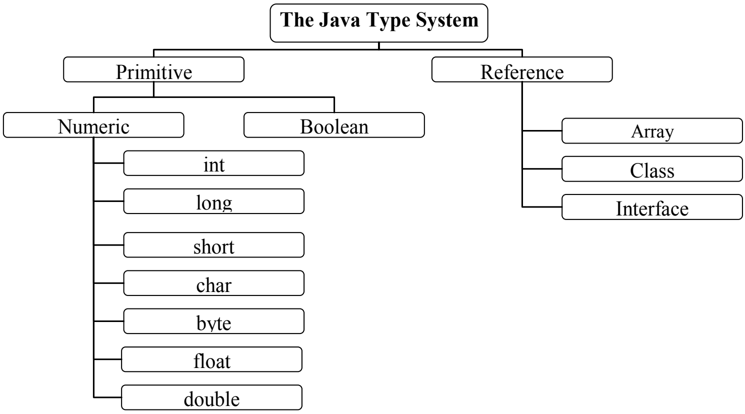


Figure 5.9 The type system of Java

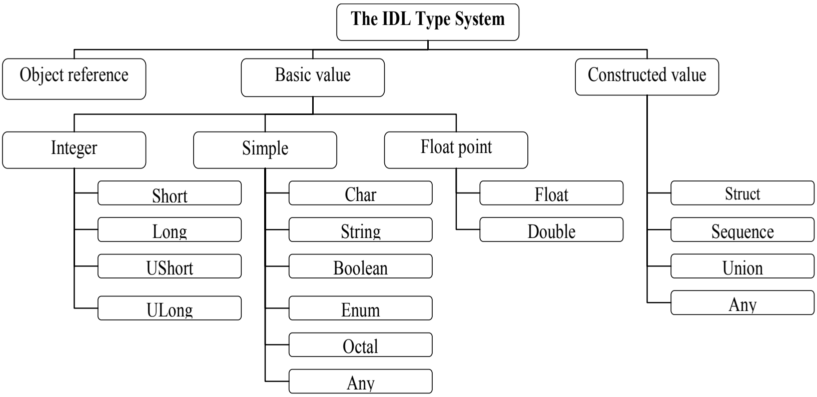


Figure 5.10 The type system of IDL

The RTPA type system defines the mathematical models of 17 primitive types and 11 abstract data types. The former will be discussed in the following subsection, and the latter will be described in Section 5.3.4 [Wang, 2002a].

5.3.1.2 Primitive Types

Via comparative studies of programming languages and formal specification methodologies, RTPA elicits 17 common and essential primitive types as presented in Section 4.6.3. The RTPA primitive types and their syntaxes and domains are described in Table 5.7, where the first 11 primitive types are for mathematical and logical manipulation of data objects, and the remaining 6 are for system control. In Table 5.7, D$_m$ is the *mathematical domain* of a type, D_l the *language defined domain*, where $D_l \subseteq D_m$, determined by the physical memory constraint in the implementation of a certain programming language. It is noteworthy that although a generic algorithm is constrained by D_m, an executable program is constrained by D_l or, at most of the time, by the *user designed domain* D_u, where $D_u \subseteq D_l$.

The 16th Law of Software Engineering

Theorem 5.6 The *domain constraints of data objects* state that to let D_m, D_l, and D_u be the domains of *mathematical* (logical), *language defined,* and *user defined,* respectively, the following relationship between the domains of an identifier in programming is always held, i.e.:

$$D_u \subseteq D_l \subseteq D_m \tag{5.19}$$

According to Theorem 5.6, the following corollary can be derived.

Corollary 5.3 The *precedence of domain determination* in programming and software engineering is always:

$$D_u \Rightarrow D_l \Rightarrow D_m \tag{5.20}$$

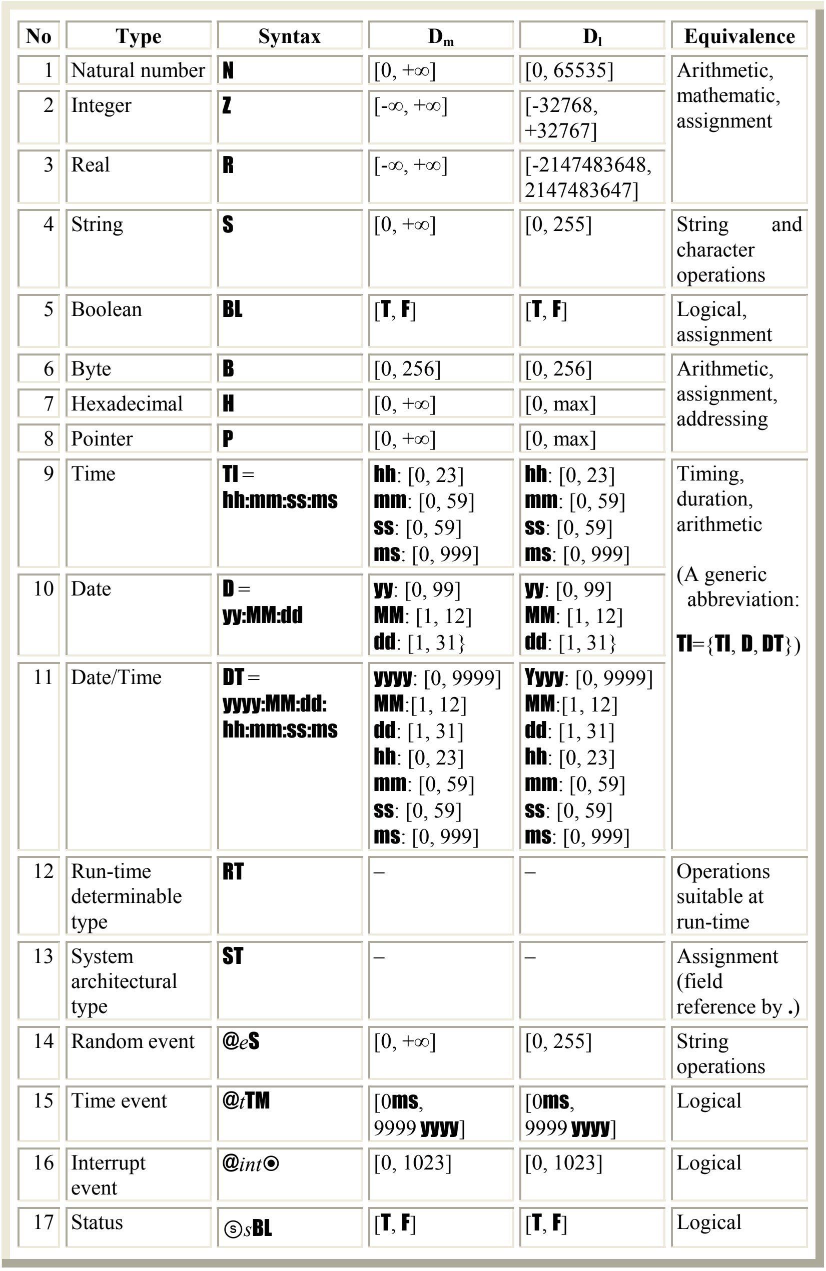
Most data objects modeling errors are various violations of the type domain rules as expressed in Eq. 5.19 and 5.20. Throughout this book, a type suffix convention is adopted for all identifiers as described below in order to avoid these fundamental problems in software engineering.

Definition 5.15 The *type suffix convention* denotes every variable x declared in a type $\mathbf{T}$, $x : \mathbf{T}$, by a bold type label attached to the variable in all invocations, i.e.:

$$x : \mathbf{T} \;\Rightarrow\; x\mathbf{T} \tag{5.21}$$

where $\mathbf{T}$ is any valid primitive or derived type as defined in Table 5.7.

Table 5.7
RTPA Primitive Types and Their Domains



No	Type	Syntax	D_m	D_l	Equivalence
1	Natural number	**N**	[0, +∞]	[0, 65535]	Arithmetic, mathematic, assignment
2	Integer	**Z**	[-∞, +∞]	[-32768, +32767]	
3	Real	**R**	[-∞, +∞]	[-2147483648, 2147483647]	
4	String	**S**	[0, +∞]	[0, 255]	String and character operations
5	Boolean	**BL**	[**T**, **F**]	[**T**, **F**]	Logical, assignment
6	Byte	**B**	[0, 256]	[0, 256]	Arithmetic, assignment, addressing
7	Hexadecimal	**H**	[0, +∞]	[0, max]	
8	Pointer	**P**	[0, +∞]	[0, max]	
9	Time	**TI** = **hh:mm:ss:ms**	**hh**: [0, 23] **mm**: [0, 59] **ss**: [0, 59] **ms**: [0, 999]	**hh**: [0, 23] **mm**: [0, 59] **ss**: [0, 59] **ms**: [0, 999]	Timing, duration, arithmetic (A generic abbreviation: **TI**={**TI**, **D**, **DT**})
10	Date	**D** = **yy:MM:dd**	**yy**: [0, 99] **MM**: [1, 12] **dd**: [1, 31}	**yy**: [0, 99] **MM**: [1, 12] **dd**: [1, 31}	
11	Date/Time	**DT** = **yyyy:MM:dd: hh:mm:ss:ms**	**yyyy**: [0, 9999] **MM**:[1, 12] **dd**: [1, 31] **hh**: [0, 23] **mm**: [0, 59] **ss**: [0, 59] **ms**: [0, 999]	**Yyyy**: [0, 9999] **MM**:[1, 12] **dd**: [1, 31] **hh**: [0, 23] **mm**: [0, 59] **ss**: [0, 59] **ms**: [0, 999]	
12	Run-time determinable type	**RT**	–	–	Operations suitable at run-time
13	System architectural type	**ST**	–	–	Assignment (field reference by **.**)
14	Random event	**@***e***S**	[0, +∞]	[0, 255]	String operations
15	Time event	**@***t***TM**	[0**ms**, 9999 **yyyy**]	[0**ms**, 9999 **yyyy**]	Logical
16	Interrupt event	**@***int*⊙	[0, 1023]	[0, 1023]	Logical
17	Status	ⓢ*s***BL**	[**T**, **F**]	[**T**, **F**]	Logical

The type suffix convention as adopted in RTPA is a convenient notation for both programmers and language processors. One of the most important advances of the type suffix convention is the improvement of readability – the key attributes in designing a programming language. Using the type suffixes, programmers may easily identify if all variables in a statement or expression are equivalent or compatible without referring to an earlier declaration that are scattered in a program across hundreds of pages in a large software. The convention also greatly simplifies type checking requirements during parsing the RTPA specifications [Wang, 2002a].

5.3.1.3 Derived and Advanced Types

The most common and powerful derived type shared by all programming languages is a *record*, also known as a *construct* in some languages. System architectures can be modeled on the basis of structured records. There are also a number of special advanced types for computing, such as the *system* types, dynamic *run-time* types, and *event, interrupt,* and *status* types. This subsection discusses those important derived types and their composing rules. User definable complex type in terms of ADTs will be described in Section 5.3.4.

5.3.1.3.1 Dynamic Run-Time Types

Definition 5.16 The *run-time type* **RT** is a nondeterministic type at compile-time that can be dynamically bound during run-time with one of the predefined primitive types.

The run-time type **RT** provides programmers a powerful tool to express and handle highly flexible and nondeterministic computing objects in data modeling. Some language such as Java and IDL [OMG, 2002] label the dynamic type **RT** as *anytype*, for which a specific type may be bound until run-time.

For example, referring to Section 4.7.2 on architectural specification and refinement in RTPA, data objects in a generic CLM schema may be specified in the type **RT**, for flexibility, while in the CLM objects, these data objects specified in **RT** will be instantiated in specific primitive types.

5.3.1.3.2 Time Types

According to Table 5.7, the time types including data, time, and date/time, are a special property of all computational systems, particularly real-time systems.

Definition 5.17 A *time type* **TM** is a complex type with a set of structural segments in **N** that models absolute (calendar-based) or relative (system-based) date and/or time, i.e.:

$$\begin{aligned}\mathbf{TM} &= \mathbf{D} \mid \mathbf{TI} \mid \mathbf{DT} \\ &= \mathbf{hh{:}mm{:}ss{:}ms} \\ &\quad \mid \mathbf{yy{:}MM{:}dd} \\ &\quad \mid \mathbf{yyyy{:}MM{:}dd{:}\ hh{:}mm{:}ss{:}ms}\end{aligned} \tag{5.22}$$

where the scope of each **TM** segment is a natural number as defined in Table 5.7.

5.3.1.3.3 Event Types

Definition 5.18 An *event* is an advanced type in computing that captures the occurring of a predefined external or internal change of status, such as an action of users, an external change of environment, and an internal change of the value of a specific variable.

The event types of RTPA can be classified into operational ($@e\mathbf{S}$), time ($@t\mathbf{TM}$), and interrupt ($@int⦿$) events as shown in Table 5.8, where @ is the *event prefix*, and **S**, **TM**, and ⊙ the type suffixes, respectively.

Table 5.8
Event Types of RTPA

No	Type	Syntax	Usage in system dispatch	Category
1	Operational event	$@e\mathbf{S}$	$@e_i\mathbf{S} \hookrightarrow_t P_i$	External or internal
2	Time event	$@t\mathbf{TM}$	$@t_i\mathbf{TM} \hookrightarrow_e P_i$	Internal
3	Interrupt event	$@int⦿$	$@int_j⦿ \hookrightarrow_i P_j$	External or internal

The *interrupt event* is a kind of special event that models the interruption of an executing process and the temporal handover of controls to an Interrupt Service Routine (ISR) resuming till its completion. In a real-time environment, an ISR should just conduct the most necessary functions and must be short enough compared with the time slice scheduled for a normal process.

5.3.1.3.4 Status Types

Definition 5.19 A *status* is an advanced type in computing that models the Boolean result of an execution of a process or a logical assertion of a given state in a process.

A status *s* is denoted by ⓢs**BL** in RTPA as described in Section 4.6.3, where ⓢ is the *status prefix*, and **BL** the Boolean type suffix for all statuses.

5.3.1.4 System Architectural Types

A special set of complex types known as the system type **ST** is widely used for modeling system architectures, particularly real-time, embedded, and distributed systems architectures. These requirements as identified in RTPA are such as system components, system processes, memory, I/O ports, device interfaces, interrupt sources, real-time events, and communication sockets. All the system types are nontrivial data objects in computing, rather than simple data or logical objects, which play a very important role in the whole lifecycle of complex system development including design, modeling, specification, refinement, comprehension, implementation, and maintenance of such systems.

5.3.1.4.1 The System Type

Definition 5.20 A *system type* **ST** is a system architectural type that models the architectural components of the system and their relations.

A generic **ST** type is CLM, which has been introduced in Definitions 4.106 and 4.108. CLMs are an abstract model of a system architectural component that represents a hardware interface, an internal logical model, and/or a common control structure of a system.

5.3.1.4.2 The System Memory Type

All logical identifiers and data objects, no matter language generated or user created, should be implemented as physical data objects and be bound to specific memory locations. This subsection explores the memory models of computing, mathematical models of addressing, and dynamic memory allocations.

Definition 5.21 The generic *system memory model*, MEM**ST**, can be described as a system architectural type **ST** with a finite linear space, i.e.:

$$\text{MEM}\mathbf{ST} \triangleq [addr_1\mathbf{H} \ldots addr_2\mathbf{H}]\mathbf{RT} \tag{5.23}$$

where $\text{addr}_1\mathbf{H}$ and $\text{addr}_2\mathbf{H}$ are the start and end addresses of the memory space, and **RT** is the type of each of the memory elements.

The entire memory space of a computer is typically divided into four general categories known as the *system area, static area, stack,* and *heap* as illustrated in Fig. 5.11.

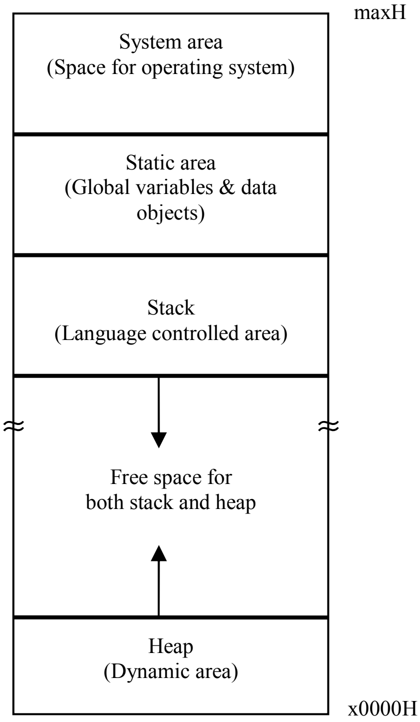


Figure 5.11 The logical memory model of computing

The *system area* is totally controlled by the operating system, and users have no access to this area. The *static area* is a language controlled area where global variables and data objects such as CLMs are allocated.

The *stack* is a language (compiler) controlled area for storing local and intermediate variables associated with embedded program blocks such as a method, function, or procedure in a program hierarchy. When a block completes its execution, all local variables scoped within it will be popped up and eliminated from the system.

The *heap* is a user controlled memory area reserved for dynamic memory allocation during run-time. A programmer may allocate or release a block of memory in heap by using special instructions like *new*(*x*) and *dispose*(x) in C++. Variables allocated in heap can be accessed by pointers or indirect addressing.

Note that in a typical implementation, the stack and heap grow in opposite directions, thus they may share the unallocated working memory efficiently. Further discussions on memory allocation and management can be found in Section 5.6.3.

5.3.1.4.3 The System Port Type

A special system architectural type is the I/O port type for modeling hardware architectures and interfaces.

Definition 5.22 The generic *system I/O port model*, PORT**ST**, can be described as a system architectural type **ST** with a finite linear space, i.e.:

$$\text{PORT}\mathbf{ST} \triangleq [ptr_1\mathbf{H} \ldots ptr_2\mathbf{H}]\mathbf{RT} \qquad (5.24)$$

where ptr$_1$**H** and ptr$_2$**H** are the start and end addresses of the port space, and **RT** is the type of each of the port I/O interfaces.

Examples of I/O CLMs may be referred to Section 4.7.2 and Example 4.22. More rigorous description of computational type systems in general, and the type rules of RTPA in particular, will be presented in Section 5.3.3.

5.3.2 BASIC DATA MODELING TECHNIQUES

As identified in Section 5.2, the most fundamental computing models and data objects in computing is bits. Therefore, it is at the center of all fundamental computing techniques to focus how real-world entities and their relations are represented and modeled by a set of given data structures and construct rules in computing and programming.

This subsection describes basic data modeling techniques for identifiers, variables, constants, expressions, memory models, and physical data objects. The modeling of advanced data structures, type theory, and abstract data types will be discussed in Sections 5.3.3 and 5.3.4.

5.3.2.1 Identifiers

Definition 5.23 An *identifier ID* is a logical name of a language entity or construct, which can be formally defined by a 7-tuple, i.e.:

$$ID \triangleq (N, \mathbb{T}, D, V, L, S, A) \qquad (5.25)$$

where the 7 attributes of an ID can be defined as follows:

- *N* is a representative symbol or name of the ID.
- $\mathbb{T}$ is the type of the ID, $\mathbb{T} \in \mathfrak{T}$ = {**N**, **Z**, **R**, **S**, **BL**, **B**, **H**, **P**, **TI**, **D**, **DT**, **RT**, **ST**, @*e***S**, @*t***TM**, @*int*⊙,ⓢ*s***BL**}, which is one of the 17 primitive types of RTPA according to Theorem 4.4.
- *D* is the domain of the ID, or the scope of its value.
- *V* is an instant value of the ID valid within the scope of values defined for the type.
- *L* is the physical location of the ID in the memory space *M*.
- *S* is the scope of life-span of the ID.
- *A* is the scope of accessibility or visibility of the ID.

Identifiers may be used to represent variables, constants, procedures, classes, or program names at different levels in programming. Definition 5.23 provides a comprehensive set of characteristics of an ID. Any ID in computing can be uniquely identified and allocated using Eq. 5.25. Important characteristics of IDs are described below.

The type of identifiers, $\mathbb{T}$, can be language provided, user defined, or one of the *system types* such as a program, a class, a procedure, or a CLM.

Definition 5.24 *Binding* is a process that associates an attribute to an identifier.

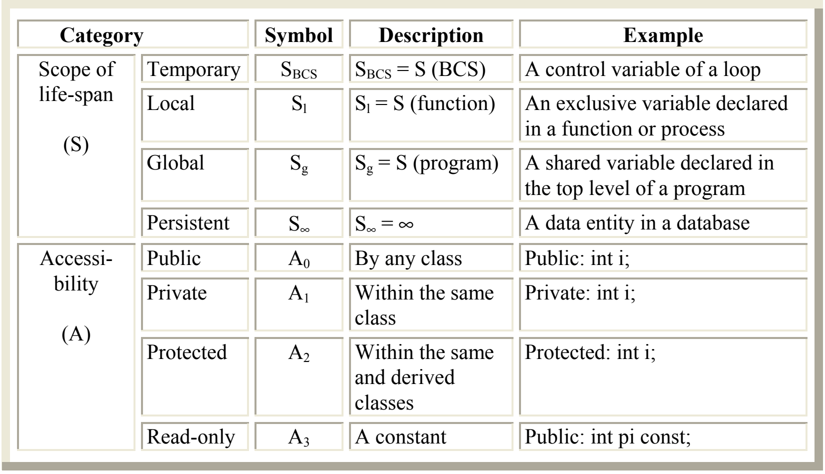
Bindings may be implemented either during execution or prior to execution of a statement with the identifier. The former are called *dynamic binding*, and the latter are called *static binding*. Static bindings are widely used to define new identifiers in imperative languages, where "the rule of declaration before use" is adopted. Dynamic bindings are used in assignments and dynamic memory allocations.

The scope of an identifier *C* as specified in Definition 5.23 can be defined below.

Definition 5.25 The *scope* of an identifier *ID* is a region in a program over which the binding between the *ID* and a given attribute is declared.

The scope of an identifier can be *temporary, local*, *global*, or *persistent* as shown in Table 5.9. Most IDs possess a local scope, except those declared at the top-level of a program. The persistent identifiers or variables have a longer lifecycle than the program that created them. This type of identifier can be found in file systems, data bases, and communication systems where the identifiers and related data are stored in an external storage such as a hard disk rather than the internal memory.

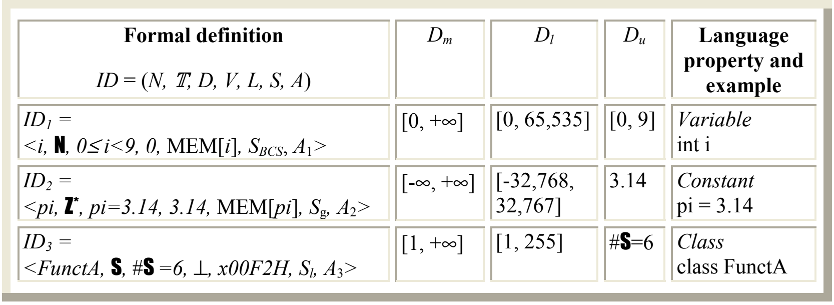
Table 5.9
Classifications of Scope of Life-Span and Accessibility of Variables



Category		Symbol	Description	Example
Scope of life-span (S)	Temporary	S_{BCS}	S_{BCS} = S (BCS)	A control variable of a loop
	Local	S_l	S_l = S (function)	An exclusive variable declared in a function or process
	Global	S_g	S_g = S (program)	A shared variable declared in the top level of a program
	Persistent	S_∞	$S_\infty = \infty$	A data entity in a database
Accessi-bility (A)	Public	A_0	By any class	Public: int i;
	Private	A_1	Within the same class	Private: int i;
	Protected	A_2	Within the same and derived classes	Protected: int i;
	Read-only	A_3	A constant	Public: int pi const;

Example 5.7 Table 5.10 characterizes three formally defined identifiers and their application examples in programming languages.

Table 5.10
Formal Definition of Identifiers



Formal definition $ID = (N, \mathbb{T}, D, V, L, S, A)$	D_m	D_l	D_u	**Language property and example**
$ID_1 =$ $<i, \mathbf{N}, 0 \leq i < 9, 0,$ MEM[i], $S_{BCS}, A_1>$	$[0, +\infty]$	[0, 65,535]	[0, 9]	*Variable* int i
$ID_2 =$ $<pi, \mathbf{Z}^*, pi=3.14, 3.14,$ MEM[pi], $S_g, A_2>$	$[-\infty, +\infty]$	[-32,768, 32,767]	3.14	*Constant* pi = 3.14
$ID_3 =$ $<FunctA, \mathbf{S}, \#\mathbf{S} = 6, \perp, x00F2H, S_l, A_3>$	$[1, +\infty]$	[1, 255]	$\#\mathbf{S}$=6	*Class* class FunctA

According to the formal definition and the illustrations of Table 5.10, identifiers using the same representation symbol would be treated differently when any of the other attributes is different except that of V. In other words, Eq. 5.25, i.e., $ID = (N, \mathbb{T}, D, V, L, S, A)$, essentially specifies a unique identifier in a program.

5.3.2.2 Variables and Constants

Real-world entities and their attributes can be abstracted and identified by symbols or identifiers. If an identifier can be quantified by a fixed value in its given scope, it is a constant; otherwise, it is a variable. Therefore, the descriptions of variables and constants share the same basis as for identifiers developed in the previous subsections.

Definition 5.26 A *variable v* is an identifier that its set of value V is multiple and changeable within the given domain D of type $\mathbb{T}$, i.e.:

$$\begin{aligned} v &= ID_v \\ &= (N, \mathbb{T}, D, V, L, S, A), \quad \#V > 1 \end{aligned} \tag{5.26}$$

A variable obtains or changes its value through the operation of assignments.

Definition 5.27 An *assignment* is an operation in programming that transfers a value q to a variable x, when their types are the same or equivalent, denoted by:

$$x\mathbb{T} := q\mathbb{T} \tag{5.27}$$

where $\mathbb{T}$ is the type suffix. Note that q can be a number, constant, or the value of an expression.

Definition 5.28 A *constant c* is an identifier that its set of value V is fixed with only one read-only value within the given domain D of type $\mathbb{T}$*, i.e.:

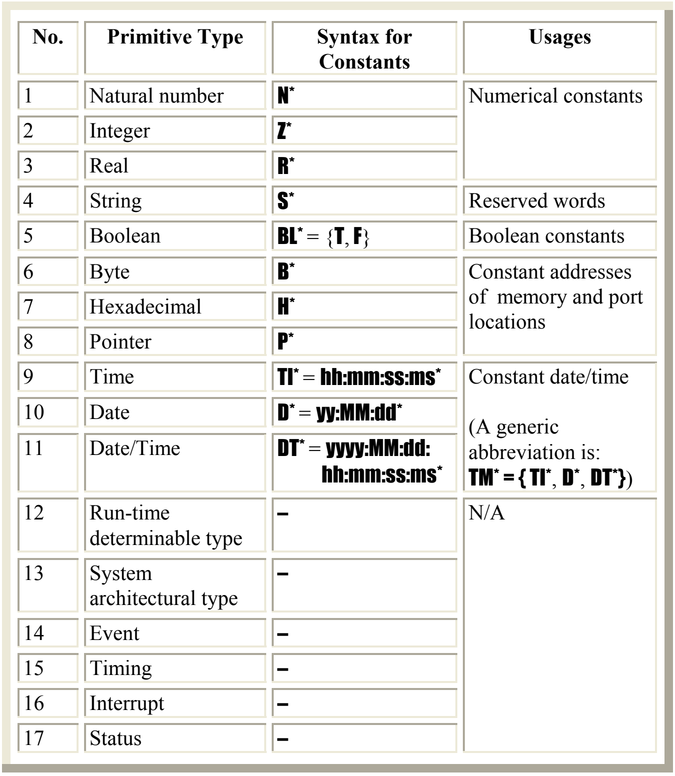
$$\begin{aligned} c &= ID_c \\ &= (N, \mathbb{T}^*, D, V, L, S, A), \quad \#V \equiv 1 \end{aligned} \tag{5.28}$$

where $\mathbb{T}$* shows that type $\mathbb{T}$ is a constant type.

Unlike a variable, a constant obtains its value through *declaration* rather than assignment, and the binding between a constant identifier and its value is fixed, which can not be changed by any operations.

According to Definitions 5.28 and 5.23, constants as those of variables can be specified in various types. That is, there are numerical constants such as $c_1 = 1$ ($\mathbb{T} = \mathbf{Z}^*$) and $c_2 = 3.14159$ ($\mathbb{T} = \mathbf{R}^*$), as well as Boolean constants such as $c_3 = \mathbf{T}$ ($\mathbb{T} = \mathbf{BL}^*$) and $c_4 = \mathbf{F}$ ($\mathbb{T} = \mathbf{BL}^*$). Therefore, most of the types $\mathbb{T}$ as defined in Table 5.7 have a corresponding form for constants $\mathbb{T}^*$ as given in Table 5.11. This convention is an extension of existing type theory, which provides additional expressive power to model constants as special data objects in computing and software engineering.

Table 5.11
Types of Constants and Their Usages



No.	Primitive Type	Syntax for Constants	Usages
1	Natural number	$\mathbf{N}^*$	Numerical constants
2	Integer	$\mathbf{Z}^*$	
3	Real	$\mathbf{R}^*$	
4	String	$\mathbf{S}^*$	Reserved words
5	Boolean	$\mathbf{BL}^* = \{\mathbf{T}, \mathbf{F}\}$	Boolean constants
6	Byte	$\mathbf{B}^*$	Constant addresses of memory and port locations
7	Hexadecimal	$\mathbf{H}^*$	
8	Pointer	$\mathbf{P}^*$	
9	Time	$\mathbf{TI}^* = \mathbf{hh{:}mm{:}ss{:}ms}^*$	Constant date/time (A generic abbreviation is: $\mathbf{TM}^* = \{\mathbf{TI}^*, \mathbf{D}^*, \mathbf{DT}^*\}$)
10	Date	$\mathbf{D}^* = \mathbf{yy{:}MM{:}dd}^*$	
11	Date/Time	$\mathbf{DT}^* = \mathbf{yyyy{:}MM{:}dd{:}\ hh{:}mm{:}ss{:}ms}^*$	
12	Run-time determinable type	–	N/A
13	System architectural type	–	
14	Event	–	
15	Timing	–	
16	Interrupt	–	
17	Status	–	

5.3.2.3 Expressions

In programming, an expression is a basic formula for building meaningful syntactic entities that may be used in evaluation of its semantic values.

Definition 5.29 An *expression exp* is a relation between a set of operands (variables or constants) $O = \{o_1, o_2, ..., o_n\}$ that is formed by a set of operators $R = \{r_1, r_2, ..., r_m\}$, i.e.:

$$\begin{aligned} exp &= O \times R \times O \\ &= \{o_i\, r_k\, o_j\},\ o_i, o_j \in O \wedge r_k \in R \end{aligned} \tag{5.29}$$

where *R* can be the arithmetical, logical, memory manipulation, or data manipulation operators.

An expression can be classified as *logical, ordinal, numerical, timing,* and *architectural*, according to the type of its value in **BL**, **N**, **R/Z/S/B/H/P**, **TM**, and **ST**, respectively.

Expressions as a language building block will discussed in Section 6.3 on formal language theory.

5.3.3 FORMAL TYPE THEORY

A type system specifies the data objects composing rules of a programming language as that of a grammar system which specifies the behavioral composing rules of the language. The basic *properties* of type systems are decidable, transparent, and enforceable [Martin-Lof, 1975; Cardelli and Wegner, 1985; Mitchell, 1990; Nordstrom et al., 1990; Cardelli, 1997; Pierce, 2002]. Type systems should be *decidable* by a type checking system that can ensure that types of variables are both well-declared and referred. Type systems should be *transparent* that diagnose reasons for inconsistency between variables or variables and their declarations. Type systems should be *enforceable* in order to check type inconsistence as much as possible.

5.3.3.1 Type Rules

A type is a category of variables that share a common property such as kinds of data, domain, and allowable operations. A formal rule of types is a mathematical relation and constraint on a given type. Type rules are defined on the basis of a type environment.

Definition 5.30 A *type environment* Θ_t is a collection of all primitive types in the given programming language or formal notation system, i.e.:

$$\begin{aligned}\Theta_t &= \mathfrak{T} \\ &= \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @e\mathbf{S}, @t\mathbf{TM}, @int\odot, ⓢs\mathbf{BL}\} \quad (5.30)\end{aligned}$$

where $\mathfrak{T}$ is the set of primitive types defined in the given notation system, i.e., RTPA.

The description of a type rule can be expressed by a formal statement called a judgment in Θ_t.

Definition 5.31 A *judgment* σ is an assertion *A* yielded in a given type environment Θ_t, denoted by:

$$\sigma \triangleq \Theta_t \vdash A \tag{5.31}$$

where it reads that Θ_t yields *A*, or *A* is declared in Θ_t .

The forms of the assertions *A* vary from judgment to judgment, but all the variables of *A* must be declared in Θ_t. For example, the following assertions are valid judgments:

- *T* is declared as a *type* in Θ_t: $\Theta_t \vdash T$ (5.32)
- *ID* is declared as a *variable* of type *T* in Θ_t: $\Theta_t \vdash ID : T$ (5.33)
- *D* is declared as a *signature S* in Θ_t: $\Theta_t \vdash D \triangleq S$ (5.34)

where Eq. 5.34 assigns a signature *S* to a *declaration D*, and *S* is essentially the type of a declaration such as *ID : T*.

Definition 5.32 A *type rule* is an assertion of the validity of the conclusion of a judgment on a type $\Theta_t \vdash A$ based on the inference of a number of *n* premise judgments $\Theta_t \vdash A_i$, $0 \le i \le n$, denoted by the following convention:

$$\frac{Premise(s)}{Conclusion} = \frac{\Theta_t \vdash A_1, \ldots, \Theta_t \vdash A_n}{\Theta_t \vdash A} \tag{5.35}$$

where the conclusion holds *iff* all of the premises are satisfied.

Definition 5.33 An *empty environment* is an axiom of reference rule that derives an empty judgment ◊, which is always valid with no premise, i.e.:

$$\frac{}{\Theta_t \vdash \lozenge} \tag{5.36}$$

or simply written without the horizontal line, i.e.: $\Theta_t \vdash \lozenge$.

Example 5.8 A type rule, *Val*(*n*), *n = 1, 2, ...,* can be derived based on the variable judgment, i.e.:

$$Val(n) \triangleq \frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash n : \mathbf{N}} \qquad (5.37)$$

where the rule asserts that the value of any numeral *n* is declared in the natural number type **N**, derived on the basis of an empty type environment.

Example 5.9 The type rule of assignment, *id* := *E*, can be derived based on the signature judgment, i.e.:

$$id \ := \ E \triangleq \frac{\Theta_t \vdash id \ : \mathbf{RT}, \Theta_t \vdash E \ : \mathbf{RT}}{\Theta_t \vdash (id \ := \ E) : \mathbf{RT}} \qquad (5.38)$$

where the rule asserts that an assignment is valid when the types in the both sides of the assignment are the same, and *E* is a constant, variable, or expression in the same type as that of *id*.

5.3.3.2 Formal Type Systems

Definition 5.34 A *formal type system* is a collection of all type rules in Θ_t for a given programming language or formal notation system.

The essential part of the formal type system of RTPA can be summarized in Table 5.12, where all of the 17 primitive types are rigorously derived by valid judgments on the basis of type rules of RTPA in its type environment Θ_t.

Similarly, the type systems of Pascal, Java, and IDL as shown in Figs. 5.8 through 5.10 can be rigorously described in the same approach. The formal definitions of these type systems are reserved as exercises for readers at the end of this chapter.

5.3.3.3 Complex Type Rules for the RTPA Derived Types

Complex and derived types of RTPA can be described by composed type rules based on those of the primitive types. As described in Chapter 4, there are two basic system modeling techniques in RTPA known as the process and CLM. The type rules of both CLMs and processes can be derived below.

Table 5.12
The Formal Type System of RTPA

No	Type	Syntax	Type Rule	Description
1	Natural number	**N**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{N}}$	A primitive type
2	Integer	**Z**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{Z}}$	*Ditto*
3	Real	**R**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{R}}$	*Ditto*
4	String	**S**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{S}}$	*Ditto*
5	Boolean	**BL**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{BL}}$	*Ditto*
6	Byte	**B**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{B}}$	*Ditto*
7	Hexadecimal	**H**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{H}}$	*Ditto*
8	Pointer	**P**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{P}}$	*Ditto*
9	Time	**TI** = **hh:mm:ss:ms**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{hh} : \mathbf{mm} : \mathbf{ss} : \mathbf{ms}}$	*Ditto*
10	Date	**D** = **yy:MM:dd**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{yy} : \mathbf{MM} : \mathbf{dd}}$	*Ditto*
11	Date/Time	**DT** = **yyyy:MM:dd: hh:mm:ss:ms**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{yyyy}: \mathbf{MM}: \mathbf{dd}: \mathbf{hh}: \mathbf{mm}: \mathbf{ss}}$	*Ditto*
12	Run-time determinable type	**RT**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{RT}}$	**RT** ∈ {**N**, **Z**, **R**, **S**, **BL**, **B**, **H**, **P**, **TI**, **D**, **DT**}
13	System architectural type	**ST**	$\frac{\Theta_t \vdash \Diamond}{\Theta_t \vdash \mathbf{ST}}$	**ST** is a CLM
14	Event	**@***e***S**	$\frac{\Theta_t \vdash @,\ \Theta_t \vdash \mathbf{S}}{\Theta_t \vdash @e : \mathbf{S}}$	A system variable
15	Timing	**@***t***TM**	$\frac{\Theta_t \vdash @,\ \Theta_t \vdash \mathbf{TM}}{\Theta_t \vdash @t : \mathbf{TM}}$	A system Variable **TM**={**TI**, **D**, **DT**}
16	Interrupt	**@***int*⦿	$\frac{\Theta_t \vdash @,\ \Theta_t \vdash \odot}{\Theta_t \vdash @int : \odot}$	A system variable
17	Status	ⓢ*s***BL**	$\frac{\Theta_t \vdash \textcircled{S},\ \Theta_t \vdash \mathbf{BL}}{\Theta_t \vdash \textcircled{S}s : \mathbf{BL}}$	A system variable

5.3.3.3.1 The Type Rules for CLMs in RTPA

CLM is a generic type for modeling and manipulating data objects and system architectures. A CLM is an abstract or logical model of a system component such as an internal data object, an external hardware device, and an interface between the system and its environment.

Definition 5.35 The type rule of a *CLM type*, CLM, is a complex system type **ST** in RTPA derived in Θ_t, i.e.:

$$\frac{\Theta_t \vdash \mathbf{ST}}{\Theta_t \vdash CLM : \mathbf{ST}} \tag{5.39}$$

The declaration of a variable, *ClmID*, with the CLM type can be denoted by using the following type rule:

$$\frac{\Theta_t \vdash \mathbf{ST},\ \Theta_t \vdash \mathbf{T},\ \Theta_t \vdash ClmID{:}\mathbf{ST},\ \Theta_t \vdash ID{:}\mathbf{T}}{\Theta_t \vdash ClmID\mathbf{ST} \triangleq ClmID\mathbf{S}\ ::\{\overset{n}{\underset{i=1}{R}} <ID_i{:}\mathbf{T}_i \mid \text{Constraint}(ID_i\mathbf{T}_i)>;\}} \tag{5.40}$$

where the *ClmID*$\mathbf{ST}$ is defined by the string type label *ClmID*$\mathbf{S}$ with an *n*-field record, each of them specifies a meta variable ID_i in type $\mathbf{T}_i$, and its constraints denoted by *Constraint*($ID_i\mathbf{T}_i$).

Example 5.10 On the basis of the CLM type rule, the declaration of a *system clock* as a CLM, *SysClock*$\mathbf{ST}$, is given in Fig. 5.12.

SysClock$\mathbf{ST}$ $\triangleq$ SysClock$\mathbf{S}$ ::
{ $<\S t_n : \mathbf{N} \mid 0 \le \S t_n\mathbf{N} \le 1M>$;
|| $<\S t : \mathbf{hh{:}mm{:}ss{:}ms} \mid 0{:}0{:}0{:}0 \le \S t\mathbf{hh{:}mm{:}ss{:}ms} \le 23{:}59{:}59{:}99>$;
|| $<MainClockPort : \mathbf{B} \mid MainClockPort\mathbf{B} = 00F1\mathbf{H}>$;
|| $<ClockInterval : \mathbf{N} \mid ClockInterval\mathbf{N} = 1\mathbf{ms}>$;
|| $<ClockIntCounter : \mathbf{N} \mid 0 \le ClockIntCounter\mathbf{N} \le 999>$
}

Figure 5.12 Specification of the architecture of system clock in RTPA

In Fig. 5.12, $\S t_n\mathbf{N}$ is the *relative* clock; and $\S t\mathbf{hh{:}mm{:}ss{:}ms}$ is the *absolute* clock. The system clock is driven by an external tick signal from port *MainClockPort*$\mathbf{B}$ at address 00F1$\mathbf{H}$ with an interval of 1ms. The *ClockIntCounter*$\mathbf{N}$ transfers every 1,000 pulses of *ClockInterval*$\mathbf{N}$ to a second inorder to update $\S t\mathbf{hh{:}mm{:}ss{:}ms}$. When it is needed, a long-range absolute *SysClock*$\mathbf{ST}$ may be specified using $\S t\mathbf{yyyy{:}MM{:}dd{:}hh{:}mm{:}ss{:}ms}$.

5.3.3.3.2 The Type Rules for Processes in RTPA

A *process* in RTPA is a basic behavioral unit for modeling software system operations onto the data objects. A process can be a meta process or a complex process composed with multiple meta processes by the relational process operators. Because processes are so frequently used in system modeling, a derived type in RTPA known as the process type can be introduced as a special system type.

Definition 5.36 The type rule of a *process type*, *Proc***ST**, is a complex system type **ST** in RTPA derived from Θ_t, i.e.:

$$\frac{\Theta_t \vdash \mathbf{ST}}{\Theta_t \vdash PROC : \mathbf{ST}} \tag{5.41}$$

The declaration of a variable, *ProcID*, with the process type can be denoted by using the following type rule:

$$\frac{\Theta_t \vdash ProcID\text{:}\,\mathbf{ST}}{\Theta_t \vdash ProcID\mathbf{ST} \triangleq ProcID\mathbf{S}} \tag{5.42}$$

$$(\mathrm{I}::<\overset{n}{\underset{i=1}{R}} ID_i\mathbf{T}_i>;\ \mathrm{O}::<\overset{m}{\underset{j=1}{R}} ID_j\mathbf{T}_j>;\ \mathrm{CLM}::<\overset{q}{\underset{k=1}{R}} ClmID_k\mathbf{ST}_k>)$$

where the *ProcID***ST** is defined by the string type label *ProcID***S** with a set of *n* inputs and a set of *m* outputs in a specific **T** type, as well as a set of *q* I/O constructs known as CLMs in a specific **ST** type.

Examples of process declarations and specifications will be provided in Section 5.5.1. Formal semantics of processes of RTPA in deductive semantics will be discussed in Chapter 6.

5.3.4 ABSTRACT DATA TYPES

Studies in algebraic specifications of software systems lead to the development of the concept on abstract data types [Gaudel, 1991]. The concept of ADT is proposed in Guttag's work [Guttag, 1975/77/02; Guttag and Horning, 1978]. More systematic description of ADTs may be found in Broy et al. (1984), Goguen (1978), and Louden (1993).

It is noteworthy that an ADT is not simply a type or complex type for data object modeling, rather than a behavioral modeling technique in computing. Because the purpose of ADTs is for encapsulation of predefined operations with related data objects, the emphases of ADT modeling techniques have been put on operational behaviors rather than expressive and comprehensive characterization of architectural data objects. However, for explaining the whole picture of data object modeling techniques in computing, ADTs are introduced in this subsection.

5.3.4.1 The Generic Model of ADTs

Definition 5.37 An *Abstract Data Type* (ADT) is a logical model of data objects, which defines both the logical architecture and valid operations of the data object, with the following schema:

$$\begin{array}{ll} \text{ADT\_ID}\mathbf{ST} \triangleq \text{ADT\_ID}\mathbf{S} :: & \\ & (\quad \text{Architecture} \\ & \quad \| \ \text{Static behaviors} \\ & \quad \| \ \text{Dynamic behaviors} \\ &) \end{array} \tag{5.43}$$

According to Definition 5.37, ADTs are abstract logical models of user defined data objects, where predefined operations on given data objects are a set of behavioral schemas (interfaces) rather than detailed implementations. Particular specifications of physical implementation of the ADT on different computer platforms are omitted.

Example 5.11 An ADT of arithmetic operations on $\mathbf{R}$, *, has been given in Example 4.12 in RTPA, i.e., $* : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$.

According to Definition 5.37, the architecture of the ADT * can be modeled as $* : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$. The static behaviors of * are all allowable operations, i.e., $+ : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$; $- : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$; $\bullet : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$; and $\div : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$. The dynamic behaviors of * are:

$$\begin{array}{l} \text{�}@op\mathbf{S}: \\ \quad + : +(\mathbf{R}, \mathbf{R})\mathbf{R} \\ \quad - : \ -(\mathbf{R}, \mathbf{R})\mathbf{R} \\ \quad \bullet : \ \bullet(\mathbf{R}, \mathbf{R})\mathbf{R} \\ \quad \div : \div(\mathbf{R}, \mathbf{R})\mathbf{R}, \ \forall r \in \mathbf{R}, \div\ (r, 0)\mathbf{R} \rightarrow \varepsilon \end{array}$$

where $@op\mathbf{S}$ represents the run-time selection of specific operations, and ε denotes an error.

ADTs possess the following properties:

- An extension of type constructions by integrating both data structures and functional behaviors.
- A hybrid data object modeling technique that *encapsulates* both user defined data structures (types) and allowable operations on them.

- The interface and implementation of an ADT are separated. Detailed implementation of the ADT is hidden to applications that invoke the ADT and its predefined operations. In other words, applications may only access the ADT as an abstract object as seen by the interfaces of the ADT.

It is noteworthy that a *class* in modern object-oriented programming [Stroustrup, 1986] can be perceived as an ADT with the properties of *encapsulation, abstraction, inheritance,* and *polymorphism*.

5.3.4.2 Modeling Complex Data Structures and Component Architectures by ADTs

Kenneth C. Louden (1993) described another form of ADT algebraic specifications with the following schema:

$$\begin{array}{l} \text{ADT}\mathbf{ST} \triangleq \text{ADT\_ID}\mathbf{S} :: \\ \qquad\qquad (\quad \text{architecture} \\ \qquad\qquad\quad \| \ \text{operations} \\ \qquad\qquad\quad \| \ \text{variables} \\ \qquad\qquad\quad \| \ \text{axioms} \\ \qquad\qquad) \end{array} \tag{5.44}$$

where the architecture is a brief date structure of the ADT and detailed properties of the ADT are specified by the axioms, and the remaining two parallel components are predefined operations and variables that represent instantiations of the architecture. Comparing Eqs. 5.43 and 5.44, it can be seen that the dynamic aspect of the ADT is not specified in the latter approach.

Example 5.12 A specification of a *stack* ADT according to Louden can be described as shown in Fig. 5.13.

Fig. 5.13 specifies that the *Stack* is an ADT with the architecture of the stack in type **ST** consisting of a set of element in type **RT**, where the specific type of **RT** will be instantiated during run-time. Predefined operations of the stack ADT, such as create, push, pop, and empty, are given in the operation section for the schemes, and in the axiom section for the instances. The Last-In First-Out (LIFO) behavior of the *Stack* is specified with additional information provided in the axioms of the *Stack*. Note that the architecture or data structure of the ADT is usually informally described in this approach, and the creation of the *Stack* is not well defined.

ADT **Stack** (element : **RT**) : **ST**

operations:
create: $\rightarrow$ stack
push: stack $\times$ element $\rightarrow$ stack
pop: stack $\rightarrow$ element
empty: stack $\rightarrow$ Boolean

variables:
s: stack;
e: element

axioms:
create (s) = s
push (s, e) = $\overset{\frown}{e\ s}$
pop (create (s)) = error
pop (push (s, e)) = e
empty (create (s)) = **T**
empty (push (s, e)) = **F**

Figure 5.13 A stack ADT

The operations specified in an ADT are a set of functions, which may be classified in the following categories:

- *Constructor*: An operation that its codomain is the type of the ADT.
- *Inspector*: An operation that its codomain is different from the type of the ADT. Inspectors can be either *predicates* that result in a Boolean output or *selectors* that result in a non-Boolean output.
- *Destructor*: An operation that its codomain is a subset of the type of the ADT.

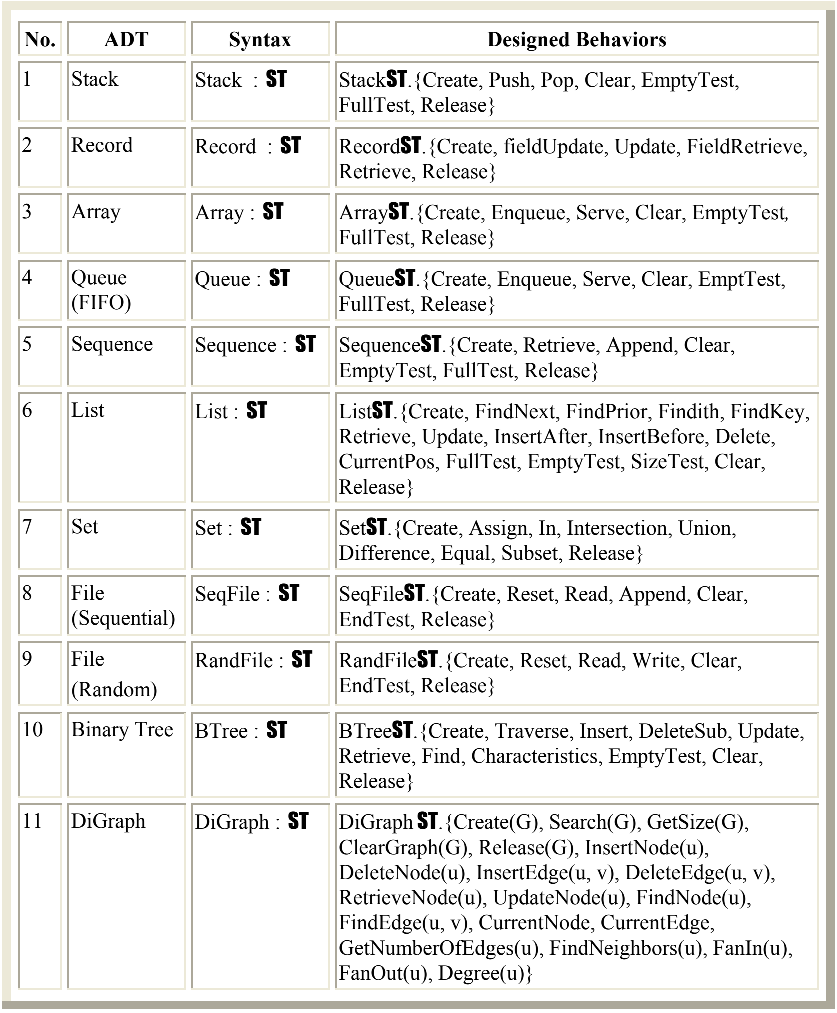
According to the above classification, the *create* and *push* operations of the stack in Example 5.12 are constructors, *pop* is a destructor, and *empty* is an inspector or more specifically a predicate.

5.3.4.3 Typical ADTs Modeled in RTPA

An ADT in RTPA is described as a logical model of derived data objects that possesses predefined operations on the logical model. Unlike the conventional approaches to ADT specifications that treat ADTs as static data types, ADTs are treated as dynamic entities in RTPA, which have both architectures and behaviors to server as both structural and operational models.

A set of 11 ADTs, which models typical and frequently used complex data objects in data structural and system architectural modeling, has been predefined in RTPA as shown in Table 5.13. The ADTs, which are developed recursively by using the RTPA notation and primitive types, are a coherent part of the RTPA notation system. Users may use the ADTs and their designed behaviors in system specifications as those of the primitive types by directly invoking their structures and related operations.

Table 5.13
Abstract Data Types Defined in RTPA



No.	ADT	Syntax	Designed Behaviors
1	Stack	Stack : **ST**	Stack**ST**.{Create, Push, Pop, Clear, EmptyTest, FullTest, Release}
2	Record	Record : **ST**	Record**ST**.{Create, fieldUpdate, Update, FieldRetrieve, Retrieve, Release}
3	Array	Array : **ST**	Array**ST**.{Create, Enqueue, Serve, Clear, EmptyTest, FullTest, Release}
4	Queue (FIFO)	Queue : **ST**	Queue**ST**.{Create, Enqueue, Serve, Clear, EmptTest, FullTest, Release}
5	Sequence	Sequence : **ST**	Sequence**ST**.{Create, Retrieve, Append, Clear, EmptyTest, FullTest, Release}
6	List	List : **ST**	List**ST**.{Create, FindNext, FindPrior, Findith, FindKey, Retrieve, Update, InsertAfter, InsertBefore, Delete, CurrentPos, FullTest, EmptyTest, SizeTest, Clear, Release}
7	Set	Set : **ST**	Set**ST**.{Create, Assign, In, Intersection, Union, Difference, Equal, Subset, Release}
8	File (Sequential)	SeqFile : **ST**	SeqFile**ST**.{Create, Reset, Read, Append, Clear, EndTest, Release}
9	File (Random)	RandFile : **ST**	RandFile**ST**.{Create, Reset, Read, Write, Clear, EndTest, Release}
10	Binary Tree	BTree : **ST**	BTree**ST**.{Create, Traverse, Insert, DeleteSub, Update, Retrieve, Find, Characteristics, EmptyTest, Clear, Release}
11	DiGraph	DiGraph : **ST**	DiGraph **ST**.{Create(G), Search(G), GetSize(G), ClearGraph(G), Release(G), InsertNode(u), DeleteNode(u), InsertEdge(u, v), DeleteEdge(u, v), RetrieveNode(u), UpdateNode(u), FindNode(u), FindEdge(u, v), CurrentNode, CurrentEdge, GetNumberOfEdges(u), FindNeighbors(u), FanIn(u), FanOut(u), Degree(u)}

In the RTPA specifications of the ADTs, three related perspectives of ADTs are described: the architecture, static behaviors, and dynamic behaviors as modeled in Definition 5.37. With the RTPA specification and refinement method, the features of ADTs as both static data types and dynamic behavioral objects or components can be specified formally and precisely.

Example 5.13 A *stack* ADT specified in RTPA is as shown in Fig. 5.14.

Stack**ST** ≜ Stack**ST**.Architecture
|| Stack**ST**.StaticBehaviors
|| Stack**ST**.DynamicBehaviors (5.45)

The architecture of the *Stack***ST** is specified by RTPA as shown in Eq. 5.12, where both the architectural CLM and an access model are provided for the *Stack***ST**.

StackST.Architecture ≜ CLM : **ST**
|| AccessModel : **ST**
|| Events : **S**
|| Status : **BL**

StackST.Architecture.CLM ≜ StackID**S** ::
(<Element : **RT**>,
<Size : **N** | Size**N** ≥ 0>,
<CurrentPos : **P** | 0 ≤ CurrentPos**P** ≤ Size**N**-1>
)

StackST.Architecture.AccessModel ≜ StackID**S**(CurrentPos**P**)**RT**

Figure 5.14 The architecture of the stack ADT

In Fig. 5.14, the *access model* of the *Stack***ST** is a logic model for supporting external invoking of the *Stack***ST** in operations, such as *push* and *pop*. The other parts of the model are designed for internal manipulations of the *Stack***ST**, such as creation, memory allocation, and release.

System static behaviors in RTPA describe the configuration of processes of the *Stack***ST** and their relations. The schemas of the seven static behaviors of the *Stack***ST** are specified as follows.

StackST.StaticBehaviors ≜
Stack.Create (<**I** :: StackInst**S**, SizeInst**N**, ElementInst**RT**>;
<**O**:: ⓢStackID.Allocated**BL**, ⓢStackID.Exist**BL**>)
| Stack.Push (<**I ::** StackInst**S**, ElementInst**RT**>; <**O::** ⓢStackID.Pushed**BL**>)
| Stack.Pop (<**I ::** StackInst**S**>; <**O::** ⓢStackID.Popped**BL**, Element**RT**>)
| Stack.Clear (<**I ::** StackInst**S**>; <**O::** ⓢStackID.Cleared**BL**>)
| Stack.EmptyTest (<**I ::** StackInst**S**>; <**O::** ⓢStackID.Empty**BL**>)
| Stack.FullTest (<**I ::** StackInst**S**>; <**O::** ⓢStackID.Full**BL**>)
| Stack.Release (<**I ::** StackInst**S**>; <**O::** ⓢStackID.Released**BL**>)

Figure 5.15 The static behaviors of the stack ADT

The refinement of detailed specifications of two static behaviors of Stack, *Stack.push* and *Stack.pop*, is given in Fig. 5.16.

```
Stack.PushST (<I:: <StackInstS, ElementInstRT>; <O:: ⓢStackID.PushedBL>) ≜
 {
       < StackIDS, ElementRT> := < StackInstS, ElementInstRT>
       → (� ⓢStackID.ExistBL = T
             → (  � (CurrentPosP^ < SizeN-1)
                     → ↑ (CurrentPosP^)
                     → ElementRT ⋖  StackIDS(CurrentPosP^)RT
                     → ⓢStackID.PushedBL := T
                 | � ~
                     → ⓢStackID.PushedBL := F
                     → ! (@’StackID.Full’)
                 )
         | � ~
             → ⓢStackID.PushedBL := F
             → ! (@’StackID.ExistBL = F’)
         )
 }

Stack.PopST (<I:: StackInstS>; <O:: <ⓢStackID.PoppedBL, ElementRT>) ≜
 {
       StackIDS := StackInstS
       → (� ⓢStackID.ExistBL = T
             → ( � (CurrentPosP^ > 0)
                     → StackIDS(CurrentPosP^)RT ⋗ ElementRT
                     → ↓ (CurrentPosP^)
                     → ⓢStackID.PoppedBL := T
                 | � ~
                     → ⓢStackID.PoppedBL := F
                     → ! (@’StackID.Empty’)
                 )
         | � ~
             → ⓢStackID.PoppedBL := F
             → ! (@’StackID.ExistBL = F’)
         )
  }
```

Figure 5.16 The specification of detailed behaviors of the stack ADT

According to the RTPA system modeling and refinement scheme as described in Section 4.7, the specifications of system static behaviors are only functional components of the system. To put the components into a live, coherent, and real-time system, the dynamic behaviors of the system, in terms of process deployment and dispatch, are yet to be specified. The dynamic behaviors of the *Stack*ST can be specified below in RTPA.

```
StackST.DynamicBehaviors ≙
 { ( @CreateStack  ↳ Stack.Create (<I:: StackInstS, SizeInstN, ElementInstRT>;
                                   <O:: ⓢStackID.AllocatedBL, ⓢStackID.ExistBL>)
   | @Push         ↳ Stack.Push (<I:: StackInstS, ElementInstRT>;
                                 <O:: ⓢStackID.PushedBL>)
   | @Pop          ↳ Stack.Pop (<I :: StackInstS>;
                                <O:: ⓢStackID.PoppedBL, ElementRT>)
   | @Clear        ↳ Stack.Clear (<I:: StackInstS>;  <O:: ⓢStackID.ClearedBL>)
   | @StackEmpty   ↳ Stack.EmptyTest (<I:: StackInstS>; <O:: ⓢStackID.EmptyBL>)
   | @StackFull    ↳ Stack.FullTest (<I:: StackInstS>;  <O:: ⓢStackID.FullBL>)
   | @ReleaseStack ↳ Stack.Release (<I:: StackInstS>;  <O:: ⓢStackID.ReleasedBL>)
   ) → ⊗
 }
```

Figure 5.17 The dynamic behaviors of the stack ADT

The process dispatch mechanism of the *Stack*ST as defined in Fig. 5.17 specifies detailed dynamic process relations at run-time by a set of *event-driven* relations.

On the basis of the above example, it is demonstrated that the important architectural and dynamic features of ADTs can be described by RTPA, including dynamic memory allocation, event and timing manipulation, exception detection, etc., which are hardly dealt with in other approaches to ADT specifications.

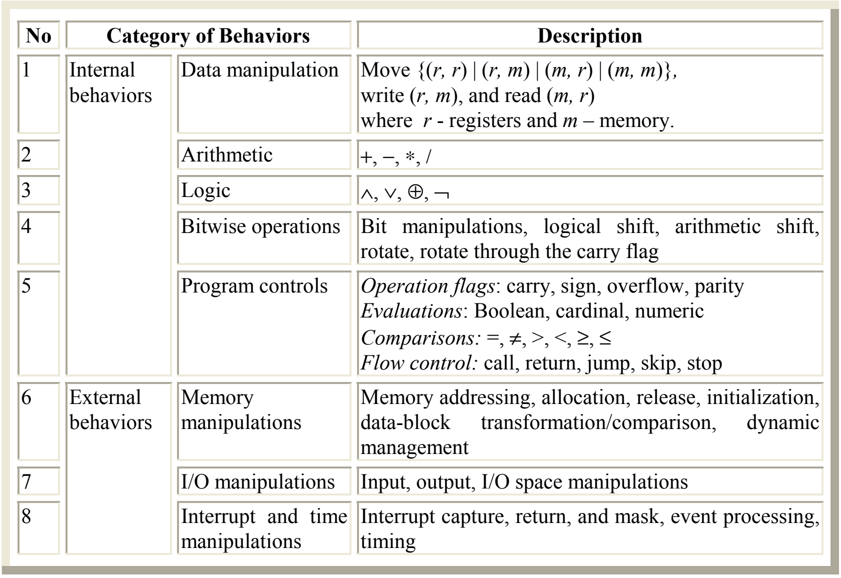
These 11 typical ADTs have also been used for the construction of the RTPA type library, which enables the ADTs to be reused in real-time system and non real-time specifications via RTPA [2002a].

5.4 Behavioral Modeling and Manipulation

Behaviors of programs and software systems are observable computing processes and operation consequences on the data objects modeled in the computing environment. On the basis of the discussions on data objects modeling and manipulation in the preceding section, this section describes how internal and interactive behaviors embodied on data objects in computing may be formally modeled and manipulated.

Built upon the VNA machines as described in Section 5.2.4, fundamental computing behaviors modeled by various instruction sets of computers can be classified into eight categories, such as *data manipulations, arithmetical operations, logical operations, bitwise operations, program controls, memory manipulations, I/O manipulations,* and *interrupt and time manipulations*, as shown in Table 5.14.

Table 5.14
Taxonomy of Fundamental Instructions in Computing



No	Category of Behaviors		Description
1	Internal behaviors	Data manipulation	Move {(*r, r*) | (*r, m*) | (*m, r*) | (*m, m*)}, write (*r, m*), and read (*m, r*) where *r* - registers and *m* – memory.
2		Arithmetic	+, –, ∗, /
3		Logic	∧, ∨, ⊕, ¬
4		Bitwise operations	Bit manipulations, logical shift, arithmetic shift, rotate, rotate through the carry flag
5		Program controls	*Operation flags*: carry, sign, overflow, parity *Evaluations*: Boolean, cardinal, numeric *Comparisons:* =, ≠, >, <, ≥, ≤ *Flow control:* call, return, jump, skip, stop
6	External behaviors	Memory manipulations	Memory addressing, allocation, release, initialization, data-block transformation/comparison, dynamic management
7		I/O manipulations	Input, output, I/O space manipulations
8		Interrupt and time manipulations	Interrupt capture, return, and mask, event processing, timing

The eight categories of fundamental computing behaviors defined on abstract data objects can be grouped into internal and external (interactive) behaviors [Mandrioli and Ghezzi, 1987]. Sections 5.4.1 and 5.4.2 will focus on the modeling and manipulation of internal behaviors. The modeling of external and interactive behaviors will be discussed in Section 5.4.3.

5.4.1 INTERNAL BEHAVIORS MODELING

Definition 5.38 The *internal behaviors* of a software system are computing operations and processes implemented on internal data objects contained in registers, cache, and the stack.

As shown in Table 5.14, the internal behaviors encompass those of data manipulation, arithmetic, logic, bitwise, and program controls. Because the first four categories of behaviors are intuitive, this subsection will put emphases on the program control mechanisms in computing and software engineering.

5.4.1.1 Basic Control Structures (BCS's)

Definition 5.39 *Basic Control Structures* (BCS's) are a set of essential flow control mechanisms that are used for constructing logical architectures of software systems.

The most commonly identified BCS's in computing and program languages are known as the *sequential, branch*, *iterations, recursion, function call, parallel,* and *interrupt* structures as shown in Table 5.15. These BCS's provide important compositional rules for programming. Based on them, complex computing functions and processes can be composed.

According to Table 4.9, it can be seen that the 10 BCS's identified in Table 5.15 are a subset of process relations $\mathfrak{R}$ as defined in RTPA, i.e.:

$$\begin{aligned} BCS = \{&\rightarrow, |, |\ldots|\ldots, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \| \; (\oiint), ↯\} \\ &\subseteq \mathfrak{R} \end{aligned} \tag{5.46}$$

where $\mathfrak{R}$ has been formally specified in Theorem 4.7 and explained in Sections 4.6.5, 6.6.2, and 6.6.3.

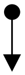
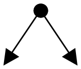
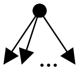
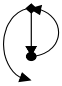
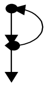
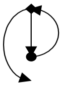
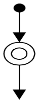
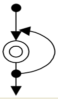
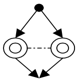
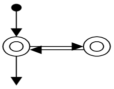
5.4.1.2 Control Flow Graphs

BCS's, or more general the RTPA process relations, model the fundamental flow control mechanisms and program composing rules in computing. To abstract the entire control structure of a program, the technique of control flow graph, which is a combinational representation of the digraph models of BCS's as modeled in Table 5.15, may be applied.

Definition 5.40 A *Control Flow Graph* (CFG) is a directed graph (*digraph*) model of program control structure, where a block of *sequential* instructions is abstractly represented by an edge, a *branch* BCS is denoted by two fan-out edges, and an *iteration* BCS is represented by a branch and sequential BCS's.

Example 5.14 A program, *MaxFinder*, is formally described in RTPA as shown in Fig. 5.18. Its function is to find the maximum number *max*$\mathbf{N}$ from a set of *n* inputted integers {X[1]$\mathbf{N}$, X[2]$\mathbf{N}$, …, X[n]$\mathbf{N}$}.

Table 5.15
BCS's and their Mathematical Models

Category	BCS	Notation	Structural model	RTPA model
Sequence	Sequence	$\rightarrow$		$P \rightarrow Q$
Branch	Branch	$\|$		�$exp\mathbf{BL} = \mathbf{T} \rightarrow P$ $\|$ �$\sim \rightarrow Q$
	Switch	$\|$ … $\|$		�$\text{exp}\mathbb{T}$ **=** $\text{i} \rightarrow P_i$ $\| \sim \rightarrow \oslash$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$
Iteration	While-loop	R^*		$\overset{\mathbf{F}}{\underset{\text{exp}\mathbf{BL}=\mathbf{T}}{R}} P$
	Repeat-loop	R^+		$P \rightarrow \overset{\mathbf{F}}{\underset{\text{exp}\mathbf{BL}=\mathbf{T}}{R}} P$
	For-loop	R^i		$\overset{n\mathbf{N}}{\underset{i\mathbf{N}=1}{R}} P(i\mathbf{M})$
Embedded component	Function Call	$\rightarrowtail$		$P \rightarrowtail F$
	Recursion	$\circlearrowleft$		$\overset{0}{\underset{i\mathbf{N}=n\mathbf{N}}{R}} P^{i\mathbf{M}} \circlearrowleft P^{i\mathbf{M}\text{-}1}$
Concurrence	Parallel	$\|\|$		$P \| \| Q$
	Interrupt	↯		P ↯ Q

```
MaxFinderST ({I:: X[1]N, X[2]N, …, X[n]N }; {O:: maxN }) ≙
{
   XmaxN := 0                                   // 1
   → $\overset{n\mathbf{N}}{\underset{i\mathbf{N}=0}{R}}$ (                                  // 2
            � X[iN]N > XmaxN                    // 3
               → XmaxN := X[iN]N                // 4
          )                                     // 5
   → maxN := XmaxN                              // 6
}
```

Figure 5.18 Formal description of the *MaxFinder* program

The corresponding CFG of this program is shown in Fig. 5.19, where the number label on a node refers to the instruction number marked in Fig. 5.18.

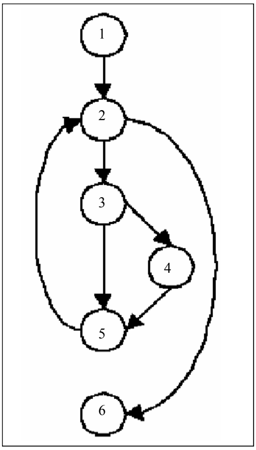


Figure 5.19 The CFG of the program MaxFinder

When a program is abstracted by a CFG, i.e., a problem is reduced to a digraph, well-defined graph theory can be used to analyze its properties and complexities. Examples will be given in Section 10.7.2.

5.4.2 ITERATIVE AND RECURSIVE BEHAVIORS MODELING

As modeled in Table 5.15, the interactive and recursive behaviors are an important part of the internal behaviors in computing. Iterative and recursive control structures are the most fundamental mechanisms of computing, because they make programming more effective and expressive. However, iteration constructs are perhaps the most diverse and confusable instructions in programming languages at both syntactic and semantic levels. Although a wide variety of notations have been proposed for describing iterations, there is still a lack of a unified mathematical notation that may be used to express the notion of repetitive, recursive, and predicative behaviors and architectures in computing.

When analyzing the syntactic and semantic problems inherent in iterations in programming, B.L. Meek concluded that: "There are some who argue that this demonstrates that the procedural approach to programming languages must be inadequate and fatally flawed, and that coping with something so fundamental as looping must therefore entail looking at computation in a different way rather than trying to devise better procedural syntax. There are others who would argue the possible applications of looping so it cannot simply be removed or obviated. As ever it is probably this last argument that will hold sway until (or unless) someone proves them wrong, whether with a brilliant stroke of procedural syntactic genius, or an effective and comprehensive new approach to the whole area [Meek, 1991]."

This section adopts the big-R notation [Wang, 2002a/06f] as developed in Section 4.5.3 as a unified mathematical means for representing and modeling iterations and recursions in computing. Based on the big-R notation, fundamental properties of iterative and recursive behaviors of software systems are comparatively analyzed.

5.4.2.1 Formal Description of Iterations

The importance of iterations in computing is rooted in the basic need for effectively describing recurrent and repetitive software behaviors and system architectures. However, unlike the high commonality in branch structures among programming languages, the syntaxes of loops are far more than unified. There is even a lack of common semantics of all forms of loops in modern programming languages.

Based on the inductive property of iterations, the big-R notation as defined in Eq. 4.59 is found to be a convenient means to describe all types of iterations including the while-, repeat-, and for-loops.

Definition 5.41 The *while loop* R^* is an iterative construct in which a process *P* is executed repeatedly as long as the conditional expression *exp*$\mathbf{BL}$ is true, i.e.:

$$
\begin{aligned}
\mathrm{R}^{*}P &\triangleq \overset{\mathbf{F}}{\underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{\mathrm{R}}}\ P\\
&= \gamma \bullet (\ \ \text{�}\ exp\mathbf{BL}=\mathbf{T}\\
&\qquad\qquad\quad \to P\\
&\qquad\qquad\quad \curvearrowright \gamma\\
&\qquad\quad |\ \text{�} \sim\\
&\qquad\qquad\quad \to \otimes\\
&\qquad)
\end{aligned}
\tag{5.47}
$$

where $\curvearrowright$ denotes a jump to a given label γ, $\otimes$ denotes the exit of the loop, and * denotes an iteration for 0 to n times, $n \geq 0$. That is, P may not be iterated in the while-loop at run-time if $exp\mathbf{BL} = \mathbf{F}$ at the very beginning.

According to Eq. 5.47, the semantics of the while-loop can be reduced to a series of repetitive conditional operations where the branch "? ~ → ∅" denotes an exit of the loop when $exp\mathbf{BL} \neq \mathbf{T}$. Note that the update of the control expression $exp\mathbf{BL}$ is not necessarily to be explicitly specified inside the body of the loop. In other words, the termination of the while-loop, or the change of $exp\mathbf{BL}$, can either be a result of internal effect of P or that of other external events.

Definition 5.42 The *repeat loop* R^{+} is an iterative construct in which a process P is executed repetitively for at least once until the conditional expression $exp\mathbf{BL} = \mathbf{F}$, i.e.:

$$
\begin{aligned}
\mathrm{R}^{+}P &\triangleq P \to \mathrm{R}^{*}P\\
&= P \to \overset{\mathbf{F}}{\underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{\mathrm{R}}}\ P\\
&= P \to \gamma \bullet (\ \text{�}\ \mathrm{exp}\mathbf{BL}=\mathbf{T}\\
&\qquad\qquad\qquad \to P\\
&\qquad\qquad\qquad \curvearrowright \gamma\\
&\qquad\qquad\ |\ \text{�} \sim\\
&\qquad\qquad\qquad \to \otimes\\
&\qquad\qquad)
\end{aligned}
\tag{5.48}
$$

where $^{+}$ denotes an iteration for 1 to n times, $n \geq 1$. That is, P will be executed at least once in the repeat loop until exp$\mathbf{BL} \neq \mathbf{T}$.

According to Eq. 5.48, the semantics of the repeat-loop is deduced to a single sequential operation of P plus a serial of repetitive conditional operations whenever exp$\mathbf{BL} = \mathbf{T}$. Or simply, the semantics of the repeat-loop is equivalent to a single sequential operation of P plus a while-loop of P.

In both Eqs. 5.47 and 5.48, the loop control variable $exp\mathbf{BL}$ is in the type Boolean. When a loop control variable i as an index is adopted in a numeric type, say in type $\mathbf{N}$ with known lower bound $n_1\mathbf{N}$ and upper bounds $n_2\mathbf{N}$, then a special variation of iteration, the *for* loop, can be derived below.

Definition 5.43 The *for loop* R^{i} is an iterative construct in which a process P indexed by an identification variable $i\mathbf{N}$, $P(i\mathbf{N})$, is executed repeatedly in the scope $n_1\mathbf{N} \le i\mathbf{N} \le n_2\mathbf{N}$, i.e.:

$$\begin{array}{l}
\mathrm{R}^{i}\, P(i\mathbf{N}) \triangleq \mathop{\mathrm{R}}\limits_{i\mathbf{N}=n_1\mathbf{N}}^{n_2\mathbf{N}} \mathrm{P}(i\mathbf{N}) \\
\quad = i\mathbf{N} := n_1\mathbf{N} \\
\qquad \to \gamma \bullet (\ \text{�}\ i\mathbf{N} \le n_2\mathbf{N} \\
\qquad\qquad\qquad \to P(i\mathbf{N}) \\
\qquad\qquad\qquad \to \uparrow(i\mathbf{N}) \\
\qquad\qquad\qquad \curvearrowright \gamma \\
\qquad\qquad\quad\ \mid \text{�}\ \sim \\
\qquad\qquad\qquad \to \otimes \\
\qquad\qquad\quad) \\
\quad = i\mathbf{N} := n_1\mathbf{N} \\
\qquad \to exp\mathbf{BL} = i\mathbf{N} \le n_2\mathbf{N} \\
\qquad \to \mathop{\mathrm{R}}\limits_{\exp\mathbf{BL}=\mathbf{T}}^{\mathbf{F}} (\ P(i\mathbf{N}) \\
\qquad\qquad\qquad \to \uparrow(i\mathbf{N}) \\
\qquad\qquad\quad)
\end{array} \qquad (5.49)$$

where $i\mathbf{N}$ denotes the loop control variable, and $\uparrow(i\mathbf{N})$ increases $i\mathbf{N}$ by one.

According to Eq. 5.49, the semantics of the *for* loop is a special case of while-loops where the loop control expression is $exp\mathbf{BL} = i\mathbf{N} \le n_2\mathbf{N}$, and the update of the control variable $i\mathbf{N}$ must be explicitly specified inside the body of the loop. In other words, the termination of the for-loop is internally controlled.

Based on Definition 5.42, the most simple *for* loop that iteratively executes $P(\mathrm{i}\mathbf{N})$ for k times, $1 \le i \le k$, can be derived as follows:

$$\mathop{\mathrm{R}}^{i} P(i\mathbb{N}) \triangleq \mathop{\mathrm{R}}_{i\mathbb{N}=1}^{k} P(i\mathbb{N}) \qquad (5.50)$$

It is noteworthy that a general assumption in Eqs. 5.49 and 5.50 is that *i* is a natural number and the iteration step $\Delta i\mathbb{N} = +1$. In a more generic situation, *i* may be an arbitrary integer $\mathbb{Z}$ or in other numerical types, and $\Delta i\mathbb{Z} \neq +1$. In this case, the lower bound of a for-loop can be described as an expression, or the incremental step $\Delta i\mathbb{Z}$ can be explicitly expressed inside the body of the loop, e.g.:

$$\begin{aligned} \mathop{\mathrm{R}}^{i} \mathrm{P}(i\mathbb{Z}) \triangleq \mathop{\mathrm{R}}_{i\mathbb{Z}=0}^{-10} (\ & \mathrm{P}(i\mathbb{Z}) \\ & \to i\mathbb{Z} := i\mathbb{Z} - \Delta i\mathbb{Z} \\ &) \end{aligned} \qquad (5.51)$$

where $\Delta i\mathbb{Z} \geq 1$.

5.4.2.2 Formal Description of Recursions

Recursion is a powerful tool in mathematics for a neat treatment of complex problems following a fundamental *deduction-then-induction* approach. Gödel, Herbrand, and Kleene developed the theory of recursive functions using an equational calculus in the 1930s [Kleene, 1952; McDermid, 1991]. More recent work on recursions in programming may be found in [Peter, 1967; Hermes, 1969; Hoare, 1985; Wilson and Clark, 1988]. The idea is to construct a class of effectively computable functions from a collection of base functions using fundamental techniques such as function composition and inductive referencing.

5.4.2.2.1 Properties of Recursions

Recursion is an operation that a process or function calls or refers to itself.

Definition 5.44 A *recursion* of process *P* can be defined by mathematical induction, i.e.:

$$\begin{aligned} &\mathrm{F}^0(P) = P, \\ &\mathrm{F}^1(P) = \mathrm{F}(\mathrm{F}^0(P)) = \mathrm{F}(P), \\ &\ldots \\ &\mathrm{F}^{n+1}(P) = \mathrm{F}(\mathrm{F}^n(P)), \ \ \mathrm{n} \geq 0 \end{aligned} \qquad (5.52)$$

A recursive process should be terminable or noncircular, i.e., the depth of recursive d_r must be finite. The following theorem guarantees that $d_r < \infty$ for a given recursive process or function [Lipschutz and Lipson, 1997].

Corollary 5.4 A recursive function is noncircular, i.e., $d_r < \infty$, *iff*:

a) A *base value* exists for certain arguments for which the function does not refer to itself;

b) In each recursion, the argument of the function must be closer to the base value.

Example 5.15 The factorial function can be recursively defined as shown in Eq. 5.53.

$$
\begin{array}{l}
(n\mathbf{N})! \triangleq \{ \\
\quad \text{�}\ n\mathbf{N} = 0 \\
\qquad \to (n\mathbf{N})! := 1 \\
\quad | \ \text{�} \sim \\
\qquad \to (n\mathbf{N})! := n\mathbf{N} \bullet (n\mathbf{N}\text{-}1)! \\
\quad \}
\end{array} \qquad (5.53)
$$

Example 5.16 A C++ implementation of the factorial algorithm as given in Example 5.15 is provided below.

```
int factorial (int n)
   {
     int factor;
     if (n==0)
        factor = 1;
     else factor = n * factorial(n-1);
     return factor;
   }
```

(5.54)

In addition to the usage of recursion for efficiently modeling repetitive behaviors of systems as above, it has also been found useful in modeling many fundamental language properties.

Example 5.17 Assume the following letters are used to represent the corresponding syntactic entities in the angler brackets:

P <program>,
L <statement list>,
S <statement>,
E <expression>,
I <identifier>,
A <letter>,
N <number>, and
D <digit>

The abstract syntax of grammar rules for a simple programming language may be recursively specified in BNF (see Section 6.3.6) as follows.

```
E ::=  E ‘+’ E
     | E ‘-‘ E
     | E ‘*’ E
     | ‘(‘ E ‘)’
     | I
     | N
I ::= I A | A
A ::= ‘a’ | ‘b’ | … | ‘z'
N ::= N D | D
D ::= ‘0’ | ‘1’ | … | ‘9’                    (5.55)
```

In Eq. 5.55 expression *E* is recursively defined by operations on *E* itself, an identifier *I*, or a number *N*. Further, *I* is recursively defined by itself and/or letter *A*; and *N* is recursively defined as itself and/or digit *D*. Since any form of *E* as specified above can be eventually deduced on terminal letters (‘a’, ‘b’, …, ‘z’), and digits (‘0’, ‘1’, …, ‘9’), or predefined operations (‘+’, ‘-‘, ‘*’, ‘(‘, ’)’), the BNF specification of *E* as shown in Eq. 5.55 is called well defined.

5.4.2.2.2 The Mathematical Model of Recursions

Definition 5.45 *Recursion* is an embedded process relation in which a process *P* calls itself. The recursive process relation can be denoted as follows:

$$\mathrm{P} \circlearrowleft \mathrm{P} \tag{5.56}$$

The mechanism of recursion is a series of embedding (*deductive*, denoted by $\circlearrowleft$) and de-embedding (*inductive*, denoted by $\circlearrowright$) processes. In the first phase of embedding, a given layer of nested process is deduced to a lower layer till it is embodied to a known value. In the second phase of de-embedding, the value of a higher layer process is induced by the lower layer starting from the base layer, where its value has already been known at the end of the embedding phase.

Recursion processes are frequently used in programming to simplify system structures and to specify neat and provable system functions. It is particularly useful when an infinite or run-time determinable specification has to be clearly expressed.

Instead of using self-calling in recursions, a more generic form of embedded construct that enables inter-process calls is known as the function call, $P \rightarrowtail Q$, as defined in RTPA, where the called process Q can be regarded as an embedded part of process P.

Using the big-R notation, a recursion can be defined formally as follows.

Definition 5.46 *Recursion* $\overset{\circlearrowleft}{\mathrm{R}}\, \mathrm{P}^i$ is a multi-layered embedded process relation in which a process P at layer i of embedment, P^i, calls itself at an inner layer $i\text{-}1$, $P^{i\text{-}1}$, $0 \leq i \leq n$. The termination of P^i depends on the termination of $P^{i\text{-}1}$ during its execution, i.e.:

$$\overset{\circlearrowleft}{\mathrm{R}}\, \mathrm{P}^i \triangleq \mathop{\mathrm{R}}\limits_{i\mathbf{N}=n\mathbf{N}}^{0} \begin{array}{l} (\ \text{�}\ i\mathbf{N} > 0 \\ \qquad \to \mathrm{P}^{i\mathbf{N}} := \mathrm{P}^{i\mathbf{N}\text{-}1} \\ \ \mid \text{�}\ \sim \\ \qquad \to \mathrm{P}^{0} \\ ) \end{array} \tag{5.57}$$

where n is the *depth of recursion* or embedment that is determined by an explicitly specified conditional expression $exp\mathbf{BL} = \mathbf{T}$ inside the body of P.

Example 5.18 Using the big-R notation, the recursive description of the algorithm provided in Example 5.15 can be improved as follows:

$$\begin{array}{rl} (n\mathbf{N})! \triangleq & \overset{\circlearrowleft}{\mathrm{R}}\ (n\mathbf{N})\ ! \\ = & \mathop{\mathrm{R}}\limits_{i\mathbf{N}=n\mathbf{N}}^{0} \begin{array}{l} (\ \text{�}\ i\mathbf{N} > 0 \\ \qquad \to (i\mathbf{N})! := i\mathbf{N} \bullet (i\mathbf{N}\text{-}1)! \\ \ \mid \text{�}\ \sim \\ \qquad \to (i\mathbf{N})! := 1 \\ ) \end{array} \end{array} \tag{5.58}$$

5.4.2.3 Comparative Analysis of Iterations and Recursions

In the literature, iterations were often mixed with recursions, or an iteration was perceived as a special type of recursion. Although, both iteration $\mathop{R}\limits^{i} P(i)$ and recursion $\mathop{R}\limits^{\circlearrowleft} P^i$ are repetitive and cyclic constructs in computing, the fundamental differences between their traces of execution at run-time are that the former is a linear structure, i.e.:

$$\mathop{R}\limits^{i} P(i) = P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \tag{5.59}$$

However, the latter is an embedded structure, i.e.:

$$\mathop{R}\limits^{\circlearrowleft} P^i = P^n \circlearrowleft P^{n-1} \circlearrowleft \ldots \circlearrowleft P^1 \circlearrowleft \mathrm{P}^0 \circlearrowright P^1 \circlearrowright \ldots \circlearrowright P^{n-1} \circlearrowright P^n \tag{5.60}$$

The generic forms of iterative and recursive constructs and their trace models in computing can be contrasted as illustrated in Figs. 5.20 and 5.21 as follows.

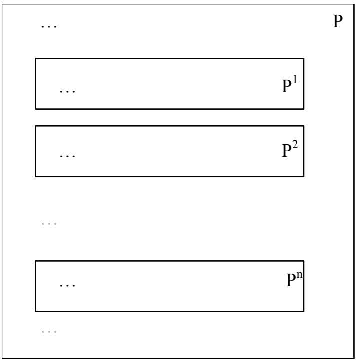


Figure 5.20 The linear architecture of iterations

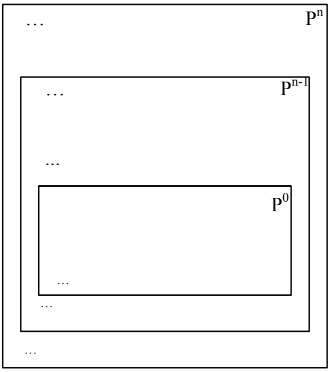


Figure 5.21 The nested architecture of recursions

It is noteworthy that there is always a pair of counterpart solutions for a given repetitive and cyclic problem with either the recursive or iteration approach. For instance, the corresponding iterative version of Example 5.15 can be described below.

Example 5.19 Applying the big-R notation, the iterative description of the algorithm as provided in Example 5.15 is shown below.

$$
\begin{array}{l}
(n\mathbf{N})! \triangleq \{ \\
\qquad \text{factorial}\mathbf{N} := 1 \\
\qquad \to \underset{i\mathbf{N}=1}{\overset{n\mathbf{N}}{R}} (\ \text{factorial}\mathbf{N} := i\mathbf{N} \bullet \text{factorial}\mathbf{N}) \\
\qquad \to (n\mathbf{N})! := \text{factorial}\mathbf{N} \\
\quad \}
\end{array}
\qquad (5.61)
$$

It is interesting to compare both formal algorithms of factorial with recursion and iteration as shown in Eqs. 5.58 and 5.61, respectively.

Example 5.20 On the basis of Example 5.19, an iterative implementation of Example 5.15 in C++ can be developed as follows.

```
int factorial (int n)
   {
     int factor = 1;
     for (int i = 1; i <= n ; i++)
          factor = i * factor;
     return factor;
   }                                    (5.62)
```

The above examples show the difference between the recursive and iterative techniques for implementing the same algorithm for repetitive and cyclic computation. Contrasting Examples 5.18 and 5.19, or Examples 5.16 and 5.20, it can be seen that the recursive solution for a given problem is usually more expressive, but less efficient in implementation in terms of time and space complexity, than its iterative counterpart. As Peter Deutsch, the creator of the GhostScript interpreter, put it: “To iterate is human, to recurse divine.”

The efficient treatment of repetitive and recurrent behaviors and architectures has been recognized as one of the most premier needs in computing. Case studies on the applications of the big-R notation as introduced in Section 4.5.3 in denoting iterative and recursive computing behaviors demonstrated in this subsection show that a convenient notation may dramatically reduce the difficulty and complexity in expressing the most frequently used and highly recurring concept and notion in computing.

5.4.3 EXTERNAL AND INTERACTIVE BEHAVIORS MODELING

Definition 5.47 The *external behaviors* of a software system are interactive computing operations and processes implemented on external data objects modeled in the memory and I/O space that represent the system architectural model and environment.

As shown in Table 5.14, interactive behaviors modeling in computing encompass memory manipulation, external interface (I/O) manipulation, operating event handling, timing event handling, interrupt handling, and exception handling.

5.4.3.1 Memory Manipulations

Memory manipulations are the most frequently used techniques of interactive behavioral modeling in computing, which deal with memory I/O operations on the system memory model $\mathrm{MEM}\mathbf{ST} \triangleq [addr_1\mathbf{H} \ldots addr_2\mathbf{H}]\mathbf{RT}$ as given in Definition 5.21.

Typical memory manipulation behaviors are memory allocation, memory release, read, and write, as modeled in RTPA meta processes (Table 4.8).

5.4.3.1.1 Modes of Addressing

As formally described in Definition 4.71, addressing is a function π: $id\mathbb{T} \to ptr\text{Þ}\mathbf{P}$ that maps a given logical $id\mathbb{T}$ into the physical memory block identified by $ptr\text{Þ}\mathbf{P}$ in $\mathrm{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n-1]\mathbb{T}$, and $\mathbb{T} \in \{\mathbf{P}, \mathbf{H}, \mathbf{N}, \mathbf{Z}\}$.

Addressing is one of the most important and special operations in computing, which is used to find the physical address of a logical data object represented by an identifier in the memory space. Addressing techniques can be classified as *absolute* and *relative*, where the letter can be further divided into *direct* and *indirect* addressing dependent on whether an address is directly provided or indirectly inferred. Typical addressing modes in computing can be summarized in Table 5.16, where a formula describes how a memory address in hexadecimal, $addr\mathbf{H}$, can be obtained based on available information in registers and offsets.

Table 5.16
Typical Addressing Modes in Computing

No.	Category	Mode	Description
	Absolute		[$addr\mathbf{H} = data\mathbf{H}$]
1		Immediate	$addr\mathbf{H}$ = PhysicalAddress**H**
	Relative		[$addr\mathbf{H} = exp\mathbf{H}$]
2	- Direct	Register	$addr\mathbf{H}$ = Register**H**
3		Offset	$addr\mathbf{H}$ = Register**H** + offset**H**
4		Index	$addr\mathbf{H}$ = Register**H** + Index**H**
5		Base	$addr\mathbf{H}$ = BseReg**H** + offset**H**
6		Segment	$addr\mathbf{H}$ = SegReg**H** + offset**H**
7	- Indirect	Register	$addr\mathbf{H}$ = (Register**H**)**H**
8		Offset	$addr\mathbf{H}$ = (Register**H**)**H** + offset**H**
9		Index	$addr\mathbf{H}$ = (Register**H**)**H** + Index**H**
10		Base	$addr\mathbf{H}$ = (BaseReg**H**)**H** + offset**H**
11		Segment	$addr\mathbf{H}$ = (SegReg**H**)**H** + offset**H**

In high-level programming languages, addressing is usually relative and logical. That is, the address of a given data object is referred to as a logical location rather than an absolute memory address. However, in many situations in computing, such as system specification, architectural modeling, real-time system development, dynamic memory manipulation, and operating system development, physical and absolute addressing are necessary.

In addition, physical port address manipulations are required for I/O space manipulations.

5.4.3.1.2 Memory Read and Write

As given in Definition 4.74, a memory *read* denoted by $\gtrdot$ is a meta process that gets data $x\mathbb{T}$ from a given memory location MEM[$ptr\mathbf{P}$], where $Ptr\mathbf{P}$ is a pointer that identifies the physical memory address, i.e., MEM[$ptr\mathbf{P}$]$\mathbb{T} \gtrdot x\mathbb{T}$, where $\mathbb{T} \in \mathcal{T}$.

As given in Definition 4.75, a memory *write process* denoted by $\lessdot$ is a meta process that puts data $x\mathbb{T}$ to a given memory location MEM[$ptr\mathbf{P}$], where $ptr\mathbf{P}$ is a pointer that identifies the physical memory address, i.e., $x\mathbb{T} \lessdot$ MEM[$ptr\mathbf{P}$]$\mathbb{T}$where $\mathbb{T} \in \mathcal{T}$.

5.4.3.1.3 Dynamic Memory Allocation

Typical memory allocations for almost all the primitive types of variables as modeled in Table 5.7, except **RT**, are static that are controlled by the system. That is, the logical name of a variable is permanently bound to a fixed element or a continuous block of the physical memory throughout its lifecycle and accessibility scope. The condition for enabling *static memory allocation* at compile-time is that the size of the given variable or its length of memory occupation is determinable. Otherwise, the memory allocation should be dynamic, and be postponed until run-time.

Definition 5.48 *Dynamic memory allocation* is a binding process that associates a logical name of a data object, usually consisting of multiple similar elements, with a series of inter-linked physical locations in the heap during run-time.

Example 5.21 A *generic digraph model* can be logically represented by a list of dynamically allocated nodes as shown in Fig. 5.22 using the doubly-linked list model.

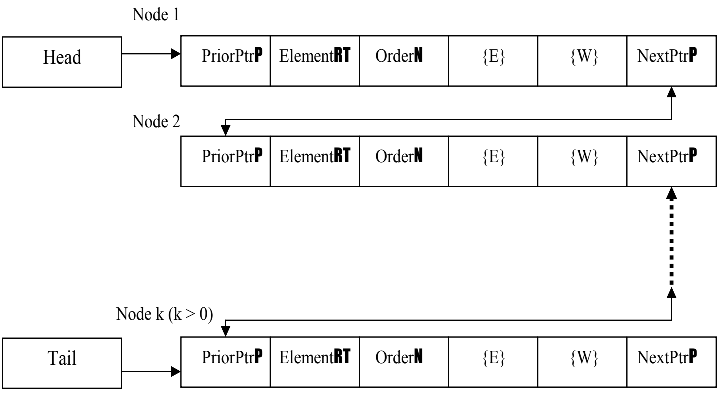


Figure 5.22 The architectural model of a digraph

In the doubly-linked list model of digraph, two global pointers, *head and tail*, pointing to a designated first and last node, respectively. The nodes are represented as a set of arrays, each of which points to a list of attributes consisting of two pointers (prior and next), the order (number of edges), a set of edges (*E*) in which the elements are node numbers connected to the given node, and a set of weights (*W*) denoting the weight of each corresponding edges.

A formal description of the architecture of the digraph in RTPA is shown in Fig. 5.23. The digraph CLM describes the abstract data structures of its doubly-linked list model. Each node in the digraph is modeled in DiGraph**ST**.Architecture.NodeCLM**ST**. In the node CLM, an *Element***RT** is a data field for accommodating information of specific applications, and is specified as a run-time determinable type in **RT**. Attached to each node is a set of edges $\mathop{\mathcal{R}}\limits_{i\mathbf{N}=1}^{Order\mathbf{N}} Edge(i\mathbf{N})\mathbf{S}$ depending on the value of *Order***N**, and the edges may be specified for a *Weight***N** corresponding to each of them in *E*. The

PriorPtr$\mathbf{P}$ and *NextPtr*$\mathbf{P}$ of a node specify the bidirectional links of a node. A complete specification of the DiGrapg static and dynamic behaviors may be referred to [Wang and Adewumi, 2004].

$$\begin{array}{l}
\textbf{DiGraph}\mathbf{ST}\textbf{.Architecture.CLM}\mathbf{ST} \triangleq \text{DiGraph}\mathbf{S} :: \\
(\ \ \text{<GSize: } \mathbf{N} \mid \text{GSize}\mathbf{N} \geq 1\text{>}, \qquad \text{// Number of Nodes} \\
\quad \mathop{\mathit{R}}\limits_{i\mathbf{N}=1}^{GSize\mathbf{N}} \text{<Node(i}\mathbf{N}\text{) : } \mathbf{ST}\text{>}, \\
\quad \text{<Head : } \mathbf{P}\text{>}, \\
\quad \text{<Tail : } \mathbf{P}\text{>} \\
) \\
\\
\textbf{DiGraph}\mathbf{ST}\textbf{.Architecture.NodeCLM}\mathbf{ST} \triangleq \text{Node}\mathbf{S} :: \\
(\ \ \text{<Element : } \mathbf{RT}\text{>}, \\
\quad \text{<PriorPtr : } \mathbf{P}\text{>}, \\
\quad \text{<NextPtr : } \mathbf{P}\text{>}, \\
\quad \text{<Order: } \mathbf{N} \mid 0 \leq \text{Order}\mathbf{N} \leq \text{SizeofEdges}\mathbf{N}\text{>}, \\
\quad < \mathop{\mathit{R}}\limits_{i\mathbf{N}=1}^{Order\mathbf{N}} \text{Edge(i}\mathbf{N}\text{) : } \mathbf{S}\text{>}, \\
\quad < \mathop{\mathit{R}}\limits_{i\mathbf{N}=1}^{Order\mathbf{N}} \text{Weight(i}\mathbf{N}\text{) : } \mathbf{N}\text{>}, \\
)
\end{array}$$

Figure 5.23 Formal specification of the architecture of the digraph

The key difference between dynamic memory allocation and static memory allocation is whether the size of memory requirement of a given variable is *run-time* vs. *compile-time* determinable. This is in line with the definitions of dynamic and static behaviors of processes as recognized in Section 4.7.1. Another difference is the mechanisms of their physical implementations, where dynamic memory allocation uses the *user-controllable heap* that is a user controlled memory area as shown in Fig. 5.11; while static memory allocations uses *language-controlled stacks* in system memory.

Dynamically allocated variables and data objects in the heap can be accessed by pointers or indirect addressing. As described in Definition 4.72, memory allocation is an inverse function of addressing, i.e., π^{-1}: ptrÞ$\mathbf{P} \rightarrow id\mathbb{T}$, that associates a physical memory block MEM[$ptr\mathbf{P}$, $ptr\mathbf{P}+n-1$]$\mathbb{T}$ with the given logical $id\mathbb{T}$.

Because dynamic allocated variables and complex data objects reside in the user-controllable heap, a dynamic memory allocation and its release should be carried out by explicit instructions provided by users. Typical dynamic memory allocation and release instructions are as follows:

$$
\begin{array}{lll}
\text{a)} & \text{C++:} & (ptr\mathbf{P} = new\ \mathbb{T},\ \ delete\ ptr\mathbf{P}) \\
\text{b)} & \text{C:} & (ptr\mathbf{P} = malloc(n),\ \ free(ptr\mathbf{P})) \\
\text{c)} & \text{Java:} & (obj\mathbf{C} = new\ object\mathbf{C}(),\ \ system.gc())
\end{array}
\qquad (5.63)
$$

where in Java, *gc* stands for *garbage collection* and **C** denotes a type of class.

Memory allocation is a key meta process for dynamic memory manipulation in RTPA. The memory allocation process in RTPA, $id\mathbb{T} \Leftarrow \text{MEM}[ptr\mathbf{P}, ptr\mathbf{P}+n\text{-}1]\mathbb{T}$, is implemented in Fig. 5.24.

```
MemoryAllocation (<I :: ObjectIDS, NofElementsN, ElementTypeRT >;
                         <O:: ⓢObjectID.ExistedBL>) ≙
{
 nN := NofElementsN
   $\overset{n}{\underset{i=1}{R}}$ (New ObjectID(iN) : ElementTypeRT)
 → ( � ⓢObjectAllocatedBL := T
         → ⓢObjectID.ExistedBL = T
     | � ~
         → ⓢObjectID.ExistedBL = F
   )
}
```

Figure 5.24 Implementation of dynamic memory allocation process in RTPA

Memory release is a process that dissociates a given memory block from a logical identifier $id\mathbb{T}$, and returns the memory block to the system through a mechanism known as system garbage collection. The memory release process of RTAP, $id\mathbb{T} \not\Leftarrow \text{MEM}[\bot]\mathbb{T}$, is implemented in Fig. 5.25.

```
MemoryRelease (<I :: ObjectIDS>; <O:: ⓢObjectID.ReleasedBL>) ≙
{
  delete ObjectIDS                  // System.GarbageCollection( )
  → ObjectIDS := null
  → (� ⓢObjectReleasedBL := T
         → ⓢObjectID.ReleasedBL = T
     | � ~
         → ⓢObjectID.ReleasedBL = F
    )
}
```

Figure 5.25 Implementation of the memory release process in RTPA

The released memory block that was logically identified by $id\mathbf{S}$ will then be collected by the system garbage management mechanism provided by an operating system.

5.4.3.2 Events Handling

Events capture and handling are important behaviors of all open systems, particularly for real-time systems. As given in Definition 5.18, the event types in computing can be classified into operational, time, and interrupt events as shown in Table 5.17, where @ is the event prefix, and $\mathbf{S}$, $\mathbf{TM}$, and $\odot$ the type suffixes. The operational events occur randomly. A special kind of operational events is the exception events. Another special type of system events may be classified as the interrupt events.

Table 5.17
Event Types of RTPA

No	Type	Syntax	Usage in system dispatch	Category
1	Operational event	$@e\mathbf{S}$	$@e_i\mathbf{S} \hookrightarrow_e P_i$	Internal or external
2	Time event	$@t\mathbf{TM}$	$@t_i\mathbf{TM} \hookrightarrow_t P_i$	Internal
3	Interrupt event	$@int\odot$	$@int_j\odot \hookrightarrow_i P_j$	External or internal

5.4.3.2.1 Operating Event Handling

As given in Definition 5.102, an *event-driven dispatch* behavior of software system, denoted by $\hookrightarrow_e$, is a process relation in which the ith process P_i is triggered by a predesignated system event $@e_i\mathbf{S}$, i.e., $@e_i\mathbf{S} \hookrightarrow_e P_i$, $i \in \{1, \ldots, n\}$.

5.4.3.2.2 Time Event Handling

As given in Definition 4.78, *absolute timing event* manipulation known as *timing*, denoted by $\underline{\underline{@}}$, is a meta process that sets the value of a timing variable $@t\mathbf{TM}$ as the absolute time of the current system clock $\S\mathrm{t}\mathbf{TM}$, i.e., $@\mathrm{t}\mathbf{TM}\ \underline{\underline{@}}\ \S\mathrm{t}\mathbf{TM}$, where $\mathbf{TM}$ is an abbreviation of $\mathbf{TI} = \mathbf{hh{:}mm{:}ss{:}ms}$, $\mathbf{D} = \mathbf{yy{:}MM{:}dd}$, and $\mathbf{DT} = \mathbf{yyyy{:}MM{:}dd{:}hh{:}mm{:}ss{:}ms}$, respectively.

Similarly, the *related timing event* manipulation known as *duration* given in Definition 4.79, denoted by $\triangleq$, is a meta process that sets a relative time $@\mathrm{t}_\mathrm{n}\mathbf{Z}$ as an integer based on the relative system clock $\S t_n\mathbf{Z}$ and the given period $\Delta n\mathbf{Z}$, i.e., $@t_n\mathbf{TM} \triangleq \S t_n\mathbf{TM} + \Delta n\mathbf{N}$, where the unit of all relative timing variables is $\mathbf{ms}$.

As given in Definition 4.101, a *time-driven dispatch* behavior of software system, denoted by ↳$_t$, is a process relation, in which the *i*th process P_i is triggered by a predefined system time $@t_i\mathbf{TM}$, i.e., $@t_i\mathbf{TM}$ ↳$_t$ P_i, $i \in \{1, ..., n\}$.

5.4.3.2.3 Interrupt Event Handling

The interrupt mechanism describes execution priority and control taking-over between processes. As given in Definition 4.100, an *interrupt*, denoted by ↯, is a process relation in which a running process P is temporarily held before termination by another higher priority process Q on interrupt event $@e\odot$ at the interrupt point $\odot$, and the interrupted process will be resumed when the high priority process has been completed, i.e., P ↯ $Q = P \,||\, (@int\odot \nearrow Q \searrow \odot)$, where $\nearrow$ and $\searrow$ denote *interrupt service* and *interrupt return*.

As given in Definition 4.103, an *interrupt-driven dispatch*, denoted by ↳ *i*, is a process relation in which the *i*th process P_i is triggered by a predefined system interrupt $@int_i\odot$, i.e., $@int_i\odot$ ↳ P_i, $i \in \{1, …, n\}$.

5.4.3.2.4 Exceptional Event Handling

As given in Definition 4.82, an *exception detection*, denoted by !, is a meta process that logs a detected exception event $@e\mathbf{S}$ at run-time, i.e., $!(@e\mathbf{S})$. The RTPA exception detection mechanism is a fundamental process for safety and dependable system specification, which enables system exception detection, handling, or postmortem analysis to be implemented.

5.5 Program Modeling: Coordination of Computational Behaviors with Data Objects

Program modeling provides various encapsulation methodologies for integrating and coordinating computing behaviors and data objects into a coherent system. Since the scale of a program can be very large, methodologies for program modeling, construction, and refinement play important roles in programming and software engineering.

5.5.1 THE UNIFIED MATHEMATICAL MODEL OF PROGRAMS

The concept of program was treated as for granted in computing and software engineering. Although there are various perceptions of programs, a rigorous and generic mathematical model of programs, the key object under study in software engineering, is yet to be sought.

The mathematical models of programs and software can be described and analyzed at various composition levels, such as those of *statement*, *process*, and *system* from bottom-up, according to the hierarchical architecture of the program. It is noteworthy that a statement is the *minimum functional unit* of programs at the most fundamental level of programming. If the mathematical models of all fundamental instructions (known as the *meta processes* in RTPA) [Wang, 2002a/02b/03c/06a/07a] and their relational composition rules (known as the *process relations* in RTPA) in a given language can be defined, the mathematical models of the process and program at the higher levels can be derived and established inductively.

5.5.1.1 The Abstract Model of Statements

A statement as an instance of an instruction in a programming language is the smallest functional unit of a program that specifies an explicit action and results in the change of one or more data objects logically modeled by variables.

Definition 5.49 A *statement s* in a program is an instantiation of a meta instruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

Definition 5.50 A *generic model of a statement s* in computing can be described as a function *p*, that maps a set of inputs *I* into a set of outputs *O*, i.e.:

$$s \triangleq p: I \to O \tag{5.64}$$

The above IPO model of statements can be illustrated as shown in Fig. 5.26. A statement is usually a relational function between a variable on the left-hand side and an expression on the right-hand site.

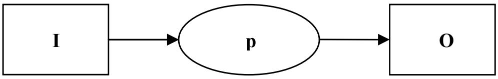


Figure 5.26 A statement as an IPO process

A set of 17 fundamental instructions in computing, as shown in Table 4.8 and Theorem 4.6, has been identified and elicited in RTPA known as the meta processes. Although existing programming languages may implement a larger set of instructions, the additional ones are logical combinations of the 17 essential meta processes.

5.5.1.2 The Abstract Model of Processes

A process at the middle or component level of the program hierarchy is composed by individual statements with given rules of algebraic compositions.

Definition 5.51 A *process P* is a composed component in a program that forms a logical combination of *n* meta statements s_i and s_j, $1 \leq i < n$, $1 < j \leq m$, according to certain composing relations r_{ij}, i.e.:

$$\begin{aligned} P &= \mathop{\mathcal{R}}_{i=1}^{n-1}(s_i \; r_{ij} \; \mathrm{s}_j), j = i+1 \\ &= (...(((s_1) \; r_{12} \; \mathrm{s}_2) \; r_{23} \; \mathrm{s}_3) \; ... \; r_{n-1,n} \; \mathrm{s}_n) \end{aligned} \qquad (5.65)$$

where r_{ij} is a set of algebraic relations or composing rules.

A comprehensive set of those composing rules has been modeled in RTPA known as process relations as described in Table 4.9 and Theorem 4.7.

5.5.1.3 The Abstract Model of Programs

Definition 5.52 A *program* $\wp$ is a triple of a finite list of instructive statements *S* that describes the computational behaviors, a set of data objects *D* that model the internal states and external environment, and their interrelations *R*, i.e.:

$$\wp = (S, D, R) \qquad (5.66)$$

The above definition puts emphases on the algebraic model of programs. A dynamic structural model of programs can be described below.

Definition 5.53 A *program* $\wp$ is a composition of a finite set of k processes at the component level according to certain process dispatching rules, i.e.:

$$\wp = \mathop{R}\limits_{i=1}^{k} (@e_i\mathbf{S} \hookrightarrow P_i) \tag{5.67}$$

where ↳ denotes a process dispatch according to a predesignated event $@e_i\mathbf{S}$, which may be an external, a system timing, or an interrupt event.

Formal descriptions of the event-, time-, and interrupt-driven dispatching mechanisms of program systems are described in Section 5.4.3.2.

The process in Definition 5.53 has been formally described in Theorem 4.3 by the *cumulative relational model of processes*. Based on Theorem 4.3, the cumulative relational model of programs can be derived in the following theorem.

The 17th Law of Software Engineering

Theorem 5.7 The *generic mathematical model of programs* states that a software system or a program $\wp$ is a set of complex embedded cumulative relational processes P_k dispatched by system-level events e_k, i.e.:

$$\begin{aligned} \wp &= \mathop{R}\limits_{k=1}^{m} (@e_k\mathbf{S} \hookrightarrow P_k) \\ &= \mathop{R}\limits_{k=1}^{m} [@e_k\mathbf{S} \hookrightarrow \mathop{R}\limits_{i=1}^{n-1} (s_i(k)\ r_{ij}(k)\ \mathrm{s}_j(k))],\ j = i+1 \end{aligned} \tag{5.68}$$

Theorem 5.7 provides a unified mathematical model for programs [Wang, 2006h], which reveals that a program is a finite and nonempty set of embedded binary relations between a current statement and *all previous ones* that formed the *semantic context* or environment of computing. It also reveals that the nature of programs is a cumulative (nonlinear) relational composition of a set of finite meta processes. The law indicates that no program is context-free, because every statement is relational to the consequences of all its previous statements that form the context of the given statement in a program and constitute the semantics of the given execution.

According to Theorem 5.7, a program can be reduced to the composition of a finite set of k processes at the component level. Then, each of the processes can be further reduced to the composition of a finite set of n statements at the bottom level. The definitions, syntaxes, and formal semantics of each of the meta processes and process relations may be referred to RTPA [Wang, 2002a/02b/03c/06a/07a]. A complex process and a program can be derived from the meta-processes by the set of algebraic process relations.

5.5.2 PROGRAMS MODELING AT COMPONENT LEVEL

Typical program modeling methodologies at the component and system levels can be classified into algorithms, classes, patterns, and frameworks from the bottom up. This subsection presents the first three models in the component level of the program hierarchy. System level models in terms of software frameworks will be discussed in Section 5.5.3.

5.5.2.1 Algorithms

An algorithm is a computational construct that provides an efficient method, which can be described and implemented by a finite list of instructive statements, for solving a particular and frequently encountered problem.

Definition 5.54 An algorithm A is a frequently recurring function f that maps a set of input X into a set of output Y by a finite set of statements or a finite-step process, i.e.:

$$\mathit{A} = f\!: X \rightarrow Y \tag{5.69}$$

The criteria that warrant a piece of program as an algorithm are due to its reusability, finiteness, and efficiency.

Example 5.22 A problem called the *In-Between Sum*, *IBSum*, for two given integers A and B, and $A < B$. For instance, when $A = 2$ and $B = 5$, $IBSum = 3 + 4 = 7$.

A direct algorithm for IBSum can be described by the following:

$$\sum_{i=A+1}^{B-1} i \tag{5.70}$$

A more efficient algorithm can be derived based on the difference of sums of *B-1* and *A* on the basis of known fact that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, i.e.:

$$\text{IBSum} = \sum_{i=1}^{B-1} - \sum_{i=1}^{A}$$

$$= \frac{(B-1) \bullet B}{2} - \frac{A \bullet (A+1)}{2} \qquad (5.71)$$

The second algorithm as shown in Eq. 5.71 is well designed because its complexity in average and worst cases is a constant *c*, i.e., *O(c)*, which is not dependent on the distance between A and B. However, the complexity of the first direct algorithm is *O(n)*, which is more complicated when *n* is considerably large.

The second *IBSum* algorithm is formally described in RTPA as shown in Fig. 5.27.

```
IBSAlgorithm.Architecture ≙ <Input: ST>
                            || <Output: ST>
                            || <Event: ST>
                            || <Status: ST>
=  InputST:: (<A: N | 0 < AN < 65535>,
              <B: N | 0 < BN < 65535, BN > AN>)
 || OutputST:: (<IBSum: N>)
 || EventST:: (<@IBSAlgorithmS>)
 || StatusST:: (<ⓢIBSResultBL>)

IBSAlgorithm.StaticBehaviors ≙ IBS_Algorithm

IBS_Algorithm ({I:: AN, BN}; {O:: ⓢIBSResultBL, IBSumN}) ≙
{
        → MaxN := 65535
        → (  � (0 < AN < maxN) ∧ (0 < BN < maxN) ∧ (AN < BN)
                 → IBSumN := $\frac{(B\mathbf{N} - 1) \bullet B\mathbf{N}}{2} - \frac{A\mathbf{N} \bullet (A\mathbf{N} + 1)}{2}$
                 → ⓢIBSResultBL := T
            | � ~
                 → ⓢIBSResultBL := F
                 → ! (@'AN and/or BN out of range, or AN ≥ BN')
           )
}
```

Figure 5.27 RTPA specification of the algorithm of In-Between Sum

5.5.2.2 Classes and Object-Orientation

Bertrand Russell (1872-1970) proposed that the world could be described by a set of objects, classes, and relations in 1900 [Russell, 1961], which is considered the earliest concept of modern object notions that has latterly been adopted in software engineering [Goldberg and Robson, 1983; Stroustrup, 1982/86/87; Booch, 1986; Rumbaugh et al., 1991].

Object-oriented programming is one of the significant developments of software engineering that emerged in the 1980s represented by SmallTalk [Goldberg and Robson, 1983] and C++ [Stroustrup, 1982/86/87]. In computer science and software engineering, a set of fundamental conceptual tools has been developed to cope with the complexity of problem specification and solution. Some of the important methodologies are *abstraction, information hiding, functional decomposition, modularization* and *reusability*. Object-orientation technologies have inherited the merits of these fundamental approaches and represented them in well organized mechanisms such as *encapsulation, inheritance, reusability,* and *polymorphism* [Gunter and Mitchell, 1994].

5.5.2.2.1 Mathematical Models of Classes

The abstract model of a generic class can be modeled using the following mathematical structures in RTPA.

Definition 5.55 A *class* is a dynamic construct in object-oriented programming to build hierarchical architectures of a system, which can be formally described below:

$$
\begin{array}{l}
\textbf{Class}\mathbf{ST} \triangleq \{\ \ \text{Architecture} : \mathbf{ST} \\
\qquad\qquad\quad \|\ \text{StaticBehaviors} : \mathbf{ST} \\
\qquad\qquad\quad \|\ \text{DynamicBehaviors} : \mathbf{ST} \\
\qquad\qquad\quad \}
\end{array} \tag{5.72}
$$

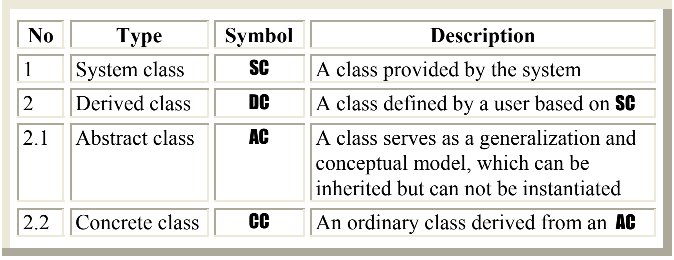
$$
\begin{array}{l}
\text{Class}\mathbf{ST}.\text{Architecture}\mathbf{ST} \triangleq \{\ \ <\text{Interfaces} : \mathbf{ST}> \\
\qquad\qquad\qquad\qquad\qquad\quad \|\ <\text{Implementations} : \mathbf{ST}> \\
\qquad\qquad\qquad\qquad\qquad\quad \}
\end{array} \tag{5.73}
$$

$$
\begin{array}{l}
\text{Class}\mathbf{ST}.\text{Architecture}\mathbf{ST}.\text{Interfaces}\mathbf{ST} \triangleq \text{ClassID}\mathbf{ST} :: \\
\qquad\qquad\qquad\qquad\qquad \{\ \ <\text{Attributes} : \mathbf{RT}> \\
\qquad\qquad\qquad\qquad\qquad\ \ \|\ <\text{Methods} : \mathbf{ST}> \\
\qquad\qquad\qquad\qquad\qquad \}
\end{array} \tag{5.74}
$$

The *interfaces* of a class are the means of access for users of the class, which model a set of attributes and methods. The *implementations* of the class are hidden behind the interfaces to realize detailed functions. For a well packaged class, the only access means to it is via its interfaces. The implementations of methods and related data structures are hidden inside the class, which enable the methods to be changed independently without affecting the interfaces of the class.

The types of classes in object-oriented methodologies can be classified into the categories of *system classes* (**SC**) and user *derived classes* (**DC**). The latter can be further divided into *abstract classes* (**AC**) and *concrete classes* (**CC**). These classes, as well as *methods* (**M**) modeled in a class, are treated as derived types of the system architecture type (**ST**) in RTPA as defined in Table 5.18.

Table 5.18
Taxonomy of Class Types



No	Type	Symbol	Description
1	System class	**SC**	A class provided by the system
2	Derived class	**DC**	A class defined by a user based on **SC**
2.1	Abstract class	**AC**	A class serves as a generalization and conceptual model, which can be inherited but can not be instantiated
2.2	Concrete class	**CC**	An ordinary class derived from an **AC**

Definition 5.56 *An object* is an instantiation of a class or multiple classes, which cannot be further inherited by other classes or objects.

Tracing back the history of programming methodologies, it can be seen that object-orientation is a natural extension and combination of two main stream programming methodologies: the *functional-oriented* programming and the *data-oriented* programming.

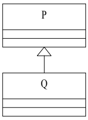
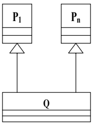
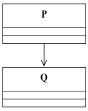
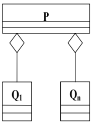
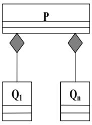
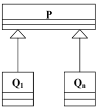
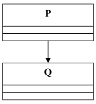
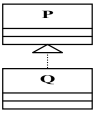
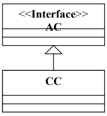
Definition 5.57 *Object Orientation* (OO) is a kind of system design and/or implementation methodologies that supports integrated functional- and data-oriented programming and system development.

5.5.2.2.2 Associations between Classes and Objects

The associations between classes can be classified into nine categories in OO methodologies as summaries in Table 5.19 [ORG, 2005]. Formal definitions of the mathematical semantics of these OO associations [Wang and Huang, 2005] are given in RTPA.

The associations of classes form a foundation to denote complicated relations between classes in software patterns, which will be discussed in Section 5.5.2.3. More formal treatment of the mathematical semantics of OO associations among classes may be referred to concept algebra as presented in Section 15.3.3 [Wang, 2006e]

Table 5.19
OO Associations and their Mathematical Semantics in RTPA

OO Associations	UML	RTPA
Inheritance (Inh)		$Inh \triangleq Q : P$
Multiple inheritance (MInh)		$MIng \triangleq Q : \mathop{R}\limits_{i=1}^{n} P_i$
Delegation (Del)		$Del \triangleq Q \rightarrowtail P$
Aggregation (Agg)		$Agg \triangleq \mathop{R}\limits_{i=1}^{n} P.M_i : Q_i,$ $P \oiint Q_i$
Composition (Com)		$Com \triangleq \mathop{R}\limits_{i=1}^{n} P.M_i : Q_i,$ $P \parallel Q_i$
Generalization (Gen)		$Gen \triangleq \mathop{R}\limits_{i=1}^{n} Q_i : P$
Instantiation (Ins)		$Ins \triangleq Q : P$
Dependency (Dep)		$Dep \triangleq \mathop{R}\limits_{i=1}^{n} Q.M_i : P$
Abstract/ Concrete class (AC_CC)		$CC\_AC \triangleq CC : AC$

5.5.2.2.3 Basic Attributes of Object-Orientation

OO technologies were originally designed for programming. Therefore OO was initially an implementation tool rather than a design tool. However, as OO programming became broadly accepted, it was found that OO technologies could be used not only in programming, but also in system design and analysis.

The fundamental attributes that can be commonly identified in OO technologies are encapsulation, inheritance, reusability, and polymorphism. A set of formal descriptions of these basic attributes is given below.

Definition 5.58 *Encapsulation* is a basic attribute of OO technologies by which functions and data structures of a class is integrated into a package, and the class can only be accessed through its methods with specific messages.

Definition 5.59 *Inheritance* is a basic attribute of OO technologies by which methods and related data structures modeled in a class can be inherited by derived classes or objects as existing system functions and/or structural types.

Definition 5.60 *Reusability* is a basic attribute of object-oriented technologies by which classes and their hierarchy modeled in an OO system can be reused by different applications as existing system resources.

Definition 5.61 *Polymorphism* is a basic attribute of OO technologies that provides evolvability and tailorability for inheritance by which the inherited methods and related data structures of a class can be partially redefined or overloaded.

Within the above set of basic attributes, encapsulation is a direct representation of the fundamental principles of abstraction, information hiding, and modularization in OO methodologies. Inheritance and reusability are powerful features for improving productivity and quality in software and system development. Polymorphism is a supplement of flexibility to other attributes.

OO technologies are useful conceptual modeling approach and generically applicable in software system design, analysis, and implementation. However, a number of drawbacks of OO systems as discussed below have also been identified in applications. For example, by using common OO languages, programmers must know details of a complicated structure of the foundational class hierarchy provided by a compiler in order to inherit or reuse a software component and/or a data type. This approach significantly increases the difficulty of mastering OO languages, and generates inherent complexity, subsequently, in OO software testing and maintenance.

Moreover, inherence of a made class hierarchy in an OO language is not tailorable. Programmers have to inherit anything that is contained in a given class and its ancestors, even just a small portion of functions of the class intended to be reused. This inefficient implementation of object technology results in a special phenomenon so called '*fatware*,' of which only an empty object encapsulation in common OO languages cost more than 10 kbytes in implementation. Also, inheritance from the root of a vendor's class system causes difficulty in testing and maintenance. For example, a project reported that more than half of the bugs in porting a Borland C++ based software into MS C++ environment were caused by the incompatibility between the two foundational class structures [Wang, 2001a].

Another drawback of OO technologies is that the data objects and their architectures are bounded with a set of predefined operations. When an object needs to use data objects defined in multiple classes, or to only use one data object from a large set of them defined in a class, the efficiency will be dramatically decreased by multiple inheritance. This is not acceptable in the design and implementation of real-time systems. More generally, there is only one global data object model for an entire real-time system, and it is commonly shared by all classes. If the system architectural model is defined in the fundamental class of the system, every other derived class has to inherit the whole data model, but only use a very small portion of it. These result in very high level of coupling between the data objects and their behavioral (operational) models and low efficiency in implementation, particularly for real-time systems.

In general, it is noteworthy that OO is only a high-level logical metaphor for software component and system modeling. There are no such concepts of object and class at machine or target language level. That is, either OO or non-OO technique results in the same implementation of a system at the machine level. In fact, OO techniques may result in low efficient code in both speed and memory usages than those of non-OO techniques. However, the gains from OO are a better modeling approach to complex systems, an improved readability of program, more available fundamental object libraries, and a wide-range support of tools.

5.5.2.3 Patterns

A *pattern* according to the Oxford Dictionary of English is a regular or logical form, a model, design, or a set of instructions for making something, or an excellent example. Christopher Alexander and his colleagues (1977), working in civil engineering, proposed that: "The *pattern* is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing."

Software patterns [Gamma, 1995; Taibi and Ngo, 2003; Vu and Wang, 2004] are a new component modeling technology built upon classes and object-oriented techniques. Software patterns are presented as a means of encapsulating the experience of architects and programmers in order to facilitate effective software reuse and design experience sharing. Patterns provide the following advantages in software architectural design: (a) experience encapsulation; (b) architectures of reusable components; and (c) enhanced documentation of software designs.

5.5.2.3.1 The Concept of Software Patterns

A pattern is a set of interacting classes. Patterns can be used as a powerful tool for capturing software design notions and best practices, which provide common solutions to core problems in software development.

Definition 5.62 A *pattern* is a complex software construct that incorporates a set of classes for a recurring architectural and behavioral design described by *abstract classes*, *concrete classes*, *instantiations*, and their *associations*.

Gamma and his colleagues (1995) proposed that software patterns can be classified into three categories known as the *creational, structural,* and *behavioral patterns*. The classification and description of software patterns can be summarized in Table 5.20, where the first three categories of patterns are adopted according to Gamma et al. (1995), while the fourth category of patterns is user defined according to Definition 5.62.

5.5.2.3.2 The Mathematical Model of Patterns

A pattern is a highly reusable and coherent set of complex classes that are encapsulated to provide certain functions [Wang and Huang, 2005]. Pattern specification is based on class specifications as described in Section 5.5.2.2. Using RTPA notations and methodology, a pattern is denoted by three parallel components known as the architecture, static, and dynamic behaviors at the top level. Then, the architecture of the pattern is refined by a CLM. The static and dynamic behaviors of the pattern are denoted by a set of collaborating processes.

Definition 5.63 The *generic mathematical model of software pattern* can be formally described by the four-level hierarchical model, as shown in Fig. 5.28, known as the *interfaces, implementations, instantiations*, and *associations* among the interfaces, implementations, and instantiations.

Table 5.20
Classification of Software Patterns

Category	Pattern	Description
1. Creational patterns	CP	CP deals with initializing and configuring classes and objects
1.1	Factory Method	Method in a derived class creates associates
1.2	Abstract Factory	Factory for building related objects
1.3	Builder	Factory for building complex objects incrementally
1.4	Prototype	Factory for cloning new instances from a prototype
1.5	Singleton	Factory for a singular (sole) instance
2. Structural patterns	SP	SP deals with decoupling interface and implementation of classes and objects
2.1	Adapter	A translator adapts a server interface for a client
2.2	Bridge	Abstraction for binding one of many implementations
2.3	Composite	Structure for building recursive aggregations
2.4	Decorator	Decorator extends an object transparently
2.5	Facade	Facade simplifies the interface for a subsystem
2.6	Flyweight	Many fine-grained objects shared efficiently
2.7	Proxy	One object approximates another
3. Behavioral patterns	BP	BP deals with dynamic interactions among societies of classes and objects
3.1	Chain of Responsibility	Request delegated to the responsible service provider
3.2	Command	Encapsulate a request as an object
3.3	Interpreter	Language interpreter for a small grammar
3.4	Iterator	Aggregate objects are accessed sequentially
3.5	Mediator	Coordinate interactions between its associates
3.6	Memento	Snapshot captures and restores object states privately
3.7	Observer	Update dependents automatically when a subject changes
3.8	State	Objects whose behavior depends on its state
3.9	Strategy	Abstraction for selecting one of many algorithms
3.10	Template Method	Algorithm with some steps supplied by a derived class
3.11	Visitor	Operations applied to elements of an heterogeneous object structure
4. User defined patterns	UDP	An abstraction of a class or an algorithm by separation of its interface and implementation details

The generic pattern model given in Fig. 5.28 may be treated as a super meta pattern, which reveals that any specific software pattern can be specified at four structural levels. According to Definition 5.62, the features of patterns lie in the hierarchical architectures as described by Pattern**ST**.Architecture**ST** (Eq. 5.76) in Fig. 5.28. It is noteworthy that a class is modeled as a two-level structure with the class *interfaces* and *implementations*. However, the architectural model of a pattern is a four-level hierarchy featured with the extended refinement levels of *instantiations* and *associations* for deriving applications of the pattern.

The *interface* of a pattern, Pattern**ST**.Architecture**ST**.Interfaces**ST** (Eq. 5.77), isolates users of the pattern from its internal implementations. Users may only access the pattern via its interfaces. This mechanism enables the implementation of the pattern to be independent of its users. Whenever the internal implementations need to be changed, it is transparent to the users of the pattern as long as the interfaces remain the same.

Since a pattern is a highly reusable design of a software object, the *implementation* of a pattern, Pattern**ST**.Architecture**ST**.Implementation**ST** (Eq. 5.78), will be kept in a generic concrete class, while the detailed and application specific functions related to users' specific requirements, which are captured via the interface, will be implemented at the lower-level concrete classes known as the *instantiations* Pattern**ST**.Architecture**ST**. Instantiations**ST** (Eq. 5.79) at run-time.

The fourth component in the generic pattern hierarchy is the internal *associations*, Pattern**ST**.Architecture**ST**.Association**ST** (Eq. 5.77), which is used to model the interrelationships among the rest of the three-level abstractions of classes and interfaces within the pattern.

The formal model of generic patterns can be used as a formula to derive specific descriptions of any software pattern. Case studies will be provided in the following section, which show that all typical and classic design patterns fit the generic pattern models of RTPA as developed in this section.

It is noteworthy that a pattern is a generic model of reusable functions. Specific behaviors in an execution instance are dependent on run-time information provided by uses of the pattern.

5.5.2.3.3 Pattern Modeling: Formal Models vs. UML Models

The conventional descriptive means for design patterns are natural languages or UML class diagram. Due to the inherited complexity, the architectural and semantic descriptive power of the above means is found inadequate [Vu and Wnag, 2004; Wang and Huang, 2005]. This section contrasts the descriptive power of RTPA and UML on denoting pattern architectures and behaviors.

$$
\begin{aligned}
\text{Pattern}\textbf{ST} \triangleq \{ \;& \text{Architecture} : \textbf{ST} \\
& \| \; \text{StaticBehaviors} : \textbf{ST} \\
& \| \; \text{DynamicBehaviors} : \textbf{ST} \\
\}&
\end{aligned}
\qquad (5.75)
$$

$$
\begin{aligned}
\text{Pattern}\textbf{ST}.\text{Architecture}\textbf{ST} \triangleq \{ \;& \text{<Interfaces>} \\
& \| \; \text{<Implementations>} \\
& \| \; \text{<Instantiations>} \\
& \| \; \text{<Associations>} \\
\}&
\end{aligned}
\qquad (5.76)
$$

$$
\begin{aligned}
&\text{Pattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Interfaces}\textbf{ST} \triangleq \text{PatterID}\textbf{ST} :: \\
&\qquad \{ \mathop{R}\limits_{i\textbf{N}=1}^{n\textbf{N}} \text{<Attributes(}i\,\textbf{N}\text{)} : \textbf{RT}\text{ >} \\
&\qquad \| \mathop{R}\limits_{j\textbf{N}=1}^{m\textbf{N}} \text{<AbstractClass(}j\textbf{N}\text{)} : \textbf{AC}\text{>} \\
&\qquad \}
\end{aligned}
\qquad (5.77)
$$

$$
\begin{aligned}
&\text{Pattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Implementations}\textbf{ST} \triangleq \\
&\qquad \{ \mathop{R}\limits_{k\textbf{N}=1}^{q\textbf{N}} \text{<ConcreteClass(}k\textbf{N}\text{)} : \textbf{CC}\text{>} \\
&\qquad \}
\end{aligned}
\qquad (5.78)
$$

$$
\begin{aligned}
&\text{Pattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Instantiations}\textbf{ST} \triangleq \\
&\qquad \{ \mathop{R}\limits_{l\textbf{N}=1}^{r\textbf{N}} \text{<Instantiation(}l\textbf{N}\text{)} : \textbf{CC}\text{>} \\
&\qquad \}
\end{aligned}
\qquad (5.79)
$$

$$
\begin{aligned}
&\text{Pattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Associations}\textbf{ST} \triangleq \\
&\qquad \{ \mathop{R}\limits_{j\textbf{N}=1}^{m\textbf{N}} (\; \text{<Interface(}j\textbf{N}\text{)}\textbf{AC} : \textbf{SC}\text{>} \\
&\qquad\qquad | \mathop{R}\limits_{m=1}^{p} Interface(j\textbf{N})\textbf{AC}.M_m\textbf{M} : Interface(j'\textbf{N})\textbf{AC}) \\
&\qquad \| \mathop{R}\limits_{k\textbf{N}=1}^{q\textbf{N}} \text{<Implementation(}k\textbf{N}\text{)}\textbf{CC} : \textbf{AC}(j\textbf{N})\text{>} \\
&\qquad \| \mathop{R}\limits_{l\textbf{N}=1}^{r\textbf{N}} \text{<Instantiation(}l\textbf{N}\text{)}\textbf{CC} : \textbf{CC}(k\textbf{N})\text{>} \\
&\qquad \}
\end{aligned}
\qquad (5.80)
$$

Figure 5.28 The generic mathematical model of software patterns

Example 5.23 The *builder pattern* is one of the important creational patterns proposed by Gamma and his colleagues [Gamma, 1995]. A UML class diagram of the builder pattern is shown in Fig. 5.29. This pattern is designed to separate the complex object construction process from its final diversity representations. The benefit of this treatment is that complicated construction processes may be reused to produce various object representations.

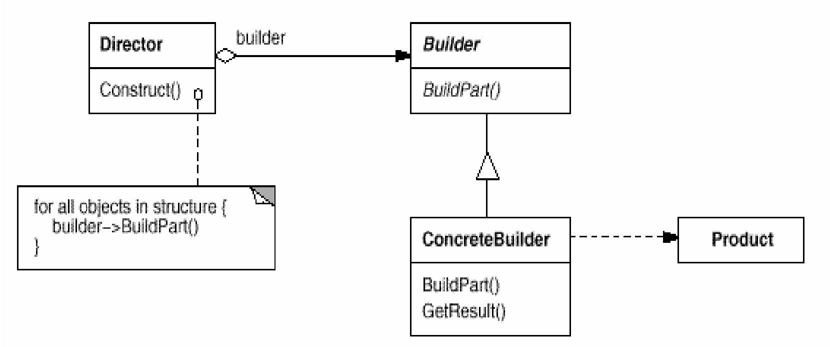


Figure 5.29 The structure of the Builder pattern

```
BuilderPatternST.StaticBehaviors ≙
{
   DirectorAC.Construct (<I:: PartIDS>; <O:: BuilderAC>)
  || BuilderAC.BuildPart (<I:: PartIDS>; <O:: ConcreteBuilderCC>)
  || ConcreteBuilderCC.BuildPart (<I:: PartIDS>; <O:: ConcreteBuilderCC.GetResult>)
  || ConcreteBuilderCC.GetResult (<I:: PartIDS>; <O:: ProductST>)
}
BuilderPatternST.DynamicBehaviors ≙ § →
{ @PartIDS
     ↳DirectorAC.Construct (<I:: PartIDS>; <O:: BuilderAC>)
        ↳BuilderAC.BuildPart (<I:: PartIDS>; <O:: ConcreteBuilderCC>)
           ↳ConcreteBuilderCC.BuildPart (<I:: PartIDS>;
                                         <O:: ConcreteBuilderCC.GetResult>)
              ↳ConcreteBuilderCC.GetResult (<I:: PartIDS>; <O:: ProductST>)

}
→ §
```

Figure 5.30 The behaviors of the Builder pattern

As shown in the formal model of the *Builder* pattern in Fig. 5.30, the *Director***AC** is an abstract class that serves as an interface of the builder pattern. The *Builder***AC** is another abstract class that represents the common and generic functions of the pattern, which may be implemented by a concrete class at lower levels. The *ConcreteBuilder***CC** is a concrete class that implements the conceptual model *Builder***AC**. Note that the implementation of the *ConcreteBuilder***CC** is still a generic model, which will be completely implemented by the *Product***CC** at run-time when a user of the pattern provides further details of instances via the interface of the pattern.

a) Architecture of the Builder Pattern

Using Eqs. 5.75 through 5.80 as the generic design formulae, the architecture of the BuilderPattern**ST** can be accurately and rigorously specified and stepwise refined as described below.

According to Eq. 5.75, the top-level architecture of the BuilderPattern**ST** can be specified as shown in Eq. 5.81.

$$\begin{aligned}\textbf{BuilderPatternST} \triangleq \{ \; & \text{Architecture} : \textbf{ST} \\ & \| \; \text{StaticBehaviors} : \textbf{ST} \\ & \| \; \text{DynamicBehaviors} : \textbf{ST} \\ \} \; & \end{aligned} \qquad (5.81)$$

Using Eq. 5.76, the architecture of the pattern, BuilderPattern**ST**. Architecture**ST**, can be specified as follows.

$$\begin{aligned}\text{BuilderPattern}\textbf{ST}.\text{Architecture}\textbf{ST} \triangleq \{ \; & \text{<Interface>} \\ & \| \; \text{<Implementation>} \\ & \| \; \text{<Instantiation>} \\ & \| \; \text{<Association>} \\ \} \; & \end{aligned} \qquad (5.82)$$

Applying Eq. 5.77, the interfaces of the pattern, BuilderPattern**ST**. Architecture**ST**.Interface**ST**, can be specified as follows.

$$\begin{aligned}& \text{BuilderPattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Interface}\textbf{ST} \triangleq \text{Builder}\textbf{ST} :: \\ & \quad \{ \; \mathop{R}_{i\textbf{N}=1}^{n\textbf{N}} \text{<Attributes(}i\textbf{N}\text{)} : \textbf{RT}\text{>} \\ & \quad \| \; \text{<Builder} : \textbf{AC}\text{>} \\ & \quad \| \; \text{<Director} : \textbf{AC}\text{>} \\ & \quad \} \end{aligned} \qquad (5.83)$$

Applying Eq. 5.78, the implementations of the pattern, BuilderPattern**ST**.Architecture**ST**.Implementation**ST**, can be specified as follows.

$$\begin{aligned}& \text{BuilderPattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Implementation}\textbf{ST} \triangleq \\ & \quad \{ \\ & \quad \text{<ConcreteBuilder} : \textbf{CC}\text{>} \\ & \quad \} \end{aligned} \qquad (5.84)$$

Applying Eq. 5.79, the instantiations of the pattern, BuilderPattern**ST**.Architecture**ST**.Instantiations**ST**, can be specified as follows.

$$\begin{array}{l}
\text{BuilderPattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Instantiation}\textbf{ST} \triangleq \\
\qquad \{ \\
\qquad\quad \langle \text{Product} : \textbf{CC} \rangle \\
\qquad \}
\end{array} \qquad (5.85)$$

The relations between the components described above, the associations of the pattern, BuilderPattern**ST**.Architecture**ST**.Associations**ST**, can be formally described on the basis of Eq. 5.80 as follows.

$$\begin{array}{l}
\text{BuilderPattern}\textbf{ST}.\text{Architecture}\textbf{ST}.\text{Association}\textbf{ST} \triangleq \\
\qquad \{ \quad \text{// Interface classes} \\
\qquad\quad (\quad \langle \text{Builder}\textbf{AC} : \textbf{SC} \rangle \\
\qquad\quad\ \ \| \langle \mathit{Director}\textbf{AC} : \textbf{SC} \mid \mathop{R}\limits_{m=1}^{p} \mathit{Director}\textbf{AC}.\mathit{M}_m\textbf{M} : \mathit{Builder}\textbf{AC} \rangle \\
\qquad\quad) \\
\qquad\ \| \text{ // Implementation classes} \\
\qquad\quad\ \langle \text{ConcreteBuilder}\textbf{CC} : \text{Builder}\textbf{AC} \rangle \\
\qquad\ \| \text{ // Instantiation classes} \\
\qquad\quad \langle \text{Product}\textbf{CC} : \text{ConcreteBuilder}\textbf{CC} \rangle \\
\qquad \}
\end{array} \qquad (5.86)$$

b) Behaviors of the Builder Pattern

Based on the formal model of the architecture of the Builder pattern, its static and dynamic behaviors can be rigorously described, as given in Fig. 5.30, using the RTPA behavioral modeling scheme described in Section 4.7.

It can be observed that the conventional pattern notations using UML class diagrams as shown in Fig. 5.29 are inadequate to denote what the behaviors and functions are, because it only provides a rough conceptual model. The RTPA notation system and methodology provide a rigorous means and generic formula for modeling any existing patterns and future user defined patterns in software engineering [Vu and Wang, 2004; Wang and Huang, 2005].

It is noteworthy that the behaviors of a pattern are highly general in nature, for which application specific details are yet to be implemented according to user's requirements during run-time until an instantiation of applications invokes the pattern.

Software patterns are a higher layer construct built upon classes and objects for modeling system architectures and behaviors. Software patterns may be adopted to enable abstraction, guide creative design, separate interface and implementation of system components, facilitate design reuse, and improve architecture design quality. The rigorous treatment of object-oriented patterns demonstrates a powerful cognitive means for comprehending existing patterns and creating new patterns in software engineering.

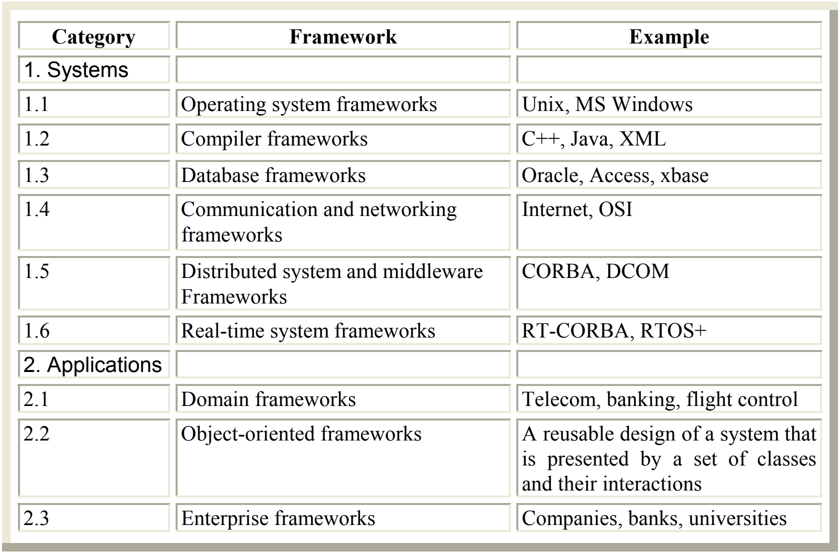
5.5.3 PROGRAMS MODELING AT SYSTEM LEVEL – FRAMEWORKS

System frameworks are the top-level system modeling techniques built upon algorithms, classes, and/or patterns in an object-oriented or component-based approach [Sparks et al., 1996; Fayad et al., 1999; Wang and King, 2000a].

Definition 5.64 A *software system framework* is an architectural model of an entire system that represents the overall structure, components, processes, and their interrelationships and interactions.

A system framework permits template-based development in software engineering. Framework technologies enable domain and design knowledge to be reused as well as that of code. The taxonomy of system frameworks can be summarized as shown in Table 5.21.

Table 5.21
Taxonomy of System Frameworks



Category	Framework	Example
1. Systems		
1.1	Operating system frameworks	Unix, MS Windows
1.2	Compiler frameworks	C++, Java, XML
1.3	Database frameworks	Oracle, Access, xbase
1.4	Communication and networking frameworks	Internet, OSI
1.5	Distributed system and middleware Frameworks	CORBA, DCOM
1.6	Real-time system frameworks	RT-CORBA, RTOS+
2. Applications		
2.1	Domain frameworks	Telecom, banking, flight control
2.2	Object-oriented frameworks	A reusable design of a system that is presented by a set of classes and their interactions
2.3	Enterprise frameworks	Companies, banks, universities

Example 5.24 A conceptual model of a Telephone Switching System (TSS) is given in Fig. 5.31 for illustrating the framework modeling methodology.

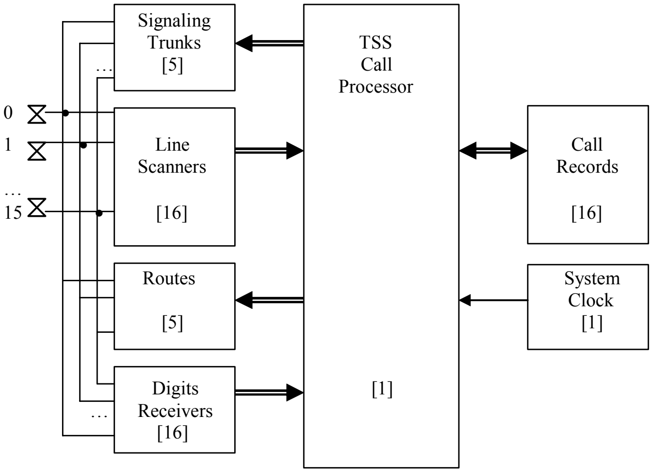


Figure 5.31 The telephone switching system (TSS)

The framework of the TSS system, encompassing its architecture, schemas of static behaviors, and schemas of dynamic behaviors, can be specified using RTPA as follows [Wang, 2002a].

a) *System architectural framework:* The architectural framework of the TSS system is given in Fig. 5.32.

```
TSS.ArchitectureST ≜ CallProcessingSubsys
                     || SubscriberSubsys
                     || RouteSubsys
                     || SignalingSubsys

=  ( CallProcessorST [1]   // specified by the system static/dynamic behaviors
     || SysClockST [1]
     || CallRecordsST [16]
   )
  || (  SubscribersST [16]            // status represented by
                                      // the line scanners and call records
     || LineScannersST [16]
   )
  || RoutesST [5]
  || (  DigitsReceiversST [16]
     || SignalingTrunksST [5]
   )
```

Figure 5.32 The architectural framework of the TSS system

b) *System static behavioral framework:* The static behavioral framework of the TSS system is given in Fig. 5.33.

```
TSS.StaticBehaviorsST ≜ SysInitial
                        || SysClock
                        || LineScanning
                        || DigitsReceiving
                        || ConnectDrive
                        || CallProcessing

TSS.StaticBehaviorsST.CallProcessing ≜
           CallOrigination
         || Dialling
         || CheckCalledStatus
         || Connecting
         || Talking
         || CallTermination
         || ExceptionalTermination
```

Figure 5.33 The static behavioral framework of the TSS system

c) *System dynamic behavioral framework:* The dynamic behavioral framework of the TSS system, in terms of TSS process deployment and dispatch, is given in Figs. 5.34 and 5.35, respectively.

```
TSS.ProcessDeploymentST ≜ § →
{   // Base level processes
   @SystemInitial
      ↳ ( SysInitial
```

$$\rightarrow \overset{\mathbf{T}}{\underset{SysShutdown\mathbf{BL}=\mathbf{F}}{R}} \text{CallProcessing}$$

```
        → ⊠
      )
 ||  // High-level interrupt processes
   @SysClock1msInt⊙
      ↳ (SysClock
          → DigitsReceiving
        )
 ||  // Low-level interrupt processes
   @SysClock100msInt⊙
      ↳  LineScanning
} → §
```

Figure 5.34 The process deployment framework of the TSS system

In Fig. 5.34, the iterative CallProcessing process is a complex process that can be further refined in the system process dispatching framework as shown in Fig. 5.35.

```
CallProcessing ≜ § →
{
 nN := 15
 → (� ⓢCallRecord.CallStatusBL = T
          → LineNumN := iN
          → ( @ CallRecord(iN).CallProcessN = 0                  // Idle
                  → ∅
              | @ CallRecord(iN).CallProcessN = 1                // Call origination
                  ↳ CallOrigination (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 2                 // Dialing
                  ↳ Dialling (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 3                // Check called status
                   ↳ CheckCalledStatus (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 4                 // Connecting
                  ↳ Connecting (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 5                // Talking
                   ↳ Talking (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 6                // Call termination
                  ↳ CallTermination (<I:: LineNumN>; <O:: CallProcessN>)
              | @ CallRecord(iN).CallProcessN = 7                // Exceptional termination
                  ↳ ExceptionalTermination (<I:: LineNumN>; <O:: CallProcessN>)
            )
   )
} → §
```

Figure 5.35 The process dispatching framework of the TSS system

In Example 5.24, all aspects of system framework descriptions follow the same system-level modeling scheme of RTPA as presented in Section 4.7.1. Further details of the TSS system framework specifications may be referred to [Wang, 2002a]. More empirical design and implementation of application frameworks have been reported in [Fayad and Schmidt, 1997; Fayad et al., 1999].

5.6 Resources and Processes Modeling and Manipulation

Theories and technologies of computational data objects, behaviors, and their interactions in the form of programs have been modeled in Sections 5.2 through 5.4, respectively. The three facets form the foundation of computing, programming, and software engineering, which enable the coordination of computing resources and processes to be modeled and manipulated at the top level – the operating system level.

This section describes typical architectures and generic functions of operating systems. In this section, the generic mathematical model of operating systems is established. The conceptual and typical commercial architectures of operating systems are surveyed. Then, the common functions of operating systems for computing resource and process manipulations, such as process and thread management, memory management, file system management, I/O system management, and network/communication management, are described. Real-time operating systems are presented with illustrations of the RTOS+ operating system to demonstrate how real-time resources and processes are coordinated and dispatched in computing.

5.6.1 ABSTRACT MODELS OF COMPUTING SYSTEMS

Any program as well as its behavior space and semantic environment must be realized and executed by a target computer. A generic computing system model, also known as a virtual machine, can be described as shown in Fig. 5.36, where all computing resources and processes are modeled and represented in the system. The hardware subsystem at the bottom of the architecture can be extended as shown in Fig. 5.5.

A mathematical model of the generic computing system on the basis of Fig. 5.36 can be described as follows.

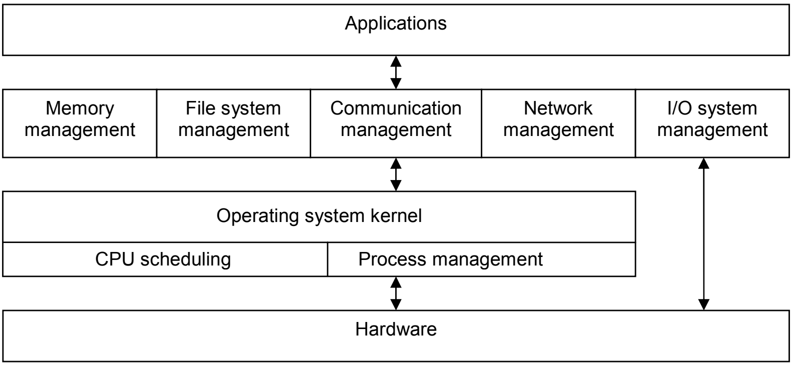


Figure 5.36 Architecture of the operating system of a GCS

Definition 5.65 The *Generic Computing System* (GCS), §, is an abstract logical model of the executing platform of a target machine denoted by a set of parallel or concurrent computing resources and processes, i.e.:

$$
\begin{array}{lll}
\S \triangleq \text{SysID}\mathbf{S} :: & & \\
\{ \; < \mathop{R}\limits_{i\mathbf{N}=0}^{n_{proc}\mathbf{N}-1} P_i\mathbf{ST}> & & \text{// Processes} \\
\; \| < \mathop{R}\limits_{addr\mathbf{P}=0}^{n_{MEM}\mathbf{H}-1} \text{MEM}[ptr\mathbf{P}]\mathbf{RT}> & & \text{// Memory} \\
\; \| < \mathop{R}\limits_{ptr\mathbf{P}=0}^{n_{PORT}\mathbf{H}-1} \text{PORT}[ptr\mathbf{P}]\mathbf{RT}> & & \text{// Ports} \\
\; \| <\S t\mathbf{TM}> & & \text{// The system clock} \\
\; \| < \mathop{R}\limits_{k\mathbf{N}=0}^{n_e\mathbf{N}-1} @e_k\mathbf{S} \hookrightarrow P_k> & & \text{// Event-driven dispatch} \\
\; \| < \mathop{R}\limits_{k\mathbf{N}=0}^{n_t\mathbf{N}-1} @t_k\mathbf{TM} \hookrightarrow P_k> & & \text{// Time-driven dispatch} \\
\; \| < \mathop{R}\limits_{k\mathbf{N}=0}^{n_{\text{int}}\mathbf{N}-1} @int_k\odot \hookrightarrow P_k > & & \text{// Interrupt-driven dispatch} \\
\; \| < \mathop{R}\limits_{i\mathbf{N}=0}^{n_V\mathbf{N}-1} V_i\mathbf{RT}> & & \text{// System variables} \\
\; \| < \mathop{R}\limits_{i\mathbf{N}=0}^{n_S\mathbf{N}-1} Ⓢ S_i\mathbf{BL}> & & \text{// System statuses} \\
\} & & (5.87)
\end{array}
$$

where || denotes the parallel relation between given components of the system, and its formal semantics is provided in Section 6.6.2.

As shown in Eq. 5.87 and Fig. 5.36, a computing system § is the executing platform or the operating system that controls all the computing resources of the abstract target machine. The system is logically abstracted as a set of processes and underlying resources, such as the memory, ports, the system clock, and system status. A process is dispatched and controlled by the system §, which is triggered by various external, system timing, or interrupt events.

Operating system technologies have evolved from rather simple notions of managing the hardware on behalf of a single user or sequentially scheduled users to multiuser time-sharing systems, and then to networked and distributed systems [Dijkstra, 1968b; Brinch-Hansen, 1971/73; Peterson and Silberschantz, 1985; Milenkovic, 1992; Tanenbaum, 1994/2001; Silberschatz et al., 2003]. Most modern operating systems are based on multi-programmed timesharing technologies.

Definition 5.66 A *Virtual Machine* (VM) is a subset of an operating system that represents various computing resources to users in a unified manner, and hides hardware details and physical implementation differences at the lower layers.

For example, the *Java Virtual Machine* (JVM) is a self-contained Java operating environment that simulates a specific computer platform and its resources.

Definition 5.67 An *Operating System* is a set of integrated system software that organizes, manages, and controls the resources and computing power of a computer, or a computer network, and provides users a logical interface for accessing the physical machine to run applications.

Almost all general-purpose computers need an operating system before any specific application may be installed and executed by users. The role of an operating system as a conceptual model of a computer is shown in Fig. 5.37.

The general-purpose operating systems can be classified into four types: the batch systems, time-sharing systems, real-time systems, and distributed systems. A *batch system* is an early type of operating system that runs similar jobs sorted by an operator as a batch through an operation console. A *time-sharing system* is a type of multitasking operating system that executes multiple jobs by automatically switching among them with predefined time slice. A *real-time operating system* is a type of special-purpose operating system that is designed for dealing with highly time-constrained events of processes and I/Os of control systems. A *distributed operating system* is a type of operating system that supports networking, communication, and file sharing among multiple computers via a network protocol [Sloane, 1993; Tanenbaum, 1994].

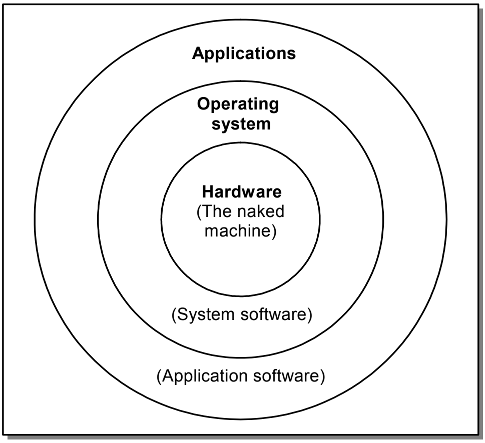


Figure 5.37 The role of an operating system in a general-purpose computer

5.6.2 ARCHITECTURES OF OPERATING SYSTEMS

The architectures of operating systems have evolved over the years from being a monolithic set of system services whose boundaries were difficult to establish to being a structured set of system services. Current operating systems are all based on the idea of building higher-level hardware abstraction from lower-level hardware-oriented functions. Thus, all kinds of hard disks, for example, are made to look and operate in the same manner by their low-level device drivers. Then, in turn, the operating system presents, with all other services in the system (such as the file system), a uniform, common view of the hard disk.

5.6.2.1 The Generic Architecture of Operating Systems

An operating system may be perceived as an agent between the computing resources of a computer or a computer network and the users as well as their applications as shown in GCS (Fig.5.36). The generic operating system may be divided into two parts: the *kernel* and the *resource* management subsystems [Peterson and Silberschantz, 1985; Silberschatz et al., 2003; and Tanenbaum, 2001]. The former is a set of central components for computing, including CPU scheduling and process management. The latter is a set of individual supporting software for various system resources and user interfaces.

The kernel is the most basic set of computing functions needed for an operating system. The kernel contains the interrupt handler, the task manager, and the interprocess communication manager. It may also contain the virtual memory manager and the network subsystem manager. With these services the system can operate all the hardware present in the system and also coordinate the activities between tasks. The services provided by an operating system can be organized in categories, where four typical *categories are task control, file manipulation, device control,* and *information maintenance*.

The following subsection presents the architectures of a number of typical operating systems such as Unix, Linux, and Windows XP. Users may compare their features with the generic architecture of GCS.

5.6.2.2 The Unix™ and Linux™ Operating Systems

The history of Unix can be traced back to 1969 based on Ken Thompson, Dennis Ritchie, and others' work [Thomas et al., 1986]. The name "Unix" was intended as a pun on Multics (UNiplexed Information and Computing System). The development of Unix was essentially confined to Bell Labs for DEC's PDP-11 (16 bits) and later VAXen (32 bits) [Earhart, 1986]. But much happened to Unix outside AT&T, especially at Berkeley. Major vendors of workstations, such as SUN's NFS, also contributed to this development.

The architecture of Unix is shown in Fig. 5.38, which can be divided into the kernel and the system programs. The Unix kernel consists of system resource management, interfaces, and device drivers, such as the CPU scheduling, file system, memory management, and I/O management.

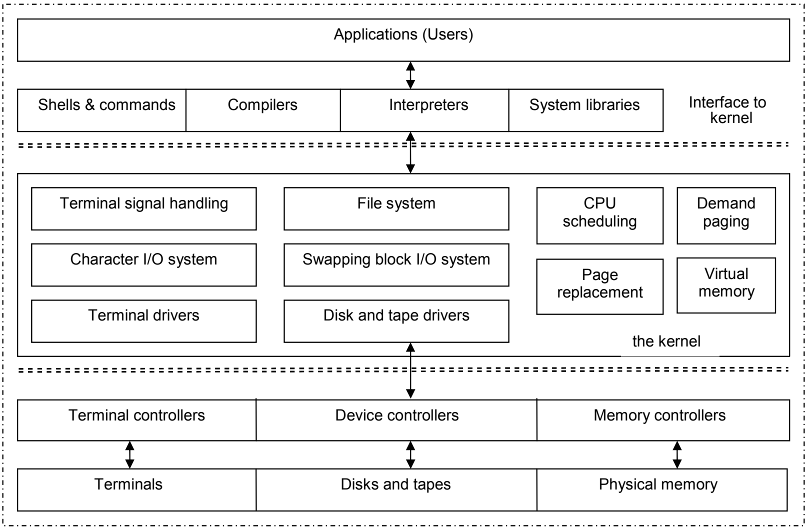


Figure 5.38 The architecture of Unix™

Linux is a complete Unix clone for Intel 386/486/Pentium machines [Siever et al., 2003]. Linux is an operating system, which acts as a communication service between the hardware and software of a computer system. The Linux kernel contains all of the features that one would expect in any operating system. Some of the features included are: multitasking, virtual memory, fast TCP/IP drivers, shared libraries, multi-user capability, and protected mode.

5.6.2.3 The Windows™ XP Operating System

Windows XP is a multitasking operating system built on enhanced technologies that integrate the strengths of Windows 2000 such as standard-based security, manageability, and reliability, with the best features of Windows 98 such as plug and play, and easy-to-use user interfaces.

The architecture of Windows XP is shown in Fig. 5.39. Windows XP adopts a layered structure that consists of the hardware abstraction layer, the kernel layer, the executive layer, the user mode layer, and applications.

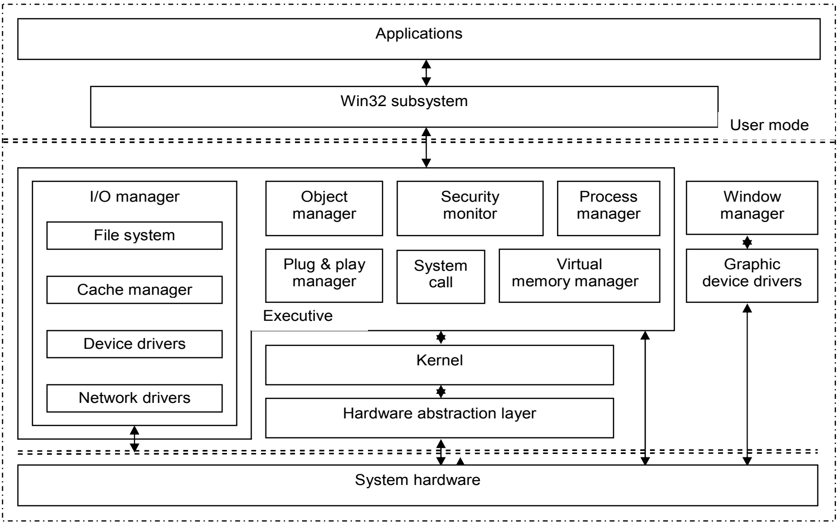


Figure 5.39 The architecture of Windows™ XP

Each kernel entity of Windows XP is treated as an object that is managed by an object manager in the executive. The kernel objects can be called by the user-mode applications via an object handle in a process. The use of kernel objects to provide basic services, and the support of client-server computing, enable Windows XP to support a wide variety of applications. Windows XP also provides virtual memory, integrated caching, preemptive scheduling, stronger security mode, and internationalization features.

5.6.3 COMPUTING RESOURCES MANIPULATION

The basic functions of operating systems can be classified as process and thread management, memory management, file system management, I/O system management, and network/communication management. This subsection describes the fundamental technologies of main operating system functions for computing resources manipulation in modern operating systems.

5.6.3.1 Process Management

A *process* is an execution of a program on a computer under the support of an operating system. A process can be a system process or a user process. The former executes system code, and the latter runs a user's application. Processes may be executed sequentially or concurrently depending on the type of operating systems.

The operating system carries out process management by the following activities:

- Detection of process requests
- Creation of processes by individual Process Control Blocks (PCBs)
- Allocation of system resources to processes, such as CPU time, memory, files, and I/O devices
- Scheduling of processes based on a predefined process state transition diagram
- Termination of processes

A typical process state transition diagram of a real-time operating system, RTOS+ [Wang and Ngolah, 2003], will be given in Fig. 5.41.

Threads are an important concept of process management in operating systems [Lewis and Berg, 1998]. A *thread* is a basic unit of CPU utilization, or a flow of control within a process, supported by a thread control block with a thread ID, a program counter, a set of registers, and a stack. Conventional operating systems are single thread systems. Multithreaded systems enable a process to control a number of execution threads. The benefits of multithreaded operating systems and multithreaded programming are responsiveness, resource sharing, implementation efficiency, and utilization of multiprocessor architectures of modern computers.

5.6.3.2 CPU Scheduling

CPU scheduling is a fundamental operating system function to maximize CPU utilization. The techniques of multiprogram and multithread are introduced to keep the CPU running different processes or threads on a time-sharing basis. CPU scheduling is the basis of multiprogrammed and multithreaded operating systems [Brinch-Hansen, 1971]. The CPU scheduler and dispatcher are two kernel functions of operating systems. The former selects which process in the ready queue should be run next based on a predefined scheduling algorithm or strategy. The latter switches control of the CPU to the process selected by the scheduler.

Some typical CPU scheduling algorithms are described as follows.

- *First-come-first-served* (FCFS) *scheduling:* This algorithm schedules the first process in the ready queue to the CPU based on the assumption that all processes in the queue have an equal priority. FCFS is the simplest scheduling algorithm for CPU scheduling. The disadvantage of the FCFS algorithm is that if there are long processes in front of the queue, short processes may have to wait for a very long time.

- *Shortest-job-first* (SJF) *scheduling:* This algorithm gives priority to the short processes, which results in the optimal average waiting time. But the predication of process length seems a difficult issue by using the SJF strategy.

- *Priority* (PR) *scheduling:* This algorithm assigns different priorities to individual processes. Based on this, CPU scheduling will be carried out by selecting the process with the highest priority. The drawback of the priority algorithm is *starvation*, a term that denotes the indefinite blocking of low priority processes under high CPU load. To deal with starvation, the *ageing* technique may be adopted that increases the priority levels of low priority processes periodically, so that the executing priorities of those processes will be increased automatically while waiting in the ready queue.

- *Round-robin* (RR) *scheduling:* This algorithm allocates the CPU to the first process in the FIFO ready queue for only a predefined time slice, and then it is put back at the tail of the ready queue if it has not yet been completed.

- *Multiprocessor scheduling:* This algorithm schedules each processor individually in a multiprocessor operating system on the basis of a common queue of processes. In a multiprocessor operating system, processes that need to use a specific device have to be switched to the right processor that is physically connected to the device.

5.6.3.3 Memory Management

Memory management is one of the key functions of operating systems because memory is both the working space and storage of data or files. Common memory management technologies of operating systems are contiguous allocation, paging, segmentation, and combinations of these methods [Silberschatz et al., 2003].

- *Contiguous memory allocation:* This method is used primarily in a batch system where memory is divided into a number of fixed-sized partitions. The contiguous allocation of memory may be carried out by searching a set of holes (free partitions) that best fit the memory requirement of a process. A number of algorithms and strategies were developed for contiguous memory allocation such as the first-fit, best-fit, and worst-fit algorithms [Tanenbaum and Tanenbaum, 2001].
- *Paging:* Paging is a dynamic memory allocation method that divides the logical memory into equal blocks known as pages corresponding to physical memory frames. In a paging system, each logical address contains a page number and a page offset. The physical address is generated via a page table where the base address of an available page is provided. Paging technology is used widely in modern operating systems to avoid the fragmentation problem as found in the early contiguous memory allocation techniques.
- *Segmentation:* This is a memory-management technique that uses a set of segments (logical address spaces) to represent user's logical view of memory independent of the physical allocation of system memory. Segments can be accessed by providing their names (numbers) and offsets.
- *Virtual memory:* When the memory requirement of a process is larger than physical memory, an advanced technique needs to be adopted known as the virtual memory, which enables the execution of processes that may not be completely in memory. The main approach to implement virtual memory is to separate the logical view of system memory from its physical allocation and limitations. Various technologies have been developed to support virtual memory such as the demand paging and demand segmentation algorithms [Silberschatz et al., 2003].

In memory-sharing systems, the sender and receiver use a common area of memory to place the data that is to be exchanged. To guarantee appropriate concurrent manipulation of these shared areas, the operating system has to provide synchronization services for mutual exclusion. A common synchronization primitive is the *semaphore*, which provides mutual exclusion for two tasks using a common area of memory. In a shared memory system the virtual memory subsystem must also collaborate to provide the shared areas of work.

5.6.3.4 File System Management

File system is the most used function of operating systems for non-programming users. A file is a logical storage unit of data or code separated from its physical implementation and location. Types of files can be text, source code, executable code, object code, word processor formatted, or system library code. The attributes of files can be identified by name, type, location (path of directory), size, date/time, user ID, and access control information. Logical file structures can be classified as sequential and random files. The former are files that organize information as a list of ordered records; while the latter are files with fixed-length logical records accessible by its block number.

Typical file operations are reading, writing, and appending. Common file management operations are creating, deleting, opening, closing, copying, and renaming.

The file system of an operating system consists of a set of files and a directory structure that organizes all files and provides detailed information about them. The major function of a file management system is to map logical files onto physical storage devices such as disks or tapes. Most file systems organize files by a tree-structured directory. A file in the file system can be identified by its name and detailed attributes provided by the file directory. The most frequently used method for directory management is the *hash table*. Although it is fast and efficient, backup is always required to recover a hash table from unpredicted damage.

A physical file system can be implemented by contiguous, linked, and indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct-access is inefficient with linked allocation. Indexed allocation may require substantial overheads for its index block.

5.6.3.5 I/O System Management

I/O devices of a computer system encompass a variety of generic and special-purpose hardware and interfaces. Typical I/O devices that an operating system deals with are shown in Table 5.22.

Table 5.22
Types of I/O Devices

Types of I/O devices	Examples
System devices	System clock, timer, interrupt controller
Storage devices	Disks, CD drivers, tapes
Human interface devices	Keyboard, monitor, mouse
Communication devices	Serial/parallel buses, network cards, DMA controllers, MODEMs
Special devices	Application-specific control devices

I/O devices are connected to the computer through buses with specific ports or I/O addresses. Usually, between an I/O device and the bus, there is a device controller and an associated device driver program. The I/O management system of an operating system is designed to enable users to use and access system I/O devices seamlessly, harmoniously, and efficiently.

I/O management techniques of operating systems can be described as follows:

- *Polling:* Polling is a simple I/O control technique by which the operating system periodically checks the status of the device until it is ready before any I/O operation is carried out.

- *Interrupt:* Interrupt is an advanced I/O control technique that lets the I/O device or control equipment notify the CPU or system interrupt controller whenever an I/O request has occurred or a waiting event is ready. When an interrupt is detected, the operating system saves the current execution environment, dispatches a corresponding interrupt handler to process the required interrupt, and then returns to the interrupted program. Interrupts can be divided into different priorities on the basis of processor structures in order to handle complicated and concurrent interrupt requests.

- *DMA:* Direct memory access (DMA) is used to transfer a batch of large amounts of data between the CPU and I/O devices, such as disks or communication ports. A DMA controller is handled by the operating system to carry out a DMA data transfer between an I/O device and the CPU.

- *Network sockets:* Most operating systems use a socket interface to control network communications. When requested in networking, the operating system creates a local socket and asks the target machine to be connected to establish a remote socket. Then, the pair of computers may communicate by a given communication protocol.

5.6.3.6 Communication Management

A fundamental characteristic that may vary from system to system is the manner of communication between tasks. The two manners in which this is done are via messages sent between tasks or via the sharing of memory where the communicating tasks can both access the data. Operating systems can support either. In fact, both manners can coexist in a system.

In message-passing systems, the sender task builds a message in an area that it owns and then contacts the operating system to send the message to the recipient. There must be a location mechanism in the system so that the sender can identify the receiver. The operating system is then put in charge of delivering the message to the recipient. To minimize the overhead of the message delivery process, some systems try to avoid copying the message from the sender to the kernel and then to the receiver, but to provide means by which the receiver can read the message directly from where the sender wrote it. This mechanism requires the operating system intervenes if the sender wants to modify the contents of a message before the recipient has gone through its content.

In memory-sharing systems, the sender and receiver use a common area of memory to place the data that is to be exchanged. To guarantee appropriate concurrent manipulation of these shared areas, the operating system has to provide synchronization services for mutual exclusion. A common synchronization primitive is the *semaphore,* which provides mutual exclusion for two tasks using a common area of memory. In a shared memory system, the virtual memory subsystem must also collaborate to provide the shared areas of work.

The ISO *Open Systems Interconnection* (OSI) reference model was developed in 1983 [Day and Zimmermann, 1983] for standardizing data communication protocols between different computer systems and networks. The OSI reference model is an important protocol framework for regulating multi-vendor multi-OS computers interconnection in Local Area Network (LAN) and Wide Area Network (WAN) environments [Stallings, 2000]. From bottom-up, the seven layers are: physical, data link, network, transport, session, presentation, and application as shown in Fig. 5.40.

Fig. 5.40 contrasts the functional equivalence between the OSI model and TCP/IP (the Transmission Control Protocol/Internet Protocol) [Day and Zimmermann, 1983]. The TCP/IP design philosophy [Comer, 1996/2000] is to provide universal connectivity with connection-independent protocols at the network layer. Thus TCP/IP does not address the data link and physical layers which determine the communication channels. There are no separate application, presentation, and session layers in TCP/IP; instead, a combined application layer is provided in TCP/IP, which has the functions of those layers.

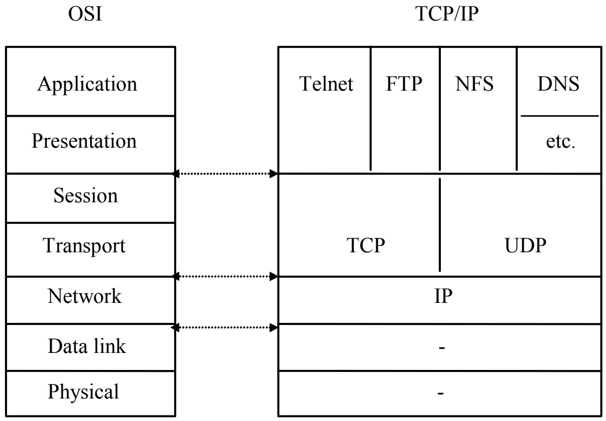


Figure 5.40 The OSI reference model and TCP/IP

The IP protocol is approximately equivalent to the OSI network layer. In a WAN, IP is presented on every node in the network. The role of IP is to segment messages into packets (up to 64 kbyte) and then route and pass the packets from one node to another until they reach their destinations. IP uses packet switching as its fundamental transmission algorithm. A message is transmitted from gateway to gateway by dynamic routed packets. IP routes packets to their destination network, but final delivery is left to TCP. The TCP protocol fulfills the role of the OSI transport layer, plus some of the functionality of the session layer. TCP is designed to provide end-to-end connectivity. TCP is not required for packet routing, so it is not included on gateways. TCP provides an acknowledgement mechanism to enable messages to be sent from destination(s) back to the sender to verify receipt of each packet that makes up a message.

5.6.3.7 Network Management

A network operating system implements protocols that are required for network communication and provides a variety of additional services to users and application programs. Network operating systems may provide support for several different protocols known as stacks, e.g., a TCP/IP stack and an IPX/SPX stack. A modern network operating system provides a socket facility to help users to plug-in utilities that provide additional services. Common services that a modern network operating system can provide are as follows.

- *File services:* File services transfer programs and data files from a computer on the network to another.
- *Message services:* Message services allow users and applications to pass messages from a computer to another on the network. The most familiar application of message services is Email and intercomputer talk facilities.
- *Security and network management services:* These services provide security across the network and allow users to manage and control the network.
- *Printer services:* Printer services enable sharing of expensive printer resources in a network. Print requests from applications are redirected by the operating system to a network workstation, which manages the requested printer.
- *Remote Procedure Calls* (*RPCs*)*:* RPCs provide application program interface services to allow a program to access local or remote network operating system functions.
- *Remote processing services:* These services allow users or applications to remotely login to another system on the network and use its facilities for program execution. The most familiar service of this type is Telnet, which is included in the TCP/IP protocol suite [Comer, 1996/2000] and many other modern network operating systems.

5.6.4 REAL-TIME/EMBEDDED RESOURCES AND PROCESSES MANIPULATION

A *Real-Time Operating System* (RTOS) is essential to implement embedded and/or real-time control systems. An RTOS is an operating system that guarantees timely processing of external and internal events of real-time systems. There were varying models of real-time operating systems developed in the last decades [Labrosse, 1999; Laplante, 1977; Liu, 2000].

Problems often faced by RTOS's are CPU and tasks scheduling, timing/event management, and resource management. RTOS requires multitasking, process threads, and explicit interrupt levels to deal with real-time events and interrupts. An extended RTOS, RTOS+ [Wang and Ngolah, 2003; Ngolah et al., 2004] is presented in this section to describe real-time resources modeling and manipulation.

5.6.4.1 The Architecture of RTOS+

RTOS+ is a portable and multitask/multithread operating system capable of handling event-, time-, and interrupt-driven processes in a real-time environment. The architecture of RTOS+ is shown in Fig. 5.41, where interactions between system resources, components, and internal control models are illustrated. There are four subsystems in RTOS+: a) the processor and task scheduler, b) the resource controller, c) the event handler, and d) the system resources. The task scheduler is the innermost operating system kernel that directly controls the CPU and all other system resources by system control blocks. The resources of RTOS+, supplemented to the CPU, are mainly the memory, system clock, I/O ports, interrupt facilities, files, and internal control models such as queues and task/resource control blocks as shown in Fig. 5.41.

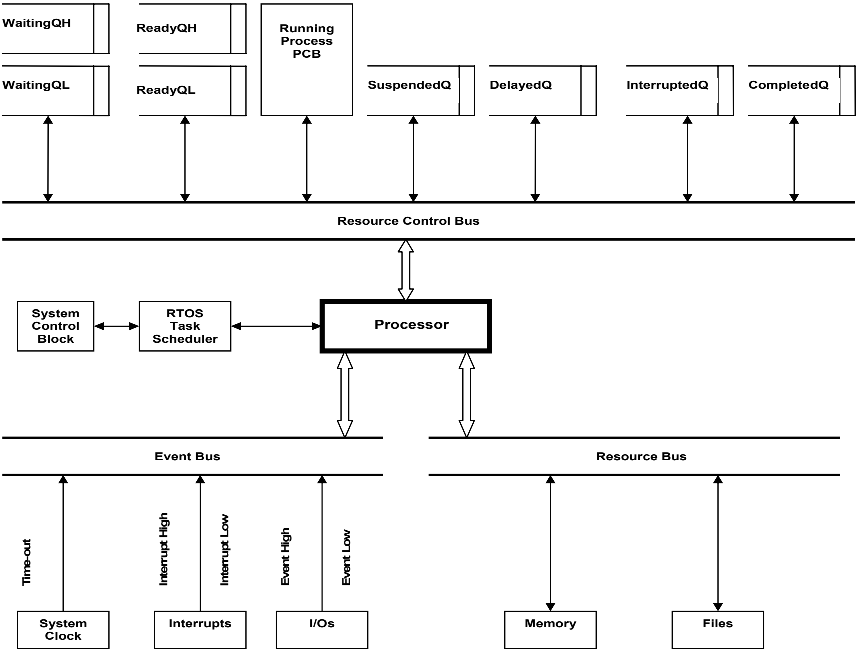


Figure 5.41 The architecture of RTOS+

The task scheduling of RTOS+ is priority-based. A fixed-priority scheduling algorithm is adopted where the priority of a task is assigned based on its importance when it is created. Tasks are categorized into four priority levels with descending priority: the high and low priority interrupts, as well as the high and base priority processes. A process, when created, will be put into a proper queue corresponding to its predefined priority level.

5.6.4.2 The Task Scheduler of RTOS+

The *Task Scheduler* is the kernel of RTOS+. Its behaviors can be modeled by a state transition diagram as shown in Fig. 5.42. The task scheduler of RTOS+ is designed for handling event-, time-, and interrupt-driven processes, in which the CPU is allocated by a fixed time-slice for executing a given process.

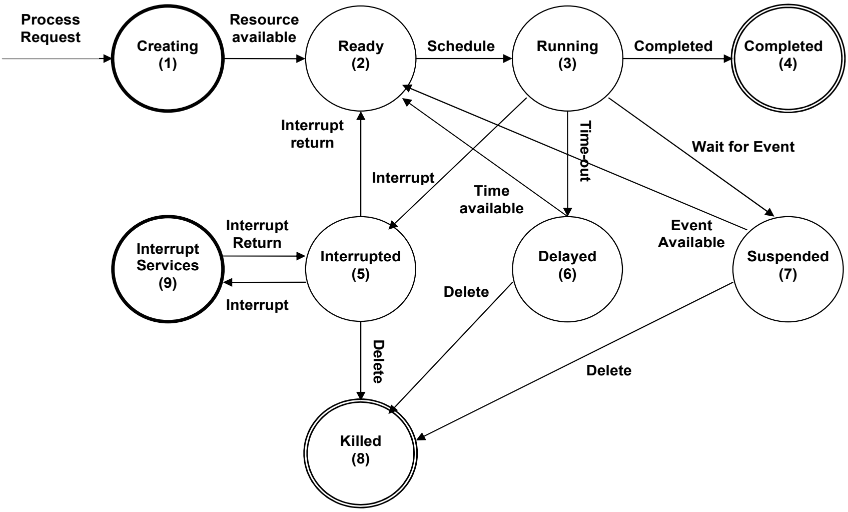


Figure 5.42 Process state transition diagram of RTOS+

Process requests are handled by the task scheduler for creating a process. When a new process is generated, it is first put into the *waiting state w*ith a PCB and a unique task ID. The system uses a Resource Control Block (RCB) to manage system resources. Each task in the waiting state must be checked to see if there are enough resources for its execution. If resources are available, it is transferred into the *ready state*; otherwise it has to be re-queued at the tail of the waiting queue until resources are available.

The task scheduler continuously checks the ready queue for any ready task. If there are ready tasks, it executes the first task in the queue until it is completed (State 4) or is suspended. There are three conditions that may cause a running task to be suspended during execution: a) interrupted by a task or event with higher priority, b) time-out for a scheduled time-slice of CPU, and c) waiting for a specific event. The scheduler may remove the CPU from a running task if a higher priority interrupt request occurs. Such interruption will cause the running task to go into the *interrupted queue* and later return to the appropriate ready queue when the interrupt service is over. A task that has exhausted its assigned time-slice must go to the *delayed queue*. When a new CPU time-slice is available, it is re-scheduled into the appropriate ready queue. A task that can no longer be executed due to waiting for an event goes into the *suspended queue*, and returns to the appropriate ready queue once the event has occurred. In any of these suspended cases, the task is put into a corresponding queue in States 5, 6, or 7, respectively. The task will be re-scheduled into ready state when the cause of the suspension is no longer true.

A task suspended may be cancelled (State 8) by the scheduler from the queues of States 5, 6, or 7 in case there is a lack of resources or under the request of users.

5.6.4.3 Process Dispatching of RTOS+

In the previous subsection, the conceptual model of RTOS+ has been established. To further refine the design of RTOS+, RTPA is adopted as a formal specification means. The dynamic behaviors of RTOS+ can be described by the interactions of parallel processes between *TaskScheduling*, *TimeManagement, ResourceManagement*, and *SystemControlUpdate* as shown in Fig. 5.43.

```
RTOSST.ProcessDeployment ≜
{ // Basic level processes
    @SystemInitial
       ↳ ( SysInitial
                    T
             ↳      R               TaskManagement
             @SysShutDown=F
             → ⊠
          )
 || // High-interrupt level processes
     ⊙ @SysClock1msInt
        ↗ ( SysClock
              ↳ TaskScheduling
              ↳ SystemControlBlockUpdate
              ↳ InterruptManagement
              ↳ TimeManagement
              ↳ HighPriorityEventScan
           )
        ↘ ⊙
 || // Low-interrupt level processes
      ⊙ @SysClock100msInt
         ↗ ( BaseEventScane
               ↳ ResourceManagement
           )
         ↘ ⊙
}
```

Figure 5.43 Real-time process deployment in RTOS+

RTOS+ runs the *TaskManagment* routine continuously by updating and dispatching various processes in different queues. If an interrupt occurs during run-time, the interrupt handling process (*SysClock100msInst*) saves the current executing environment, switches control to the interrupting process, and then automatically returns to the interrupted process following completion of a higher priority operation. *SysClock100msInt* handles low level interrupt events, such as system ResourceManagement at 100ms intervals.

Corresponding to the state transition diagram as shown in Fig. 5.42, the task scheduler of RTOS+ is specified by RTPA in Fig. 5.44. The process dispatching mechanism of RTPA is used to formally describe the RTOS+ dynamic behaviors by a set of event-driven relationships between system events and functional processes of the operating system kernel.

```
RTOSST.TaskScheduler ≜
{   � ⓢNewProcRequestBL = T
        → CreatePCB (<I:: ProcIDN>; <O:: ()>
        → ProcStateN = 1
  | � ⓢPCB.ProcStateN ≥ 1
      � ProcStateN
        | 1 → Waiting (<I::(ProcIDN); O::()>)
        | 2 → Ready (<I::(ProcIDN); O::()>)
        | 3 → Running (<I::(ProcIDN); O::()>)
        | 4 → Completed (<I::(ProcIDN); O::()>)
        | 5 → Interrupted (<I::(ProcIDN); O::()>)
        | 6 → Delayed (<I::(ProcIDN); O::()>)
        | 7 → Suspended (<I::(ProcIDN); O::()>)
        | 8 → Killed (<I::(ProcIDN); O::()>)
        | 9 → InterruptService (<I::(ProcIDN); O::(IntReturnBL)>)
         |~ → ∅
}
```

Figure 5.44 Dynamic behaviors of the RTOS+ task scheduler

Other core operations of RTOS+, such as task scheduling, time, event, and resource management, can be rigorously modeled and described for better real-time performance and improved resource utilization. On the basis of the formal specification of RTOS+ by RTPA, architectural and behavioral consistency and correctness of RTOS+ can be improved [Wang and Ngolah, 2003].

This section has described resources and processes organization and manipulation by operating systems. The conceptual and typical commercial architectures of operating systems have been surveyed. A mathematical model of generic operating systems and computing resources is established. Common functions of operating systems, such as process and thread management, memory management, file system management, I/O system management, and network/communication management, are presented. The real-time operating system, RTOS+, is introduced with formal models and event/time/interrupt-driven process dispatching techniques.

5.7 Summary

Computer science is an applied scientific and engineering discipline with the systematic study and development of computers and software as its principle subject matters. **Computing theories** encompass computational *methods*, computing *objects*, and computing *resources*, which form one of the most important and direct foundations of software engineering. The **essences of computer science** are the rigorous treatment of modeling theories and techniques for *data objects, system architectures, operational behaviors,* and their interactions incorporated in a *program*.

This chapter has explored the computing foundations of software engineering on rigorous treatment of data objects, architectures, behaviors, program modeling theories, and techniques. It has also examined what computer science may and may not provide for software engineering. Basic computation models, such as automata, Turing machines, von Neumann machines, cognitive computers, and autonomic computing machines, have been explored. Data objects modeling with type theory and system architectural modeling with CLMs have been centered. Behavioral modeling and manipulation has been focused on meta processes and BCS's because of their fundamental and highly recurring roles in computing. Program modeling has been treated as a coordination of computational behaviors and data objects/architectures. Resources manipulation and process coordination in software engineering have been presented with generic and real-time operating system models. As a result, the **computing foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Computing Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge architecture as summarized below.

Chapter 5. Computing Foundations of SE

■ Basic Computation Models

- Basic operations in computing
- Automata
  - Automata and Finite State Machines (FSMs)
  - Approaches to describe FSMs
  - Description of software behaviors by FSMs
  - FSM composition and fefinement
  - Deterministic and nondeterministic automata
  - Usage of automata
- Turing machines
  - The abstract model of computing
  - Formal description of Turing machines
  - The nature of computing
- von Neumann machines
  - The stored-program concept
  - The von Neumann architecture of computers
- Cognitive machines
  - The Wang architecture of computers
  - Cognitive computers

■ Data Object Modeling and Manipulation

- Types and data structures
  - Type systems of programming languages
  - Primitive types
  - Derived and advanced types
  - System architectural types
- Basic data modeling techniques
  - Identifiers
  - Variables and constants
  - Expressions
- Formal type theory
  - Type rules
  - Formal type systems
  - Complex type rules for the RTPA derived types
- Abstract data types (ADTs)
  - The generic model of ADTs
  - Modeling complex data structures and component architectures by ADTs
  - Typical ADTs modeled in RTPA

- ■ Behavioral Modeling and Manipulation
  - Internal behaviors modeling
    - Basic control structures (BCS's)
    - Control flow graphs
  - Iterative and recursive behaviors modeling
    - Formal description of iterations
    - Formal description of recursions
    - Comparative analysis of iterations and recursions
  - External and interactive behaviors modeling
    - Memory manipulations
    - Events handling
- ■ Program Modeling: Coordination of Computational Behaviors and Data Objects
  - The unified mathematical model of programs
    - The abstract model of statements
    - The abstract model of processes
    - The abstract model of programs
  - Program modeling at component level
    - Algorithms
    - Classes and object-orientation
    - Patterns
  - Program modeling at system level - frameworks
- ■ Resources and Processes Modeling and Manipulation
  - Abstract models of computing systems
  - Architectures of operating systems
    - The generic architecture of operating systems
    - The Unix and Linux operating systems
    - The Windows XP operating systems
  - Computing resources manipulation
    - Process management
    - CPU scheduling
    - Memory management
    - File system management
    - I/O system management
    - Communication management
    - Network management
  - Real-time/embedded resources and processes manipulation
    - The architecture of RTOS+
    - The task scheduler of RTOS+
    - Process dispatching of RTOS+

SIGNIFICANT FINDINGS OF THIS CHAPTER

- **Software engineering** was perceived as a branch of computer science. However, computer science only provides basic computing theories and programming methodologies. Software engineering has historically focused on programming methodologies, programming languages, software development models, and tools. Areas now thought critical to software engineering – nature of software, cognitive foundations, denotational mathematical means, architectural and behavioral laws, system theories, organizational and management infrastructures – have been largely overlooked.

- The **objects in computation** can be abstracted by binary digits (bits) and a few primitive types. Any complex real-world data object in computing can be reduced to these primitive types. Based on this profound ***axiom*** **of data objects in computing**, computational methods in general are assumed to be arithmetical and logical operations, and any other complex operations must be reduced to these kinds of basic forms. In addition, computing resources are dramatically simplified as a sequential memory space with binary digits or characters. That is why the hardware technology was so mature because all issues can be reduced to basic operations and basic objects by deduction.

- However, in software engineering, the development of software is a one-off activity. To the maximum extent, a program can only be reduced to known languages statements and primitive types. The method and process are highly reusable, but the objects and resources are far more complicated than those of hardware devices.

- Most software systems go wrong not because they are incorrect on **normally required functions**, but because there are wrong or not prepared for implied or **nonspecified exceptions** (Theorem 5.3). Therefore, system design and specification should focus on the entire $S_\Omega = \delta + \overline{\delta}$.

    - This is a major indicator that **distinguishes professionals and amateurs** in software engineering, where the latter focus only on required behaviors (δ); while the former model the whole behaviors of a given system ($\overline{\delta}$ + δ).

- In computing, the most generic functions and routine tasks are implemented with a hardware processor. Therefore, what left for software are *one-off applications*. As a consequence, the reuse rate of software cannot be as high as any software reuse technique promises. Just like that in the publishing and journalist industries, nobody talks reuse in composition.

• The **mathematical model of programs** is a finite set of cumulated relations between processes, which in turn is a finite set of cumulated relations of statements (Eqs. 5.65, 5.67, and 5.68).

• An **Autonomic Computing Machine** (ACM) is a nonimperative computer that autonomously carries out robotic and interactive applications based on goal- and event-driven mechanisms on the basis of nonlinear and content sensitive memory architectures.

• On the basis of the above fundamental computing models, the **entire computing theory** can be divided into:

- Data object modeling
- Operational behavioral modeling
- Program modeling
- Resource and process modeling

• The **data object modeling** process is much more important and difficult than that of behaviors modeling, because the former is an **open and creative process** and it involves both real-world entities and their abstract representation with computing resources and expressing constraints.

• **Fundamental computing behaviors** can be classified into eight categories, such as *data manipulations, arithmetical operations, logical operations, bitwise operations, program controls, memory manipulations, I/O manipulations,* and *interrupt and time manipulations*.

• **Basic Control Structures** (BCS's) are the fundamental compositional means of programming. The mathematical laws of BCS's and other process relations have been described in Section 5.4.1.

• In programming, **iterative** and **recursive behaviors** are **equivalent**. A recursive solution for an iterative problem is usually more expressive, but less efficient in implementation than its iterative counterpart.

• The **mathematical model of programs** is a finite set of cumulated relations between processes, which in turn is a finite set of cumulated relations of statements.

• A **pattern** is a computational construct that extends the structure of a single class from *interface* and *implementation* to *instantiations and associations*.

- The **Generic Computing System** (GCS), §, is an abstract logical model of the executing platform of a target machine denoted by a set of parallel or concurrent computing resources and processes (See Eq. 5.87).

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Basic Computation Models

- A **statement** is the smallest functional unit of a program that specifies an explicit action and results in the change of one or more variables. A **generic model of a statement** in programming languages can be described as a function, or a process, that maps a set of input into a set of output.

- The **fundamental operations** in computing can be classified into three categories: *computational operations, object manipulations*, and *resource manipulation*. These **fundamental computing needs** can be reduced to only *binary data, basic Boolean operations* ($\wedge$, $\vee$, and $\neg$), and *a linear memory space*. Any complex application can be implemented on the basis of these three essences of computation by certain composition rules.

- A **finite State Machine** (FSM, or **automaton**) is defined by a 5-tuple encompassing the *alphabet*, *states*, *initial state*, *final state(s)*, and the *state transition function*. The **size of state space** of an FSM, or all possible items in its transition table, $S_\Omega(FSM)$, can be determined by the product of both sizes of the sets of state S and alphabet $\sum$., i.e., $S_\Omega(FSM) = \#S \bullet \#\sum$.

- The differences of **requirement elicitation** and **system specification** in software engineering are that the former is focused on desired functions of a system δ, and the latter is on entire behaviors of the system Ω, including both δ and the undesired but potential functions $\overline{\delta}$ in the behavioral space $S_\Omega(FSM) = \#S \bullet \#\sum = \#(\delta) + \#(\overline{\delta})$. For a complex software system, the size of undesired behavior space is far more greater than that of the desired ones, i.e., $\#(\overline{\delta}) \gg \#(\delta)$.

- Automata and FSMs are a generic computing model for the rigorous description of event-driven behaviors of finite state systems. However, according to Theorem 5.1, the **efficiency of automata** decreases sharply when the size of a system is getting large and complicated. Therefore, FSMs are most useful for modeling computing behaviors at the component level.

- A **Turing Machine** (TM) is a 6-tuple encompassing the *alphabet*, *states*, *initial state*, *halting states*, *head movements*, and the *state transition function*. TM encompasses three basic components: the finite-state *control unit*, the *tape* (memory), and the *read/write head*. TM is the simplest model of computing and machine intelligence. Any complicated computing machine can be reduced to a number of basic TMs. This provides a practical approach to build large and complicated systems based on simple TMs. Although there are a variety of Turing machines, it can be proven that all Turing machines are equivalent.

- Turing's contribution is the identification of the basic requirements for computing and machine intelligence. The **essences of computing** are those of *a finite memory, a simple addressing capability for searching information in the memory, I/O operations on the memory,* and *evaluation or quantitative assessment capability*. TM theory reveals that intelligence is *memory-based*.

- **TM vs. FSM:** A TM extends the descriptive power of TM to both the output operation on the tape and the head actions associated to a state transition. An FSM is a restricted TM where the head is read-only and shift only from left to right. When a state of the FSM is a process that may be able to carry out any kind of operations, then FSM is equivalent to TM. This is why FSM is still widely applied in software engineering.

- A **von Neumann Machine** (VNM) consists of five components: the *arithmetic-logic unit* (ALU), the *control unit*, the *memory*, a set of *I/O devices*, and a *bus* that provides a data path between these components.

- The key requirements for implementing a VUM, the **stored-program controlled computer**, are the generalization of common computing architectures and the computer is able to interpret the data loaded in memory as computing instructions.

- Trends in **advanced computer architectures** beyond VNM are *parallel, networking* (*distributed*)*,* and *autonomic computers*.

- **Data object modeling** is a process to creatively elicit and abstractly represent a real-world application by logical data objects and their relations based on the constraints of given computing resources. **Operational behavioral modeling** is composed dynamic operations embodied onto the data objects. **Program architectural modeling** provides encapsulation methodologies for integrating and coordinating computing behaviors and data objects into a coherent system. **Resource and process modeling** deals with system platforms, operating resources, as well as the dynamic deployment of system data objects, architectures, and behaviors.

Data Object Modeling and Manipulation

- **Dada object modeling** is at the center of all profound computing techniques that studies how real-world entities and their relations are represented and modeled by a set of given data structures and construct rules in a programming language.

    - The **abstract representation** of any data object can be reduced to the fundamental level of *binary digits*.

    - The **logical models** of basic data objects in computing can be represented by *identifiers, variables, constants, and expressions*.

    - The **physical models** of data objects are allocated in memory as static or dynamic data.

- An **identifier** is a logical name of a language entity or construct that represents variables, constants, procedures, classes, or program names from the bottom up.

    - **Type** is the most important attribute of an identifier.

    - **Binding** is a process that associates an attribute to an identifier.

    - The **scope** of an identifier is a region in a program over which the binding between the identifier and the attribute is declared.

- A **variable** is an identifier that its domain of defined value is multiple and changeable. A variable obtains its value through **assignment**.

- A **constant** is an identifier that its value is fixed and read-only. A constant obtains its value through **declaration** rather than assignment.

- An **expression** is a relation between a set of operands (variables or constants) formed by relational operators in a given language. An expression can be classified as *logical, ordinal, numerical, timing,* and *architectural*, according to the type of its value in **BL**, **N**, **R**/**Z**/**S**/**B**/**H**/**P**, **TM**, and **ST**, respectively.

- Addressing is one of the most important and special operations in computing. **Addressing** is a function π: $id\mathbb{T} \rightarrow ptr$Þ$\mathbf{P}$ that maps a given logical $id\mathbb{T}$ into the physical memory block MEM[$ptr\mathbf{P}$, $ptr\mathbf{P}+n$-1]$\mathbb{T}$, and $\mathbb{T} \in \{\mathbf{P}, \mathbf{H}, \mathbf{N}, \mathbf{Z}\}$. Addressing locates the address of a given data object in the memory space.

- Addressing can be classified as **absolute** and **relative**, where the letter can be further divided into **direct** and **indirect** addressing dependent on whether an address is directly provided or indirectly inferred.

- **Memory allocation** is a binding process that associates a logical name with a physical location in the memory. The key difference between dynamic and static memory allocation is whether the size of memory requirement of a given variable is *run-time* or *compile-time* determinable.

  - **Dynamic memory allocation** is a binding process that associates a logical name of complex data objects consisting of multiple similar elements with a series of inter-linked physical locations in the heap during run-time, when the unit size of a given element and the number of elements are determinable.

- Types are an important logical property shared by data objects in programming. A **type** is a set in which all member data objects share a common logical property or attribute, and a type implies a set of allowable operations on data specified in this type. A **type system** specifies the type rules of a programming language as that of a grammar system which specifies the grammar rules of the language.

- The **RTPA type system** $\mathfrak{T}$ encompasses 17 primitive types as follows:

$$\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @e\mathbf{S}, @t\mathbf{TM}, @int\odot, ⓢs\mathbf{BL}\}$$

- The **type-suffix convention** of RTPA attaches every identifier of variables, constants, and expressions with a type in bold in the format of $id\mathbb{T}$, $\mathbb{T} \in \mathfrak{T}$.

- Important **derived types** in RTPA on the basis of the primitive types are those of *run-time, time, event, status,* and *CLMs*.

- A special set of complex types known as the **system types** that are widely used for modeling system architectures, particularly real-time, embedded, and distributed systems, such as *system components, system clocks, I/O interfaces, device drivers, interrupt sources, real-time events,* and *communication sockets*.

- The **Component Logical Model** (CLM) is an abstract model of a system architectural component that represents a hardware interface, an internal logical model, and/or a common control structure of a system.

- An **Abstract Data Type** (ADT) is a logical model of data objects, which defines the logical architecture and valid operations of the data object. ADTs extend type construction techniques by encapsulating both data structures and functional behaviors. The interface and implementation of an ADT can be separated.

- A **formal type system** is a collection of type rules for a given programming language. A **type rule** is an assertion of the validity of a judgment's conclusion on a type $\Theta_t \vdash A$ based on the inference of a number of premise judgments $\Theta_t \vdash A_i$, where Θ_t is the given type environment.

Operational Behavioral Modeling and Manipulation

- **Behaviors** of programs and software systems are observable computing effects and consequences on the data objects. **Fundamental computing behaviors** shared by various instruction sets of computers can be classified into eight categories, such as *data manipulations, arithmetical operations, logical operations, bitwise operations, program controls, memory manipulations, I/O manipulations, and interrupt and time manipulations*.

- **Basic Control Structures** (BCS's) are a set of essential flow control mechanisms that are used for building logical architectures of software. The most commonly identified BCS's in computing are known as the *sequential, branch*, *iteration, procedure call, recursion, parallel,* and *interrupt* structures. The BCS's provide essential compositional rules for programming. Based on them, complex computing functions and processes can be composed.

- A **Control Flow Graph** (CFG) is a directed graph model of program control structures, where a block of sequential instructions is abstractly represented by an edge, a branch BCS is denoted by two fan-out edges, and an iteration BCS is represented by a branch and sequential BCS's. When a program is abstracted by a CFG, the architecture of the problem is reduced to a graph where well-defined graph theory can be used to analyze its properties and complexity.

- The importance of **iterations** in computing is rooted in the basic need for effectively describing recurrent and repetitive software behaviors and system architectures. Based on the inductive property of iterations, the **big-R notation** is introduced to unify all types of iterations including the *while* (R^*), *repeat* (R^+), and *for* (R^i) loops.

- **Recursion** is an embedded process relation in which a process *P* calls itself, i.e. P ↻ P. The mechanism of recursion is a series of embedding (deductive, denoted by ↻) and de-embedding (inductive, denoted by ↺) processes. A recursive process should be terminable or noncircular. Recursions are used to model not only repetitive behaviors of systems, but also many fundamental language properties in computing.

- **Interactive behavior modeling** in computing encompasses *external interface* (*I/O*) *manipulation, memory manipulation, operating event handling, timing event handling, interrupt handling,* and *exception handling*.

Program Modeling: Coordination of Computational Behaviors with Data Objects

- Typical **program modeling technologies** are *statements, algorithms, classes, components, patterns,* and *frameworks* from the bottom up.

- A **statement** *S* is a function, or process, *P*, that maps a set of input *I* into a set of output *O*, i.e., $S = P: I \rightarrow O$. A statement is the smallest functional unit in programming that specifies an explicit action and results in the change of one or more variables.

- A **program** *P* is a finite list of instructive statements *S* that describes the computational behaviors, a set of data objects *D* that model the internal and external environment, and their interactions *F* that result in the change of the data objects, i.e., $P = (S, D, F)$. A program is a finite set of **cumulated relations** between all statements.

- An **algorithm** $\mathit{\Lambda}$ is a frequently recurring function *f* that maps a set of input *X* into a set of output *Y* by a finite set of statements or a finite-step process, i.e., $\Lambda = f\text{: X} \rightarrow \text{Y}$. The characteristics of algorithms are reusability, finite process, and efficiency.

- A **class** is a computational construct that models a set of data objects and predefined behaviors or operations on them by an integrated encapsulation and an abstracted interface.

- An **object** is an instance of a given class. An object forms an abstract model of a real world entity and/or a computational module, which is packaged by an integrated structure of interface and implementation, and is described by methods for its functions and by data structures for its architecture and attributes.

- **Object-oriented technologies** can be commonly identified as encapsulation, inheritance, reusability, and polymorphism.

- A **pattern** is a complex computational construct that incorporates a set of classes for a recurring architectural and behavioral design described by *abstract classes*, *concrete classes*, *instantiations*, and their *associations*.
  - Software patterns (Definition 5.63) are a new component modeling technology built upon classes and object-oriented techniques. As a set of interacting classes, patterns can be used as a powerful tool for capturing software design notions and best practices, which provide common solutions to core problems in software development.

- A **framework** is an architectural model of an entire system that represents the overall structure, components, processes, and their interrelationships and interactions.
  - Frameworks are the top level computational construct built upon algorithms, components, classes, and/or patterns in an object-oriented or component-based approach. Framework technology enables domain and design knowledge to be reused as well as that of code.
  - A framework permits a new technology known as **template-based programming**.

Resources and Processes Modeling and Manipulation

- The coordination of computing resources and processes is manipulated at the operating system level. A *program* and its *behavior space* and *semantic environment* are realized by a target computer, which can be modeled by the **Generic Computing System** (GCS) §.

- An **operating system** is a type of system software that manages and controls the resources and computing capability of a computer or a computer network, and provides users a logical interface for accessing the physical computer to execute applications. The general-purpose operating systems can be classified into four types: *the batch systems, time-sharing systems, real-time systems,* and *distributed systems*.

- A **Virtual Machine** (VM) is a subset of an operating system that represents various computing resources to the users in a unified manner, and hides hardware differences and physical implementation details at the lower layers.

- A **generic operating system** encompasses the *kernel* and the *resource management* subsystems. The former is a set of central components for computing, including CPU scheduling and process management. The latter is a set of individual supporting software for various system resources and user interfaces.

- The **kernel** encompasses the interrupt handler, the task manager, and the inter-process communication manager, the virtual memory manager, and the network subsystem manager.

- The **services** provided by an operating system can be classified into categories of *task control, file manipulation, device control,* and *information maintenance*.

- Basic computing **resource manipulations** enabled by an operating system can be classified as *process and thread management, memory management, file system management, I/O system management,* and *network/communication management*.

- A **Real-Time Operating System** (RTOS) is an operating system that guarantees timely processing of external and internal events of real-time systems. RTOS requires multitasking, process threads, and explicit interrupt levels to deal with real-time events and interrupts. An RTOS is essential to implement embedded and/or real-time control systems.

Questions and Research Opportunities

5.1 On the basis of Table 5.1, analyze and contrast the orientations and focuses of problems in software engineering and computer science in terms of their objects, methods, and resources.

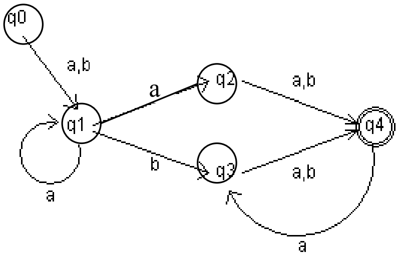
5.2 Why are *binary digits* (*bits*) treated as the most fundamental form of data object representation in computing? Why do computer science and information science share the same fundamental object form – bits?

5.3 Why are the most fundamental problems in software engineering about how the complex data objects and complicated mathematical and behavioral operations on them may be reduced to *bits* and *bits-based operations*?

5.4 Why are the most fundamental computational operations logical, arithmetic, and memory access operations on bits?

5.5 What are the different orientations on the usage of automata in computer science and software engineering?

5.6 Formally define the following FSM according to Definition 5.1.



5.7 According to Theorem 5.3, most software systems go wrong not because they are incorrect on normally required functions, but because there are wrong or not prepared for implied or nonspecified exceptions.

Analyze the following properties of the FSM as given in Ex.5.6:

a) The entire behavior space Ω of the FSM;

b) The required behavior space δ of the FSM;

c) The unspecified behavior space $\overline{\delta}$ of the FSM;

d) The ratio between the specified behavior space and the entire behavior space, i.e., (δ / Ω) • 100%.

5.8 What are the limitations or weaknesses of FSMs in computing and software engineering? What kind of software engineering problems cannot be dealt with FSMs?

5.9 Describe the extensions of a *Turing machine* (*TM*), $TM \triangleq (\Sigma, S, s, H, M, \delta)$ and its transition function δ over the structure of an FSM.

5.10 According to Theorem 5.5, what are the key findings of Turing on the *fundamental computational capabilities*?

5.11 It is perceived that virtually any complex computing activities can be reduced to a Turing machine. Based on this assertion, discuss if Turing machines are either more *fundamental* or more *powerful* in modeling computing structures.

5.12 What are the limitations or weaknesses of Turing machines in computing and software engineering? What kind of software engineering problems cannot be dealt with Turing machines?

5.13 What are the extensions of von Neumann machines over Turing machines?

5.14 Stored-program techniques unified data and ___?___ in memory. How?

5.15 Describe the architecture of a VNM machine:

$$VNA \triangleq (ALU, CU, M, I/O, B).$$

5.16 What are the limitations or weaknesses of VNMs in computing and software engineering? What kind of software engineering problems cannot be dealt with VNMs?

5.17 Discuss what would be the potential computing theories and techniques towards the development of non-VNMs.

5.18 What is a data type and what are types' usages in software engineering and computing?

5.19 Summarize the 17 primitive types of RTPA, and highlight the special types that are not modeled in exiting programming languages.

5.20 According to Theorem 5.6, the type domains of mathematics D_m, language D_l, and user defined D_u obey the following law: $D_u \subseteq D_l \subseteq D_m$. On the basis of this theorem, explain why the precedence of domain determination must be $D_u \Rightarrow D_l \Rightarrow D_m$ in software engineering.

5.21 The type system of Pascal is given in Fig. 5.8. Referring to Table 5.12, try to formally define the type system of Pascal according to Definition 5.34.

5.22 The type system of Java is given in Fig. 5.9. Referring to Table 5.12, try to formally define the type system of Java according to Definition 5.34.

5.23 The type system of IDL is given in Fig. 5.10. Referring to Table 5.12, try to formally define the type system of IDL according to Definition 5.34.

5.24 Build an informal type system model for C++ as that of Fig. 5.9. Then, develop a formal type system model of C++ according to Definition 5.34 and Table 5.12.

5.25 The schema of a function in programming languages can be formally modeled as a complex type according to type theory. Referring to Definition 5.36, try to develop a formal type rule for the function type, Func**ST**, for Java.

5.26 The schema of a class in UML may be modeled as three components known as the _ClassID_**S**, _Attributes_**RT**, and _Methods_**ST**. Try to define a formal type rule, _Class_**ST**, for UML classes.

5.27 What is the usage of the *run-time type* **RT** as modeled in RTPA? What are the counterparts in programming languages?

5.28 What is the usage of the *system (structural) type* **ST** as modeled in RTPA? What are the counterparts in programming languages?

5.29 The architectural model of an ADT, _Queue_**ST**, is partially specified as follows:

$$
\begin{aligned}
\text{Queue.Architecture}\mathbf{ST} \triangleq \text{Q}\mathbf{ST} :: \; & (\langle \_ : \mathbf{N} \mid \text{size}\mathbf{N} \geq \_ \rangle, \\
& \langle \_ : \_ \rangle, \\
& \langle \text{CurrentPos} : \_ \mid \_ \leq \_\mathbf{P} \leq \_ \rangle \\
&)
\end{aligned}
$$

a) Try to complete the specification by providing proper values or types for the eight blank places marked by _.

b) Draw a diagram to show the corresponding conceptual model of the queue.

5.30 How is a port or system interface modeled by the system structure PORT**ST** in RTPA? What are the counterparts in programming languages?

5.31 What are the characteristics modeled in the formal definition of an identifier?

5.32 Referring to that of a *variable* obtains or changes its value through assignments, how does a *constant* obtain its value? Is the binding between the constant and its value permanent or temporary?

5.33 BCS's are a set of essential flow control mechanisms that are used for constructing logical architectures of software systems. What are the relations of the 10 BCS's and the 17 process relations (operations) modeled in RTPA?

5.34 Comparatively analyzing the linear and nested structure of iterations and recursions, and corresponding formal denotations in RTPA, describe the role of the big-R notation.

5.35 Draw a conceptual model of a node in the digraph, *Node***S**, based on the following RTPA specification:

$$\begin{array}{l}
\textbf{DiGraph}\mathbf{ST}\textbf{.Architecture.NodeCLM}\mathbf{ST} \triangleq \text{Node}\mathbf{S} :: \\
\quad (\ <\text{Element} : \mathbf{RT}>, \\
\quad\quad <\text{PriorPtr} : \mathbf{P}>, \\
\quad\quad <\text{NextPtr} : \mathbf{P}>, \\
\quad\quad <\text{Order}: \mathbf{N} \mid 0 \leq \text{Order}\mathbf{N} \leq \text{SizeofEdges}\mathbf{N}>, \\
\quad\quad < \mathop{\mathit{R}}\limits_{i\mathbf{N}=1}^{Order\mathbf{N}} \ \text{Edge}(i\mathbf{N}) : \mathbf{S}>, \\
\quad\quad < \mathop{\mathit{R}}\limits_{i\mathbf{N}=1}^{Order\mathbf{N}} \ \text{Weight}(i\mathbf{N}) : \mathbf{N}>, \\
\quad)
\end{array}$$

5.36 According to Definition 5.51, i.e.:

$$P = \mathop{R}\limits_{i=1}^{n-1}(s_i\ r_{ij}\ \mathrm{s}_j), j = i+1 = (...(((s_1)\ r_{12}\ \mathrm{s}_2)\ r_{23}\ \mathrm{s}_3)\ ...\ r_{n-1,n}\ \mathrm{s}_n),$$

explain the physical meaning of a process in programs. (Hint: Refer to Theorem 4.3 on the cumulative relational processes.)

5.37 Explain Theorem 5.7, the *generic mathematical model of programs*, i.e.:

$$\wp = \mathop{R}\limits_{k=1}^{m}(@e_k\mathbf{S} \hookrightarrow P_k) = \mathop{R}\limits_{k=1}^{m}[@e_k\mathbf{S} \hookrightarrow \mathop{R}\limits_{i=1}^{n-1}(s_i(k)\ r_{ij}(k)\ \mathrm{s}_j(k))], j = i+1$$

and demonstrate how it fits an example program in any programming language.

5.38 Refer to Section 5.5.2.3 and explain how the generic mathematical model of *patterns* is applied to the *Builder* pattern deductively.

5.39 The current design and implementation of software patterns are based on object-oriented technologies. Are software patterns independent of technologies and programming languages? Why?

5.40 Refer to Section 5.5.3 and explain how the framework of the Telephone Switching System (TSS) is modeled by TSS.Architecture**ST**, TSS.StaticBehaviors**ST**, and TSS.StaticBehaviors**ST** in RTPA.

5.41 Discuss how an operating system may be described by the *Generic Computing System* (GCS), §, as given in Definition 5.56.

5.42 Read the following classic article in software engineering:

> John E. Hopcroft (1987), Computer Science: The Emergence of a Discipline, The 1986 Turing Award Lecture, *Communications of the ACM*, 30(3), pp.198-202.

Discuss the following topics in a group:

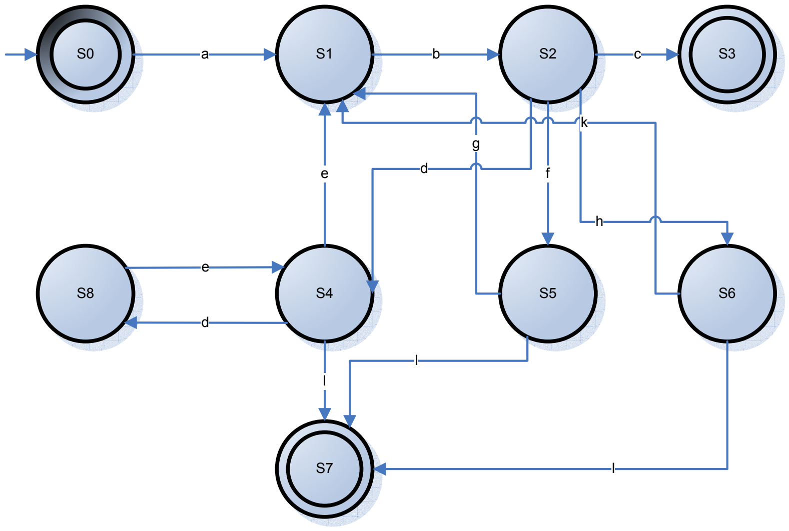
- About the author.

- What was the nature of computer science according to the author in the 1980s?
- What is the architecture of computer science as a scientific discipline?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

5.43 Using the state transition diagram (STD) of RTOS+ as shown below as an example, try to solve the following problems:

a) Formally define the STD of RTOS+ as an FSM.

b) Analyze the size of the behavioral space Ω of the FSM.



where

s_0: Creating
s_1: Ready
s_2: Running
s_3: Completed
s_4: Interrupted
s_5: Delayed
s_6: Suspended
s_7: Killed
s_8: Interrupt Services

a: Resource available
b: Schedule
c: Completed
d: Interrupt
e: Interrupt return
f : Time-out
g: Time available
h: Wait for event
k: Event available
l: Delete

5.44 Read the following classic article in software engineering:

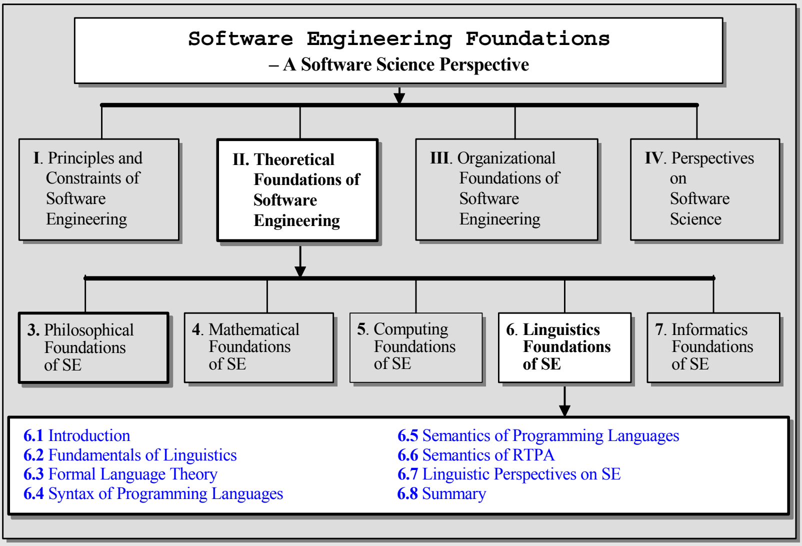
> Erich Gamma (2002), Design Patterns – Ten Years Later, in M. Broy and E. Denert *eds.*, *Software Pioneers,* Springer, Berlin, pp. 688 – 700.

Discuss the following topics in a group:

- About the author.
- Where does the concept of software design patterns comes from?
- Why are design patterns considered useful in software engineering?
- What are the limitations of patterns in software reuse?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 6

LINGUISTIC FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations

– A Software Science Perspective

I. Principles and Constraints of Software Engineering

II. Theoretical Foundations of Software Engineering

III. Organizational Foundations of Software Engineering

IV. Perspectives on Software Science

3. Philosophical Foundations of SE

4. Mathematical Foundations of SE

5. Computing Foundations of SE

6. Linguistics Foundations of SE

7. Informatics Foundations of SE

6. Linguistics Foundations of SE

Knowledge Architecture

❍ Fundamentals of linguistics

- Taxonomy of linguistics
- Semantics
- Syntaxes
- Grammars

❍ Formal language theory

- Alphabets
- Expressions
- Languages
- Language recognitions
- Strings
- Grammar theories
- BNF and EBNF

❍ Syntax of programming languages

- Lexical analyses
- Syntactical analyses
- Syntax definition and descriptions
- Syntactical analyses of RTPA

❍ Semantics of programming languages

- Taxonomy of semantics
- Denotational semantics
- Deductive semantics

❍ Semantics of RTPA

- Semantics of RTPA meta processes
- Semantics of RTPA process relations
- Semantics of system and system process dispatching

❍ Linguistics perceptions on software engineering

- Comparative analysis of natural and programming languages
- Principles of programming language design
- Characteristics of programming languages

Learning Objectives

- To understand the role of linguistics and fundamental principles of linguistics in software engineering.
- To be aware of the formal language theory and the formal grammar system.
- To be familiar with formal syntaxes and semantics.
- To understand the BNF and EBNF notations for language specification and modeling.
- To understand deductive semantics and its applications in RTPA modeling.
- To be familiar with applications of linguistics in software engineering.

"The gift of language is the single human trait that marks us all genetically, setting us apart from the rest of life."

Lewis Thomas (1974)

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature."

D.E. Knuth (1984)

6.1 Introduction

Linguistics is the discipline that studies human or natural languages. Languages are an oral and/or written symbolic system for thought, self-expression, and communication. Lewis Thomas highlighted that "the gift of language is the single human trait that marks us all genetically, setting us apart from the rest of life [Thomas, 1974]." This is because functions of languages can be identified as for memory, instruction, communication, modeling, thought, reasoning, problem-solving, prediction, and planning [Pattee, 1986; Casti and Karlqvist, 1986].

Linguists commonly agree there is a universal language structure or grammar [Chomski, 1956/57/59/62/65/82; Pattee, 1986; O'Grady and Archibald, 2000]. However, the grammar may be precise and explicit as in formal languages, or ambiguous and implied as in natural languages. Although a language string is symbolically constructed and read sequentially, all natural languages have the so called metalinguistic ability to reference themselves out of the sequences. That is, to construct strings which refer to other strings in a language.

This chapter comparatively studies natural and artificial languages, and explores fundamental theories of linguistics, language acquisition, and applications. Then, it extends linguistics to artificial languages, particularly programming languages, which investigate the theory of formal languages and the applications of mathematics in computational linguistics.

It is noteworthy that a natural language is *context sensitive*. While almost all programming languages, no matter at machine level or higher level, are supposed to be *context free*. Therefore, it is curious to query if a real-world problem and its solution(s), in a context-dependent manner, can be described by a context-free programming language without losing any information. Automata and compiler theories [Rabin and Scott, 1959; Aho et al. 1985; Louden, 1993; Lewis and Papadimitriou, 1998] indicate a context-sensitive language may be transformed into a corresponding context-free language. But the costs to do so are really dear, because the context cannot be freely removed. A common approach is to hide (imply) the context of software in data objects and intermediate data structures in programming. However, the drawbacks of this convention, or the limitations of conventional compiling technologies, make programming hard and complicated, because the computational behaviors and their data objects were separated or incoherent in the languages' descriptive power. This is an indication that a much natural and context-dependent programming language and related compiling technology are yet to be sought. We may consider that ADTs and object-oriented programming technologies are context-dependent, because the context (in the form of a set of data objects) has been encapsulated into a set of individual classes and the whole class hierarchy of a software system.

From a linguistic point of view, software engineering is the application of information technologies in communicating between a variety of stakeholders in computing, such as professionals and customers, architects and software engineers, programmers and computers, as well as computing systems and their environments. Therefore, linguistics and formal language theories may play important roles in computing theories; without them computing and software engineering theories would not be complete.

It is noteworthy that, historically, *language-centered programming* had been the dominant methodology in computing and software engineering. However, this should not be taken as granted as the only approach to software engineering, because the expressive power of programming languages is inadequate to deal with complicated software systems. In addition, the rigorousness and level of abstraction of programming languages are too low in modeling the architectures and behaviors of software systems. This is why a bridge in mechanical engineering or a building in civil engineering was not modeled or described by natural or artificial languages. This observation leads to the recognition of the need for *mathematical modeling* of both software *system architectures* and *static/dynamic behaviors*, supplemented with the support of automatic *code generation systems*.

This chapter analyzes not only how linguistics may improve the understanding of programming languages and their work products – software, but also how formal language theories extend the study of natural languages. In the remainder of this chapter, the linguistics foundations of software engineering will be presented in six sections. Fundamental theories of linguistics are reviewed in Section 6.2 on syntaxes, semantics, grammars, and linguistic analyses. Formal language theories that provide a rigorous treatment of language elements from the bottom up are described in Section 6.3. Syntaxes and semantics of programming languages and their analyses are presented in Sections 6.4 and 6.5 with the introduction of a formal semantics theory known as deductive semantics. Semantics of RTPA are formally described in Section 6.6 using deductive semantics. Comparative analyses of natural and programming languages, as well as linguistics perceptions on software engineering, are discussed in Section 6.7.

6.2 Fundamentals of Linguistics

Linguistics studies natural languages in both oral and written forms. Since languages are the basic means of human communication and tools of thinking and expression, linguistics may be perceived as one of the foundations of computing, software engineering, and information sciences. This section surveys the basic theories and principles of linguistics, which form a reference system for the study of artificial, programming, and formal languages in software engineering.

6.2.1 TAXONOMY OF LINGUISTICS

The basic function of languages is both to *communicate information* and to *express* abstract human *behaviors*. The central issue of linguistics is *grammar*, which is the rules of a language and the ways how the language is to be generated, formed, recognized, and interpreted.

Definition 6.1 *Linguistics* is a discipline that studies the nature and use of languages.

The domain of linguistics encompasses phonetics, phonology, morphology, syntax, and semantics. The first three facets of linguistics are introduced below, while syntax and semantics will be formally described in Sections 6.2.2 and 6.2.3.

Definition 6.2 *Phonetics* is a domain of linguistics that studies the articulation and perception of sounds of human speech.

The sounds of any language can be categorized into two types known as *syllabic* and *nonsyllabic* sounds. The former are those of vowels, syllabic liquids, and syllabic nasals; the latter are those of consonants and glides. It is interesting to observe that all sounds of human languages can be widely transcribed by the standard *international phonetic alphabet* defined by the International Phonetic Association [IPA, 1970].

Definition 6.3 *Phonology* is a domain of linguistics that studies the patterns of speech sounds.

Three units of phonological representation, known as the *feature*, *phoneme*, and *syllable*, are adopted in the bottom-up phonemic analyses. For a classic presentation of phonemic analysis, readers may refer to H. A. Gleason, Jr.'s *An Introduction to Descriptive Linguistics* [Gleason, 1961].

Definition 6.4 *Morphology* is a domain of linguistics that studies the formation and structure of words.

Words are the smallest *free form* and basic building blocks of a language, where the element of words is *morphemes*. Basic word formation techniques are *derivation* and *compounding* in English. A basic word may be inflected to mark grammatical contrasts in number, person, gender, case, and tense. The mental dictionary acquired by a person is called *lexicon*, which contains a collection of information about the syntactic properties, meaning, and phonological representation of words in a language. Interested readers on morphology may refer to Jensen (1990) or Spencer (1991).

In line with the theme of this chapter on comparative linguistics between natural and programming languages, the remainder of this section will put emphases on syntax and semantics of natural languages, because programming languages in nature are written languages rather than speaking ones.

6.2.2 SYNTAXES

Syntaxes deal with relations and combinational rules of words in sentences. Much of our understanding of the syntactic rules of languages has come from linguists who have studied and elicited the common rules that underlie the standard language. One of the most influential linguistic frameworks, known as the theory of *universal grammar*, was proposed by Noam Chomsky [Chomsky, 1957/65]. Universal grammar and its modern version, the *Government and Binding Theory* [Chomsky, 1982], have become a linguistic premise on grammatical analysis in linguistics, which will be discussed in Section 6.2.4.

Definition 6.5 A *syntax* is a domain of linguistics that studies sentence formation and structures.

Definition 6.6 An *abstract syntax* is the abstract description of a syntax system where concrete strings of tokens and their grammatical relations are represented and analyzed symbolically.

Linguistic studies are used to the convention of hierarchical tree schema to denote sentence structures in syntactical analyses. In a syntactic perspective, any human language, natural or artificial, is a sequential or one-dimensional (1-D) symbol stream of syntactical blocks, which can be decomposed into paragraphs, sentences, phrases, words, and letters from the top down. Although the syntax of a language is 1-D, its grammar is recursively structured in a 2-D space.

Lemma 6.1 *Syntaxes of natural languages*, *Syn*, are 2-dimensionally descriptable and are recursive.

However, the semantics of languages implied by the sequential syntaxes can be more complicated, i.e., non-sequential and multi-dimensional in most cases, such as branch, parallel, embedded, concurrent, interleaved, and interrupt structures as shown in Table 6.1 [Wang, 2007*l*].

Table 6.1
Semantic Relations of Sentences

No.	Relation	Formal Symbol	Description
1	Sequential	→	and, then
2	Branch	∣	or
3	Parallel	∥	and, simultaneously (action by the same subjects)
4	Embedded	↳	that, which, if, whether
5	Concurrent	∯	and, simultaneously (action by different subjects)
6	Interleave	⦀	Alternatively
7	Interrupt	↯	when, while, during

Table 6.1 indicates that the *semantic relations of sentences* are a set of connectors, which are a subset of the 17 process relations as defined in RTPA [Wang, 2002a/02b/03c/06a/07a].

Syntactic elements in natural languages can be classified into the categories of lexical, functional, phrasal, and relational. A summary of definitions of syntactic elements of languages is provided in Table 6.2, where an element in angular brackets is optional. In Table 6.2, there is special category of lexical components known as *complement phrases* (CPs). CPs can be a supplemental part of N/NP, V/VP, A/AP, or P/PP. The rules for defining relations between CPs and other lexical categories of sentences may be referred to O’Grady and Archibald (2000).

Table 6.2
Definition of Lexical Categories of languages

Category	Sub Cat.	Symbol	Description
Lexical			
	Noun	N	entities and abstract objects
	Verb	V	actions, states, and possessions
	Adjective	A	properties of a noun
	Adverb	Λ	properties of a verb
	Preposition	P	designates relations in space or time
Functional			
	Determiner	τ	the, a, this, these, etc.
	Degree word	δ	too, so, very, more, quite, etc.
	Qualifier	κ	almost, always, often, perhaps, never, etc.
	Auxiliary	α	will, can, may, must, should, could, etc.
	Conjunctor	γ	and, or, that, which, if, whether, etc.
	Negative	¬	not
Phrasal			a syntactic unit with one or more words as a lexical category
	Noun phrase	NP	τ N [PP]
	Verb phrase	VP	V NP etc.
	Adjective phrase	AP	[δ] A [PP]
	Adverb phrase	ΛP	[Λ] V | V [Λ]
	Prepositional phrase	PP	[δ] P [NP]
	Complement phrase	CP	supplemental part of N (NP), V (VP), A (AP), or P (PP)
Relational		ℛ	A set of connectors
	Sequential	→	and, then
	Branch	|	or
	Parallel	||	and, simultaneously (action by the same subjects)
	Embedded	↣	that, which, if, whether
	Concurrent	∯	and, simultaneously (action by different subjects)
	Interleave	|||	alternatively
	Interrupt	↯	when, while, during

Based on the definitions of lexical functions of words and phrases, the syntactic structure of sentences can be described formally as shown in Fig. 6.1. In Fig. 6.1, the *D-structure* represents relationship between subject and object in a sentence, while the *S-structure* represents the surface linear arrangements of words in a sentence [Gleason, 19961/97]. The *S*-structure consists of the sound structure of a sentence called the *phonetic form* and the meaning of the sentence called the *logical form*. The *D*-structure consists of a set of phrase structure rules and the lexicon that specifies the morphophonological and syntactical features of the sentence.

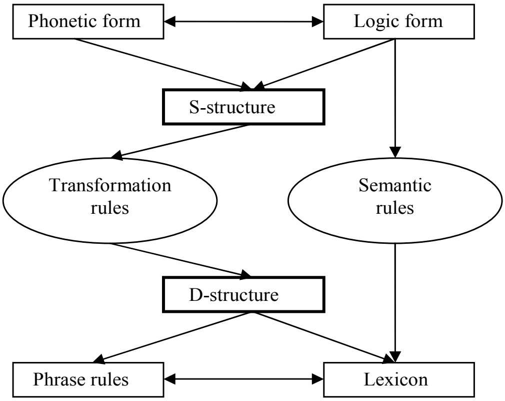


Figure 6.1 Relationships among components of universal grammar

6.2.3 SEMANTICS

Definition 6.7 *Semantics* is a domain of linguistics that studies the interpretation of words and sentences, and analysis of their meanings.

Semantics deals with how the meaning of a sentence in a language is obtained, hence the sentence is comprehended. Studies on semantics explore mechanisms in the understanding of language and the nature of meaning where syntactic structures play an important role in the interpretation of sentence and the intension and extension of word meaning [Tarski, 1944; Chomski, 1956/57/59/62/65/82].

Definition 6.8 The *mathematical model of semantics* of natural languages, *Sem*, is a 5-tuple, i.e.:

$$Sem \triangleq (J, B, O, T, S) \tag{6.1}$$

where

- *J* is the subject of the sentence;
- *B* is a behavior or action;
- *O* is the subject of the sentence;
- *T* is the time when the action is occurring;
- *S* is the space where the action is occurring.

According to Lemma 6.1 (*Syn*) and Definition 6.8 (*Sem*), the relationship between a language and its syntaxes and semantics can be illustrated as shown in Fig. 6.2. Fig. 6.2 explains that linguistic analyses are a deductive process that maps the 1-D language into the 5-D semantics of natural languages via the 2-D syntactical analyses [Wang, 2007h/07j].

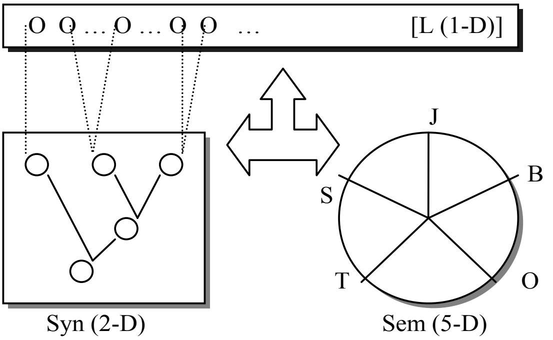


Figure 6.2 The Universal Language Processing (ULP) model

Semantic analysis and comprehension are a deductive cognitive process. According to the OAR model as developed in Section 9.4, the semantics of a sentence may be considered having been understood when: a) The logical relations of parts of the sentence are clarified; and b) All parts of sentence are reduced to the terminal entities, which are either a real-world image or a primitive abstract concept. The theoretical foundations of language cognition and comprehension will be further discussed in Chapter 9.

6.2.4 GRAMMARS

Syntactic and semantic analyses in linguistics rely on a set of explicitly described rules known as the grammar of a language. Therefore, contemporary linguistic analyses focus on the study of grammars, which is centered in language acquisition, understanding, and interpretation.

Definition 6.9 The *grammar* of a language is a set of common rules that integrates phonetics, phonology, morphology, syntax, and semantics of a given language.

The grammar governs the articulation, perception, and patterning of speech sounds, the formation of words and sentences, and the interpretation of utterance.

6.2.4.1 Properties of Grammars

O'Grady and Archibald (2000) identified five basic properties of grammars as follows:

- **Property 1.** *Generality:* All languages have a grammar.
- **Property 2.** *Parity:* All grammars are equivalent in terms of their expressive capacity.
- **Property 3.** *Universality:* Grammars are commonly alike, or basic principles and properties are shared in all languages.
- **Property 4.** *Mutability:* Grammars of all languages are constantly changing over time.
- **Property 5.** *Inaccessibility:* Grammatical knowledge of the mother tongue is built at the subconscious layer of the brain.

The above basic properties of grammars form an important part of the foundations of human intelligence. The most interesting property of grammars of natural languages is their expressive parity.

Lemma 6.2 All grammars of natural languages are equivalent.

Based on Lemmas 6.2, it is perceived that, in computing and software engineering, all programming languages are equivalent. In other words, no language may claim a primitive status over others, as long as they implement the 17 meta processes and 17 process relations of RTPA as stated in Theorems 4.6, 4.7, and 5.7 in Sections 4.6 and 5.5.

6.2.4.2 The Universal Grammar

An important discovery in modern linguistics is the existence of the universal grammar among human languages.

Definition 6.10 The *universal grammar* (UG) is a system of categories, mechanisms, and constraints shared by all human languages.

UG is perceived as innate based on recent neurolinguistic and psycholinguistic studies [Chomsky, 1982; O'Grady and Archibald, 2000; Wang, 2007h]. UG treats all languages with the same generic type of syntactic mechanisms, which include the *merge* and *transformation* operations. The former is a syntactic operation that combines words in accordance with their syntactic categories and properties; while the latter is a syntactic operation that puts words and phrases in an appropriate structure.

An instance of UG is the English grammar, which may be formally described in the following section.

6.2.4.3 The Deductive Grammar of English

Formal language theories of computing science and software engineering perceive that the grammar of any programming language or professional notation systems may be rigorously defined by the EBNF notation [Naur, 1968/73]. The author found that the formal language theory can be extended to describe and analyze the grammar of natural languages such as that of English [Wang, 2007*l*].

Definition 6.11 The *deductive grammar* is an abstract grammar that formally denotes the syntactic rules of a language based on which as a generic formula valid language sentences can be deductively derived.

On the basis of the definitions of the syntactic elements as given in Table 6.2, the English grammar can be formally described in EBNF known as the Deductive Grammar of English (DGE) [Wang, 2007*l*]. A rigorous definition of DGE at the sentence level is given in Fig. 6.3. Some aspects of DGE are simplified at the bottom level, particularly on person rules of nouns, time rules of verbs, and the matching of nouns and verbs in sentences.

According to DGE, the schema of the most complicated sentence in English that consists of all possible and legal syntactic components of DGE is shown in Fig. 6.4. The generic schema of DGE can be used as a universal formula to deductively derive any sentence in English. For example, the shortest possible sentence is given in Example 1 in Fig. 6.4. The longest possible sentence is presented in Example 3, i.e.:

> *" The unregistered new student all in the class [and another phrase] will not get the expected comprehensive handbook directly from the teacher [or another sentence]."*

```
S ::=  [Subject] Predicate
        | S γ S
Subject ::= NP
Predicate ::= VP [Object]
Object ::= NP
NP ::=   τ [AP] N [PP]
            | τ N*
            | NP γ NP
AP ::= [Λ] A
ΛP ::=   [δ] Λ
           | [κ] Λ
PP ::= [Λ] P [NP]
VP ::=  VP γ VP
           | [α] [¬] V [Object]*
           | [ΛP] V [¬] [Object]*
           | V [¬] [Object]* [ΛP]
¬ ::= <not>
N ::= <nouns>
V ::=   <be>
          | <have>
          | <do>
P ::= <propositions>
A ::= <adjectives>
Λ ::= <adverbs>
δ ::= <degree words>
κ ::= <qualifier words>
α ::= <auxiliary words>
τ ::= <determiner words>
γ ::= <conjunction words>
        | <.>
        | <;>
```

Figure 6.3 The Deductive Grammar of English (DGE)

No.	S (Sentence)																						
	Subject										Action												
	NP										VP												
	NP								γ	NP	VP											γ	VP
	τ	AP		**N**	PP				γ	NP	α	¬	**V**	Object *								γ	VP
		Λ	A		Λ	P	τ	N		…				τ	Λ	A	N	Λ	P	τ	N		…
Ex.1													Look									.	
Ex.2				I									read	a			book					.	
Ex.3	a	b	b	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w

Note: Words in Sentence 3 are defined as follows:
a – The, b – unregistered, c – new, d – student, e – all, f – in, g – the, h – class, i – and, j – …, k – will, l – not, m – get, n – the, o – expected, p – comprehensive, q – handbook, r – directly, s – from, t – the, u – teacher, v – or, w – another sentence.

Figure 6.4 The schema of a generic sentence based on DGE

The above example provided in Fig. 6.4 is an instance that uses almost all possible syntactic components. Obviously, natural sentences in practical usages are always a subset of the DGE schema. Therefore, they are rather simple and short as shown in the first two examples in Fig. 6.4.

The 1-D structured sentences as shown in Fig. 6.4 can be modeled in a 2-D graphical form as shown in Fig. 6.5. Observing Figs. 6.3 through 6.5, it is noteworthy that the syntactic structure of the DGE schema is highly recursive. The recursive characteristics in Fig. 6.5 are repetitively represented by the none phrases (NP) and verb phrases (VP).

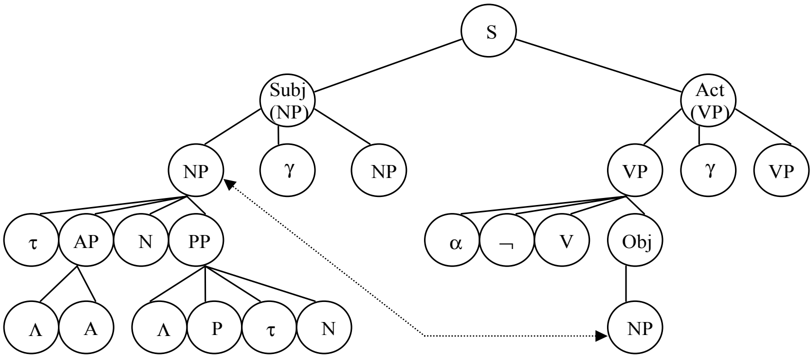


Figure 6.5 The syntax structure of the generic sentence schema in DGE

The 18th Law of Software Engineering

Theorem 6.1 The *tradeoff between syntaxes and semantics* states that in the DGE system, the complexities of the syntactic rules (or grammar) C_{syn} and of the semantic rules C_{sem} are inversely proportional, i.e.:

$$C_{syn} \propto \frac{1}{C_{sem}} \tag{6.2}$$

Theorem 6.1 indicates that the simpler the syntactic rules or the grammar, the richer or more complicated the semantics, and vice versa According to Theorem 6.1, since UG or DGE as defined in Fig. 6.3 are relatively simple, its semantics are much richer, more complicated, and more ambiguous. In contrary, because programming languages adopt very detailed and complicated grammars, their semantics are relatively concise, simple, and rigor.

The fundamental elements of natural languages can be classified as shown in Table 6.3 [Wang, 2002a/06j/07a]. Observing Table 6.3 it can be seen that although natural languages can be rich, complex, and powerfully descriptive, they share common basic structures, such as ‘to be (|=),’ ‘to have (|⊂),’ and ‘to do (|>).’

Table 6.3
Fundamental Elements in Natural Languages

<table>
<tr><th>Function</th><th>Category</th><th>Notation</th><th>Example</th></tr>
<tr><td>Identify <i>objects</i> and <i>attributes</i></td><td>To be</td><td>|=</td><td>A |= B ⇒
(A is B)</td></tr>
<tr><td>Describe <i>relations</i> and <i>possession</i></td><td>To have</td><td>|⊂</td><td>A |⊂ B ⇒
(A has B)</td></tr>
<tr><td>Describe <i>status</i> and <i>behaviors</i></td><td>To do</td><td>|></td><td>A |> B ⇒
(A does B)</td></tr>
<tr><td></td><td>Indirect to do</td><td>|>> … |></td><td>A |>> B |> C ⇒
(A has B to do C)</td></tr>
<tr><td>Describe negative facts</td><td>Not</td><td>¬</td><td>A ¬ |= B ⇒
(A is <i>not</i> B)
A ¬ |⊂ B ⇒
(A has <i>not</i> B)
A ¬ |> B ⇒
(A does <i>not</i> B)</td></tr>
</table>

The formal models of UG and DGE provide linguistics, particularly language analyzers, implementers, and recognizers, for a powerful tool to formally describe and process natural language documents. Perspective applications of DGE may be in the development of Internet searching engines, semantic analysis of natural languages, speech recognitions, and intelligent systems for natural language parsing and word processing.

6.3 Formal Language Theory

Natural languages studied in Section 6.2 are informal and nonrigorous. Every defined rule in the grammars of human languages has exceptions and their semantics depends on contexts, situations, and subjective perceptions.

In contrary, formal languages are theories and rules for rigorously specify, analyze, generate, and recognize programming languages. Formal language theories study the following objects hierarchically from the bottom up:

- Alphabets
- Strings
- Expressions
- Languages
- Grammars
- Machines capable to process formal languages

Software engineering may need to extend conventional formal language theories from language generation and recognition to software system *modeling* and *specification*.

6.3.1 ALPHABET

Any language, natural and artificial, for human or machine, is based on a set of symbols known as the alphabet.

Definition 6.12 An *alphabet* Σ is a nonempty finite set of symbols or letters.

Example 6.1 The following sets of symbols are typical alphabets:

a) Roman alphabet:	$\Sigma_R = \{a, b, \ldots, z, A, B, \ldots, Z\}$
b) The binary alphabet:	$\Sigma_B = \{0,1\}$
c) Digits:	$\Sigma_D = \{0,1, 2, \ldots, 9\}$
d) Operators:	$\Sigma_{OP} = \{+, -, *, /, :=, \ldots \}$

In software engineering, the alphabet of a programming language Σ_{PL} is usually an extension of $\Sigma_R \cup \Sigma_D \cup \Sigma_{OP}$. The alphabets of modeling and specification notation systems can be more complicated over Σ_L with extensions of special structural, architectural, and behavioral symbols.

6.3.2 STRINGS

Strings are the second-level building block of a language based on a given alphabet.

Definition 6.13 A *string s* over an alphabet Σ is a finite sequence of symbols defined on Σ. A string is also known as a *word*.

Example 6.2 The following sequences are typical strings:

a) A string over Roman alphabet Σ_R: s_1 = ‘Software engineering’.

b) A string over the binary alphabet Σ_B: s_2 = '011'.
c) An empty string over any alphabet: $s_3 = \varnothing$, or s_3 = ' '.

Major string operations are length test, concatenation, and closure.

Definition 6.14 The *length of a string s*, L_S, is the number of symbols included in s, i.e.:

$$\begin{aligned} L_S &= |s| \\ &= \#s \end{aligned} \qquad (6.3)$$

Example 6.3 Given strings as shown in Example 6.2, the lengths L_{s1} = $|s_1|$ = #(Software engineering) = 20, note that a space between symbols is counted as a symbol in the string; $L_{s2} = |s_2|$ = #(011) = 3; and $L_{s3} = |s_3| = |\varnothing|$ = #('') = 0.

Definition 6.15 The *concatenation* ○ of two strings p and q over given alphabets Σ_p and Σ_q is the connection of p and q in the given sequence that forms a combined string s over a joint alphabet $\Sigma_p \cup \Sigma_q$, i.e.:

$$s = p \circ q, \quad s \in \Sigma_p \cup \Sigma_q \text{ and } |s| = |p| + |q| \qquad (6.4)$$

where, $s(i) = p(i)$ for $i = 1, \ldots, |p|$, and $s(|p|+i) = q(\mathrm{i})$ for $i = 1, \ldots, |q|$.

Example 6.4 Let s_1 and s_2 be the strings as given in Example 6.3. Then, a string s that concatenates s_1 and s_2 is as follows:

$$\begin{aligned} s &= s_1 \circ s_2 \\ &= \text{'Software engineering'} + \text{'011'} \\ &= \text{'Software engineering011'}, \; s \in \Sigma_R \cup \Sigma_B \end{aligned}$$

Concatenation can be extended to operations on more than two strings as described below.

Lemma 6.3 The concatenation operation of arbitrary strings p, q, and r on an alphabet Σ is constrained by the following laws:

a) Associative: $(p \circ q) \circ r = p \circ (q \circ r)$ (6.5)
b) Antisymmetric: $p \circ q \neq q \circ p$ (6.6)

Both laws on concatenation can be proven directly by using Definition 6.15.

Definition 6.16 The *closure* of an alphabet $\sum$, $\sum^*$, is the set of all strings on $\sum$, including the empty string $\varnothing$.

The number of strings that can be generated by a closure $\sum^*$, $\#(\sum^*)$, may easily grow to an infinitive when the size of the alphabet $\#\sum$ is larger enough.

Example 6.5 a) Let alphabet $\sum_1 = \{a, b\}$, the closure of $\sum_1$ is $\sum_1^* = \{\varnothing, a, b, ab, ba\}$. b) A subset of the closure of Roman alphabet $\sum_R$ that includes all strings with length of three is: $\sum_{R3}^* = \{s \in \sum_R^* \mid \#(s) = 3\} = \{abc, bcd, ..., xyz, \ldots\}$.

Example 6.6 The following strings belong to a specific closure of alphabet:

(a) s_1 = 'Software engineering' $\in \sum^*_R$
(b) s_2 = '011' $\in \sum^*_B$
(c) $s = s_1 \circ s_2$ = 'Software engineering011' $\in (\sum_R \cup \sum_B)^*$

6.3.3 EXPRESSIONS

Expressions are the third-level building block of a language based on a given alphabet.

Definition 6.17 An *expression e* is a string on an alphabet or a number of strings concatenated by a set of special symbols known as *operators*.

Definition 6.18 A *regular expression* e_r is an expression or a special kind of strings that consists of single symbols on a given alphabet, or of those single symbols combined with the symbols of the empty string $\varnothing$, union $\cup$, repeat *, and parentheses ( ).

Example 6.7 According to Definition 6.18, the following expression

$$e_{r1} = (a \cup b)^* \, a$$

is a regular expression.

6.3.4 GRAMMAR THEORIES

As described in Section 6.2.4, the concept of formal grammars was developed by linguist Noam Chomsky in the 1950s [Chomsky, 1956]. In this subsection, readers are prepared with introductions to some terms before the discussions on formal grammars are presented.

6.3.4.1 Production Rules of Grammars

A production rule is a function that facilities formal reasoning in grammar analyses. On the basis of a set of productions, a grammar can be formally defined.

Definition 6.19 A *terminal* is a constant string on a given alphabet Σ that has a defined semantics and cannot be broken down further. Terminals will be denoted by lower case italic letters.

Definition 6.20 A *nonterminal* is a variable string on a given alphabet Σ that is a syntactical category or combined term with semantics dependent on further deductions. Nonterminals will be denoted by italic capital letters.

Definition 6.21 A *production p* is a function that produces an ordered pair (α, β), i.e.:

$$p\text{: } \alpha \rightarrow \beta \tag{6.7}$$

where α and β is a terminal, nonterminal, or their combinations.

Example 6.8 The following are productions:

$$A \rightarrow a, A \rightarrow Aa, A \rightarrow B, \text{ and } aB \rightarrow b$$

These productions may be shortly denoted by:

$$A \rightarrow (a, Aa, B), \text{ and } aB \rightarrow b$$

A production with all terminals on its right-hand side (RHS) is a *final product* with a derived semantics or physical meaning; while a product with at least one nonterminal on its RHS is an *intermediate product* with its semantics pending on further deduction.

6.3.4.2 Taxonomy of Grammars

There are various grammars classified according to the types of productions adopted for establishing the rules of a grammar, the relations of a grammar with the contexts, or the techniques for grammar recognitions.

6.3.4.2.1 Chomsky Grammars

According to Noam Chomsky [Chomsky, 1956], formal grammars can be classified as Type 0 through Type 3 from the bottom up with increasing rigor, based on the types of production rules adopted in the grammars.

Definition 6.22 A *Type 0 grammar, G_0*, is a grammar that has no restrictions on its productions.

Definition 6.23 A *Type 1 grammar, G_1*, is a grammar that satisfies the following conditions:

$$\forall p \in G_1,\ p\colon \alpha \rightarrow \varnothing \vee (p\colon \alpha \rightarrow \beta \Rightarrow |\alpha| \leq |\beta|) \tag{6.8}$$

Definition 6.24 A *Type 2 grammar, G_2*, is a grammar that satisfies the following condition:

$$\forall p \in G_2,\ p\colon A \rightarrow \beta \tag{6.9}$$

where A is a nonterminal.

Definition 6.25 A *Type 3 grammar, G_3*, is a grammar that satisfies the following conditions:

$$\forall p \in G_3,\ p\colon s_0 \rightarrow \varnothing \vee p\colon A \rightarrow a \vee p\colon A \rightarrow aB \tag{6.10}$$

where s_0 is the start symbol, A and B are nonterminals, and a is a single terminal.

Corollary 6.1 The four-type *Chomsky grammars*, G_0 through G_3, satisfy the following relations:

$$G_3 \subseteq G_2 \subseteq G_1 \subseteq G_0 \tag{6.11}$$

That is, a higher level grammar imposes stronger restrictions on its production rules than those of the lower level grammar(s).

Corollary 6.1 can be directly proven on the basis of Definitions 6.22 through 6.25.

6.3.4.2.2 Grammars Classified by Relations with the Contexts

A context of a production is a certain configuration of all symbols in the strings and expressions of a production.

Definition 6.26 A *context-sensitive grammar* G_s is a grammar that is constrained by the following condition:

$$\forall p \in G_s, p: \alpha A \alpha' \rightarrow \alpha\beta\alpha' \qquad (6.12)$$

where $\alpha A \alpha'$ is the *context*, and A is a nonterminal symbol that can be replaced in the given context.

Definition 6.27 A *context-free grammar* G_f is a grammar that is constrained by the following condition:

$$\forall p \in G_f,\ p: A \rightarrow \beta \qquad (6.13)$$

where p is context-independent.

Definition 6.28 A *regular grammar* G_r is a grammar that is constrained by the following conditions:

$$\forall p \in G_r,\ p: s_0 \rightarrow \varnothing \vee p: A \rightarrow a \vee p: A \rightarrow aB \qquad (6.14)$$

Corollary 6.2 The three types of grammars classified with regard to their contexts, G_s, G_f, and G_r, satisfy the following relations:

$$G_r \subseteq G_f \subseteq G_s \qquad (6.15)$$

and

$$G_s = G_1 \qquad (6.16a)$$
$$G_f = G_2 \qquad (6.16b)$$
$$G_r = G_3 \qquad (6.16c)$$

6.3.4.2.3 Formal Description of Context-Free Grammars

Context-free grammars, or Type 2 grammars due to Chomsky, are a category of the most widely used grammars in language generation, specification, recognition, and processing.

Definition 6.29 A *context-free grammar* G_f is a language generator that can be described by a 4-tuple:

$$G_f \triangleq (\Sigma, s_0, T, R) \qquad (6.17)$$

where

(i) Σ is a finite nonempty set of alphabet;
(ii) T the set of terminals, $T \subseteq \Sigma$;
(iii) s_0 the start symbol, $S_0 \in (\Sigma \setminus T)$, which is a nonterminal;
(iv) R the set of rules, $R \subseteq (\Sigma \setminus T) \times \Sigma^*$, which is called productions.

Example 6.9 According to Definition 6.27, the following given grammar $G_{f1} = (\Sigma, s_0, T, R)$ is a context-free grammar where:

$$\begin{aligned} &\Sigma = \{a, b, \varnothing\} \\ &s_0 = a \\ &T = \{b\} \\ &R = \{s_1 \to aMb, M \to A, M \to B, A \to \varnothing, A \to aA, \\ &\qquad B \to \varnothing, B \to bB\} \end{aligned}$$

In the rule set R, a capital letter represents a nonterminal symbol or a variable. Usually, a grammar G may be simply denoted by its productions R, because R contains all necessary information for defining G.

6.3.4.2.4 Grammars Classified by Language Recognition Techniques

According to the relations to language parsing techniques, context-free grammars can be classified into two categories known as the *LL*(*k*) [Aho and Ullman, 1972] and *LR*(*k*) grammars [Knuth, 1965].

Definition 6.30 An *LL*(*k*) *grammar* is a class of context-free grammars, where the first *L* denotes that the parsing is from *l*eft to right, the second *L* specifies that the next production is derived by *l*eft-most derivation, and *k*, $k \geq 1$, denotes that at most *k*-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.

When k > 1, the *LL*(*k*) grammar is called a strong *LL*(*k*) grammar. Any *LL*(*k*) grammar is unambiguous and suitable for top-down parsing. *LL*(*1*) grammars are widely used in compiling systems because most of the syntaxes of high-level programming languages can be defined by an *LL*(*1*) grammar [Aho, Sethi, and Ullman, 1985].

Definition 6.31 An *LR*(*k*) *grammar* is a class of context-free grammars, where the letter *L* denotes that the parsing is from *l*eft to right, the letter *R* specifies that the next production is derived by *r*ight-most derivation in reverse, and k, $k \geq 1$, denotes that at most k-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.

LR(k) grammars are the most widely used techniques in compiling systems because they are suitable for bottom-up parsing in order to reduce shift in input scanning [Knuth, 1965]. Any deterministic language can be defined by an *LR*(*1*) grammar [Hopcroft and Ullman, 1979]. The relationship between the two parsing-driven grammars is that any *LL*(k) grammar is necessarily *LR*(k) [Knuth, 1965].

6.3.5 LANGUAGES

A language is a set of expressions and strings over an alphabet that is formed following certain properties and rules known as the grammar.

Definition 6.32 A *language* on a given alphabet Σ is a subset of expressions e over Σ^*, i.e.:

$$L = \{e \in \Sigma^* \mid p(e)\} \tag{6.18}$$

where $p(e)$ denotes each e of L possesses the common property or satisfies the grammar rule p, i.e., $\forall e \in L \Leftrightarrow p(e)$.

According to Definition 6.32, the following strings and expressions belong to a language:

- The empty string $\varnothing$
- All individual symbols in Σ
- All finite sequences of combinations of the individual symbols on Σ

Example 6.10 a) Let $\Sigma_1 = \{a, b\}$, then language $L_1 = \{s \in \Sigma_1^*\} = \{\varnothing, a, b, ab, ba\}$. b) Let $\Sigma_2 = \Sigma_\mathrm{R}$, then language $L_2 = \{s \in \Sigma_{\mathrm{R}3}^* \mid \#(s) = 3\} = \{abc, bcd, ..., xyz, \ldots\}$.

In order to enable machines to process certain language in computing, regular languages are introduced that composite more restricted strings in the form of regular expressions as described in Section 6.3.3.

Definition 6.33 A *regular language* L_r over an alphabet Σ is a set of regular expressions on Σ^*, i.e.:

$$L_r = \{e_r \in \Sigma^* \mid p(e_r)\} \qquad (6.19)$$

Regular languages are all languages that can be described by regular expressions. In other words, every regular expression represents a regular language.

Example 6.11 According to Definition 6.8, the regular expression $e_{r1} = (a \cup b)^*\ a$ defines a regular language L_{r1} as follows:

$$L_{r1} = \{e_{r1} \in \{a, b\}^* \mid e_{r1} = (\mathrm{a} \cup \mathrm{b})^*\ \mathrm{a}\}$$

which encompasses all strings determined by e_{r1} in the form $\{a, b\}^*\ a$, representing any string repeated by *ab* and ends with *a*.

The relationship between formal languages and machines can be described by the following corollary.

Corollary 6.3 A *language* is *regular iff* it is accepted by a finite automaton.

Kleene proved the above theorem [Kleene, 1956], which reveals that automata are machines designed for recognizing and executing given instructions in the form of regular expressions or restricted strings with a set of given rules, i.e., the grammar.

Definition 6.34 A *context-free language L_f* is a language generated by a context-free grammar G_f, i.e.:

$$L_f = L(G_f) \qquad (6.20)$$

Example 6.12 According to Definition 6.34, for a given context-free grammar $G_{f1} = (\Sigma, s_0, T, R)$ as shown in Example 6.9, the language

$$L(G_{f1}) = a\ (a^* \cup b^*\ )\ b$$

is a context-free language generated by G_{f1}.

In computing, context-free grammars are used for language *generators*, while automata are used for language *recognizers*. Context-free grammars are extremely useful in modeling syntaxes of programming languages, and parsers of compilers for programming languages. Techniques on using $LL(k)$

and EBNF for RTPA recognition will be presented in Section 6.4.4 [Tan and Wang, 2006; Tan, Wang, and Ngolah, 2004a/04b/05/06; Ngolah, Wang, and Tan, 2005b/06].

6.3.6 BNF AND EBNF

A Backus-Naur form is a recursive notation for describing the productions of a context-free grammar. It is developed based on the work of John Backus with contributions by Peter Naur [Naur, 1963/78].

Definition 6.35 A *Backus-Naur Form* (BNF) is defined by a 5-tuple:

$$BNF \triangleq (\Sigma, T, V, P, S) \tag{6.21}$$

where

(i) Σ is a finite nonempty set of alphabet;

(ii) T is a finite set of terminals, $T \subseteq \Sigma$;

(iii) V is a finite set of nonterminals, $V \subseteq \Sigma \wedge V = \Sigma \setminus T$;

(iv) P is a finite set of production rules denoted by $\alpha ::= \beta$.

(v) S is a finite set of metasymbols that denote relations of the multiple derived products βs separated by alternative selection |.

Example 6.13 The BNF counterparts of the productions as shown in Example 6.8, $A \to (a\ , Aa, B)$, $aB \to b$, can be recursively denoted by:

$$A ::= a \mid Aa \mid B$$
$$aB ::= b$$

Definition 6.36 An *abstract syntax* is the abstract description of a syntax where strings of tokens or nodes in a parse tree are represented by a symbol, usually a single letter.

Example 6.14 Assume the following letters be used to represent their corresponding concrete syntactic entities: P <program>, L <statement list>, S <statement>, E <expression>, I <identifier>, A <letter>, N <number>, and D <digit>. The abstract syntax of a *Sample Programming Language* (SPL) [Louden, 1993] for integer arithmetic expressions, variables, basic statements, assignments, and loop constructs can be described in BNF as shown in Fig. 6.6.

```
P ::= L
L ::=  L1 ‘;’ L2
     | S
S ::=  I ‘:=’ E ‘;’
     | ‘if’ E ‘then’ L1 ‘;’
     | ‘if’ E ‘then’ L1 ‘else’ L2 ‘;’
     | ‘while’ E ‘do’ L ‘;’
E ::=  E1 ‘+’ E2
     | E1 ‘-‘ E2
     | E1 ‘*’ E2
     | ‘(‘ E ‘)’
     | I
     | N
I ::= I A | A
A ::= ‘a’ | ‘b’ | … | ‘z’
N ::= N D | D
D ::= ‘0’ | ‘1’ | … | ‘9’
```

Figure 6.6 An abstract syntax representation of the sample language SPL

BNF is found very useful to define context-free grammars of programming languages, because its simplicity of notations, highly recursive structures, and the support of many compiler generation tools, such as YACC [Johnson, 1975], LEX [Lesk, 1975], and ANTLR [Parr, 2000].

In applications it is realized the descriptive power of BNF may be greatly improved by introducing a few extended metasymbols, particularly the ones for *repetitive* and *optional* structures of grammar rules. There are a variety of extended BNFs proposed for grammar description and analysis [Wirth, 1976]. A typical EBNF is given below.

Definition 6.37 An *extended Backus-Naur form* (EBNF*)* is defined by a similar 5-tuple as given in Eq. 6.21, i.e.:

$$EBNF \triangleq (\Sigma, T, V, P, S') \qquad (6.22)$$

with an extended set of metasymbols $S' = \{\,|, (\,)^*, (\,)^+, [\,]\}$, where:

(i) The metasymbol β^* and β^+ are adopted to denote the repetitive structures of derived products, where $\beta^* = \mathop{R}\limits_{i=0}^{n} \beta_i$ and $\beta^+ = \mathop{R}\limits_{i=1}^{n} \beta_i$ according to the *big-R* notation [Wang, 2002a].

(ii) The metasymbol $[\beta]$ is adopted to denote optional structures of a derived product.

Example 6.15 The BNF representation of the abstract syntax of SPL as given in Example 6.14 can be simplified by using EBNF nations as shown in Fig.6.7. The improved descriptions of grammar rules in EBNF are highlighted by underlines.

P ::= L
L ::= L_1 ‘;’ L_2
| S
S ::= I ‘:=’ E ‘;’
| ‘if’ E ‘then’ L_1 [‘else’ L_2] ‘;’
| ‘while’ E ‘do’ L ‘;’
E ::= E_1 ‘+’ E_2
| E_1 ‘-‘ E_2
| E_1 ‘*’ E_2
| ‘(‘ E_1 ‘)’
| I
| N
I ::= A A*
A ::= ‘a’ | ‘b’ | … | ‘z’
N ::= D D*
D ::= ‘0’ | ‘1’ | … | ‘9’

Figure 6.7 An abstract syntax description of SPL using NBNF

6.4 Syntaxes of Programming Languages

In Section 6.3 it is demonstrated that a programming language can be designed and generated from the bottom up according to a set of predefined lexes and syntaxes. Reversely, the language can be recognized, analyzed, and reduced from the top-down via lexical and syntactic analyses. Therefore, syntactical theories of programming languages play an important role in language processing.

In software engineering it is more interested in language recognition, cognition, and its expressive power. A *programming language* in software engineering can be perceived as a special notation system for describing and specifying instructive computing information on both architectural (data) and behavioral (process) aspects of software systems.

The processing of programming languages by a compiler, interpreter, or generally a translator can be carried out in four phases: a) lexical analysis, b) syntactical analysis, c) semantic analysis, and d) code generation, as shown in Fig. 6.8.

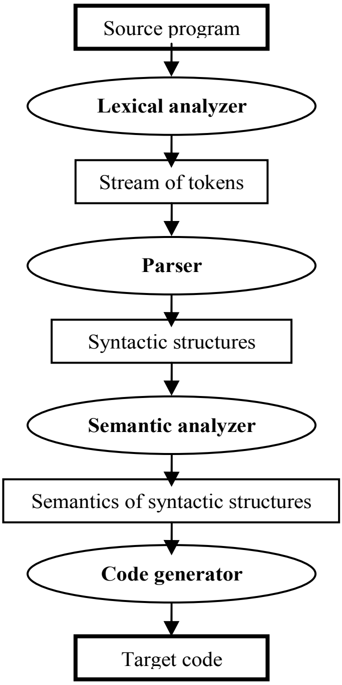


Figure 6.8 Processes of programming language compilations

This section describes the analyses of program structures in a programming language in the first two phases known as lexical and syntactical analysis. The third and fourth phases on semantic analysis and code generation will be discussed in Section 6.5. Further discussions on intelligent code generation will be presented in Section 15.4.2.

6.4.1 LEXICAL ANALYSES

A lexeme is a basic lexical unit of a language, such as a word or a phrase, where the elements of which do not separately convey the meaning of the whole. Lexemes and lexical structures are the object of study in morphology.

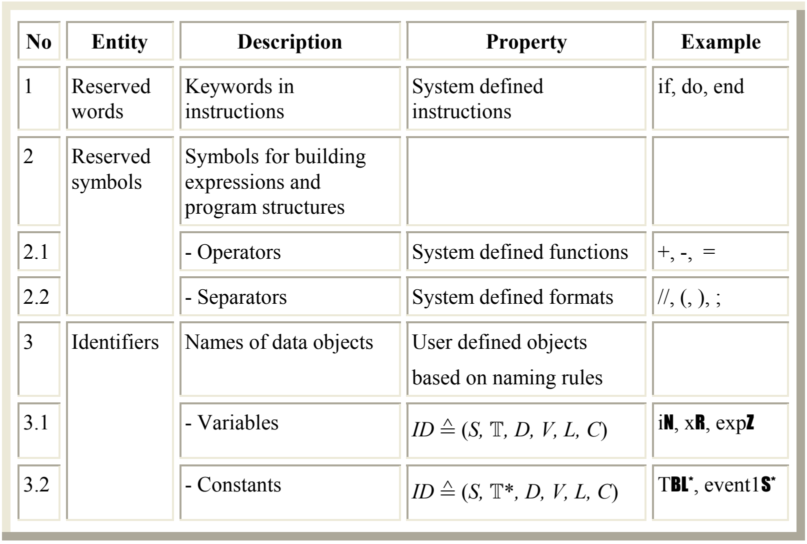
Definition 6.38 The *lexical structure* of a programming language is the structures of its lexemes, such as strings or words, known as *tokens* in language processing.

6.4.1.1 Taxonomy of Lexical Entities in Programming Languages

Tokens of a programming language can be classified into three categories that represent program entities of *reserved words, reserved symbols* (operators and separators), and *identifiers* (user-defined variables and constants) as shown in Table 6.4.

Identifiers are the most widely used entities in programming for representing variables, constants, procedures, classes, and program names. According to Definition 5.23, an identifier *ID* is a logical name of a language entity or construct, which can be essentially and uniquely specified by a 6-tuple $ID \triangleq (S, \mathbb{T}, D, V, L, C)$, where the most important properties are the representative symbol S and its type $\mathbb{T}$. The other properties in the 6-tuple are the *ID*'s domain D, instant values V, the physical location L, and the scope C. Detailed descriptions can be referred to Section 5.3.2 on basic data object modeling techniques in computing. For the definition of a constant ID in Table 6.4, $\mathbb{T}^*$ represents the type of the constant as that of variables.

Table 6.4
Taxonomy of Lexical Entities



No	Entity	Description	Property	Example
1	Reserved words	Keywords in instructions	System defined instructions	if, do, end
2	Reserved symbols	Symbols for building expressions and program structures		
2.1		- Operators	System defined functions	+, -, =
2.2		- Separators	System defined formats	//, (,), ;
3	Identifiers	Names of data objects	User defined objects based on naming rules	
3.1		- Variables	$ID \triangleq (S, \mathbb{T}, D, V, L, C)$	i**N**, x**R**, exp**Z**
3.2		- Constants	$ID \triangleq (S, \mathbb{T}^*, D, V, L, C)$	T**BL***, event1**S***

6.4.1.2 Lexical Analyses of Programs

In programming language processing, lexical analyses are conducted by the *lexical analyzer* or the *scanner*. The input of the lexical analyzer is the *source program* in a given language, and the output of the lexical analyzer is a sequence of tokens identified from the program.

Lexical analysis is aimed to chop a long list of the source code in a programming language into a finite sequence of individual tokens, for each of them, its language properties such as identifier, reserved word, or reserved symbol, as classified in Table 6.4, is clearly identified.

6.4.2 SYNTAX DEFINITIONS AND DESCRIPTIONS

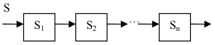
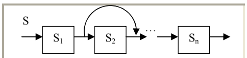
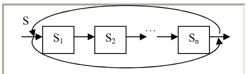
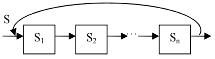
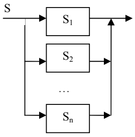
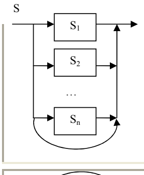
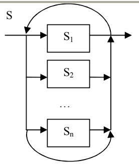
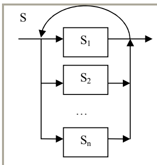
The syntax of a programming language constitutes how the language is built with lexical symbols or tokens.

Definition 6.39 The *syntax* of a programming language is a set of grammatical rules for constructing legal instructions.

The grammar rules of a given language that constrain and direct a syntactic analysis of a parser can be described by BNF or EBNF as discussed in Section 6.3.6. Although both descriptions of grammars in BNF and EBNF are equivalent, the EBNF representation is more efficient and expressive to facilitate syntactic analyses. The BNF or EBNF grammar rules are applied by the parser to determine whether an inputted sequence of tokens is legal and correct. Corresponding to the EBNF description of the syntactic structures of a programming language, a flow diagram known as *syntax diagram* can be used to illustrate the rules and syntactic structures. Syntax of programming languages can also be formally described by RTPA.

Table 6.5 contrasts the three syntactical description techniques for typical syntactic entities and structures in EBNF, syntax diagrams, and RTPA. In the syntax diagrams, a terminal and a nonterminal are represented by an oval and a square, respectively.

Table 6.5
Description of Typical Syntactic Entities and Structures

No	Syntactic structure	EBNF notation	Syntax diagram	RTPA notation
1	Serial	S ::= $S_1\ S_2 \ldots S_n$		$S = S_1 \to S_2 \to .. \to S_n$
2	Serial with option	S ::= $S_1\ [S_2] \ldots S_n$		$S = S_1 \to S_2 \to S_3 \ldots \to S_n$ $\mid S_1 \to S_3 \to \ldots S_n$
3	Repeat serial for 0 or more times	S ::= $(S_1\ S_2 \ldots S_n)^*$		$R^{*}(s_1 \to s_2 \to \ldots \to s_n)$
4	Repeat serial for 1 or more times	S ::= $(S_1\ S_2 \ldots S_n)^+$		$R^{+}(s_1 \to s_2 \to \ldots \to s_n)$
5	Alternative	S ::= $S_1 \mid S_2 \mid \ldots \mid S_n$		$S = S_1 \mid S_2 \mid \ldots \mid S_n$
6	Alternative with option	S ::= $[S_1 \mid S_2 \mid \ldots \mid S_n]$		$S = S_1 \mid S_2 \mid \ldots \mid S_n \mid \varnothing$
7	Repeat alternative for 0 or more times	S ::= $(S_1 \mid S_2 \mid \ldots \mid S_n)^*$		$R^{*}(s_1 \mid s_2 \mid \ldots \mid s_n)$
8	Repeat alternative for 1 or more times	S ::= $(S_1 \mid S_2 \mid \ldots \mid S_n)^+$		$R^{+}(s_1 \mid s_2 \mid \ldots \mid s_n)$

Example 6.16 The syntax diagram of expressions in SPL as described in Section 6.3.6, i.e.:

$$\begin{aligned} E ::= \; & E_1 \text{ ‘+’ } E_2 \\ | \; & E_1 \text{ ‘-‘ } E_2 \\ | \; & E_1 \text{ ‘*’ } E_2 \\ | \; & \text{‘(‘ } E \text{ ‘)’} \end{aligned}$$

can be derived as shown in Fig. 6.9.

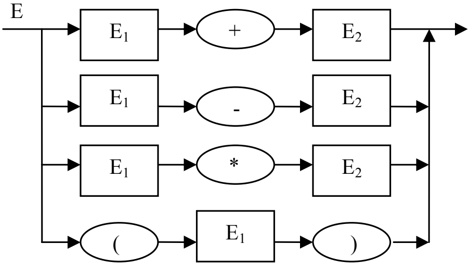


Figure 6.9 The syntactic structure of expressions in SPL

6.4.3 SYNTACTICAL ANALYSES

In programming language processing, syntactical analyses are conducted by a parser or a syntactical analyzer. The input of the parser is a sequence of tokens identified by the lexical analyzer; and the output of the parser is a set of syntactically structured program instructions.

6.4.3.1 Basic Syntactical Analysis Techniques

Fundamental syntax analysis techniques can be classified into top-down and bottom-up parsing approaches, which adopt the *LL*(*k*) and *LR*(*k*) grammars, respectively.

6.4.3.1.1 Top-Down Parsing

Definition 6.40 *Top-down parsing* is a class of parsing techniques that matches an input string to a given syntax tree in a preorder, i.e., from the root of the syntax tree to the leftmost nodes.

Top-down parsing is directed by an *LL*(*k*) grammar. Frequently used top-down parsing techniques are recursive-descent parsing and predictive parsing. $LL(k)$ parsers can be implemented using the compiler generation tool ANTLR [Parr, 2000].

Definition 6.41 *Recursive-descent* parsing is a top-down parsing technique that derives a parsing tree according to a set of left-recursive grammar rules.

For example, a recursive-descent parser can process a grammar $G_1 = \{S ::= \alpha A\beta, A ::= a \mid b\}$, where α and β are strings of tokens with terminals and/or nonterminals. It may require backtracking when a derivation can not match the whole input string in case the input string $s = \alpha a\beta$, for a grammar $G_2 = \{S ::= \alpha A\beta, A ::= ab \mid a\}$. It may also result in an infinitive loop in the case such as $G_3 = \{S ::= \alpha A\beta, A ::= ab \mid a \mid A\}$.

A parser is nondeterministic if there is at least one decision point where the parser cannot resolve which path to take. Nondeterminisms arise because of the weakness of a given grammar.

Example 6.17 The following grammar rule parsing by $LL(1)$ grammar is nondeterministic:

$$S ::= \text{aa} ; \mid \text{a} ;$$

However, by using $LL(2)$ grammar, the above rule is deterministic because the second lookahead token helps to uniquely determine which alternative to predict.

Definition 6.42 *Predictive parsing* is a restricted form of recursive-descent parsing where the backtracking is eliminated in the top-down parsing by adopting an $LL(1)$ grammar.

Any production rule in $LL(1)$ in the form of $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ must meet the following conditions: a) The first token in $\alpha_i \mid$, $1 \leq i \leq n$, should be unique; and b) The following tokens in each α_i should not be the same with any of the first tokens of α_i.

6.4.3.1.2 Bottom-Up Parsing

Complemented to the top-down approach, a bottom-up approach to syntactic analysis and parser implementation is also widely used.

Definition 6.43 *Bottom-up parsing* is a class of parsing techniques that derives a parse tree for an input string from the leaves to the root, in order to reduce the string to the start symbol of production rules.

Example 6.18 Assume the grammar rules for expressions E are given below:

$$E ::= E + E \mid E - E \mid E * E \mid (E) \mid id$$

A bottom-up rightmost derivation for the input string $S = id_1 + id_2 * id_3$ can be reduced in the following steps:

$$\begin{aligned} S &= id_1 + id_2 * id_3 \\ &\Rightarrow E + id_2 * id_3 \\ &\Rightarrow E + E * id_3 \\ &\Rightarrow E + E * E \\ &\Rightarrow E + E \\ &\Rightarrow E \end{aligned}$$

Bottom-up parsing is usually directed by an *LR*(*k*) grammar. That is, at each step of reduction, a rightmost derivation is traced out. *LR*(*k*) parsers can be implemented using the compiler generation tool YACC (Yet Another Compiler-Compiler) [Johnson, 1975].

6.4.3.2 Description of Parsing Results by Syntax Trees

Definition 6.44 An *Abstract Syntax Tree* (AST) is a tree structure that represents the parsing result, including a number of tokens and their syntactical relations, in a hierarchical diagram.

Example 6.19 An AST of a simple grammar rule and an AST of the RTPA top-level rule of specifications can be generated by a parser, respectively, as follows:

a) S ::= a | b

```
 S
/ \
a b
```

b) §(SysID**S**) ::= Architecture**ST**
|| StaticBehaviors**ST**
|| DynamicBehaviors**ST**

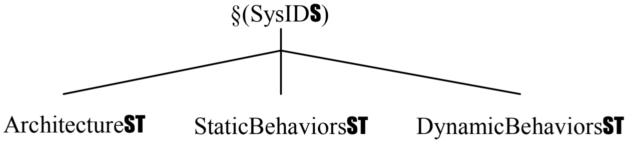


Figure 6.10 ASTs generated by parsers

6.4.4 SYNTACTICAL ANALYSES OF RTPA

The syntax of RTPA has been described in Section 4.6. This subsection describes the syntax analysis techniques for RTPA and their implementation [Tan and Wang, 2006; Tan, Wang, and Ngolah, 2004a/05/06; Ngolah, Wang, and Tan, 2005b/06]. A comprehensive set of RTPA grammar rules can be referred to [Tan, Wang, and Ngolah, 2004b].

6.4.4.1 Description of the RTPA Syntax in LL(k)

The RTPA methodology and syntax for software system specification and modeling are specified by a set of almost 300 *LL*(*k*) grammar rules [Tan, Wang, and Ngolah, 2004b]. Since *LL*(*k*) is a context-free grammar, it can be described by EBNF.

On the basis of the description of the RTPA methodology and syntax in Sections 4.6 and 4.7, the top-level RTPA grammar rule for the entire architecture of a given software system can be specified in EBNF as shown in the following rule.

```
rtpa_system_specification ::=
      top_schema architecture static_behaviors dynamic_behaviors
      EOF                                                      (6.23)
```

The first part of the system specification is the *top_schema*, which can be refined in Eq. 6.24. In this rule, the system declaration specifies the name of the system by a definition symbol. The system architecture declaration, system static behaviors declaration, and system dynamic behaviors declaration define the three subsystems in an RTPA specification, respectively.

```
top_schema ::=
      system_declaration DEFINITION_SYMBOL
      system_architecture_declaration
      PARALLEL_SYMBOL
      system_static_behaviors_declaration
      PARALLEL_SYMBOL
      system_dynamic_behaviors_declaration                     (6.24)
```

The system name is represented by a variable of kind *system-name* with type **S** and scope *global*. The system name can be used as a prefix before subsystem names, CLM names, process names, events, statuses, and constants used in the system. The system name variable should not be assigned with any value in the system specification, while it is treated as a unique identification of the system when it is referred in other system specifications in the way that it is initialized to a variable of *system architecture type* **ST**.

The *LL*(*k*) grammar rules of RTPA can also be described by a corresponding syntax diagram.

Example 6.20 An EBNF grammar rule for the component logical model (CLM) of RTPA as given in Eq. 6.25 below can be described by a corresponding syntax diagram in Fig. 6.11. More formal description of CLMs in RTPA has been provided in Section 5.3.3.3.1.

```
clm_schemas ::= ( clm_schema
                  (EQUALITY_SYMBOL
                    clm_object (PARALLEL_SYMBOL
                                 clm_object)*
                  )
                )+                                    (6.25)
```

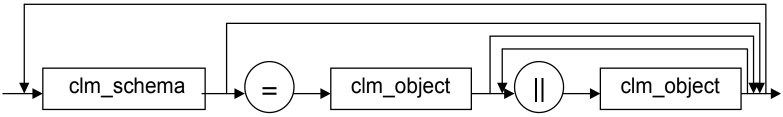


Figure 6.11 An EBNF syntax diagram of RTPA CLM schemas

6.4.4.2 Description of Special RTPA Grammar Rules by Syntactic Predicates

There are special RTPA grammar rules that cannot be described by *LL*(*k*) grammar, such as assignment, read/write, I/O, timing, duration, and the iteration processes denoted in the big-R notation. These special parsing rules of RTPA can be specified by a special means known as the syntactic predicates.

Definition 6.45 A *syntactic predicate*, denoted by:

$$<\textit{syntactic entity}> \Rightarrow <\text{production}> \qquad (6.26)$$

is a selective form of backtracking adopted for recognizing complicated language constructs that cannot be distinguished without seeing the entire structure.

A syntactic predicate is a conditional production determined by a current scanned token that directs the selection of a suitable rule in parsing.

The syntactic predicates can be implemented by the guarded predicate provided by ANTLR (ANother Tool for Language Recognizer) [Parr, 2000] in the process of RTPA parser generation [Tan, Wang, and Ngolah, 2004a/06].

Example 6.21 The grammar rules of several RTPA meta processes, such as shown in Eq. 6.27 through Eq. 6.31, may not be determined by a fixed *k* according to *LL*(*k*) grammar, because these meta processes share the same identifier with arbitrary length in an RTPA specification.

```
single_assignment ::=
   variable ASSIGNMENT_SYMBOL expression                (6.27)

read_process ::=
   memory_expression READ_SYMBOL variable               (6.28)

write_process ::=
   variable WRITE_SYMBOL memory_expression              (6.29)

input_process ::=
   port_expression INPUT_SYMBOL variable                (6.30)

output_process ::=
   variable OUTPUT_SYMBOL port_expression               (6.31)
```

To solve the above problems, a new rule using syntactic predicates as guarded directors is introduced as shown below.

```
id_prefixed_process ::=
     (pointer_variable ASSIGNMENT_SYMBOL) =>
          addressing_process
   | (variable ASSIGNMENT_SYMBOL) => single_assignment
   | (variable OUTPUT_SYMBOL) => output_process
   | (variable READ_SYMBOL) => read_process
   | (variable WRITE_SYMBOL) => write_process
   | (prefixed_identifier [subscript_expression]
          LEFT_PARENTHESIS) => process_instance_expression
   | (prefixed_identifier [subscript_expression]) => name_process
                                                           (6.32)
```

The rule given in Eq. 6.32 specifies that alternative guarding conditions should be checked before a specific alternative rule may be selected. The conditional syntactic predicates are shown in the left-hand side of "=>" in Eq. 6.32. Since the conditions are described uniquely for each rule in Eq. 6.32, a determinable choice can be implemented in the RTPA parser. The method for solving the nondeterministic problems in parsing an RTPA specification is also useful in the definition of the expression rules of RTPA.

6.4.4.3 Parsing RTPA Specifications

An advanced compiler generation tool, ANTLR [Parr, 2000], takes *LL*(*k*) grammar rules as its input and generates a corresponding *LL*(*k*) parser as the output. ANTLR generates *predicate*-*LL*(*k*) parsers that support syntactic predicates for specifying both context-free and context-sensitive grammars. The generated RTPA parser encompasses a lexical analyzer and a type checker. The parsing of an RTPA system model and system specification can be conducted as shown in Fig. 6.12.

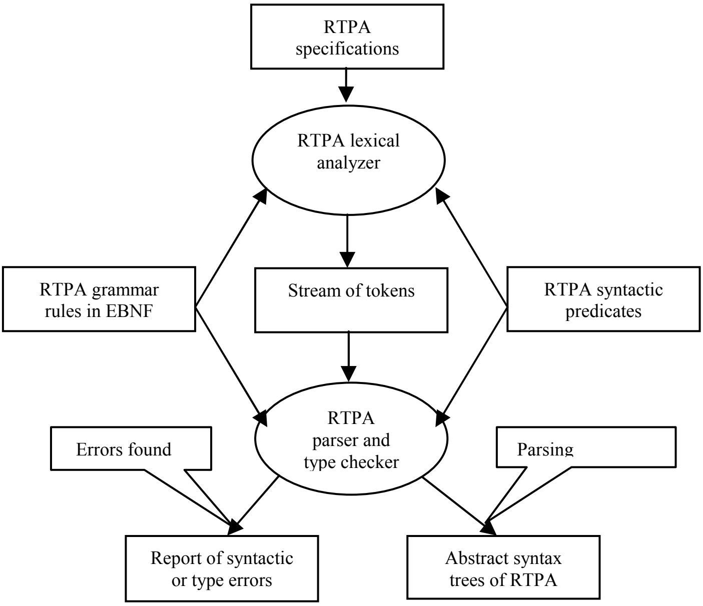

RTPA specifications
RTPA lexical analyzer
RTPA grammar rules in EBNF
Stream of tokens
RTPA syntactic predicates
RTPA parser and type checker
Errors found
Parsing
Report of syntactic or type errors
Abstract syntax trees of RTPA

Figure 6.12 Syntactic analysis of RTPA specifications

Under the support of ANTLR, the RTPA parser can be implemented in the following processes [Tan, Wang and Ngolah, 2004b/2006]:

a) To define the RTPA grammar in EBNF.

b) To convert the RTPA grammar into a set of *LL*(*k*) parsing rules. For those RTPA grammar rules that can not be described by *LL*(*k*), the ANTLR syntactic predicates are used to make them determinable within a fixed depth of look-ahead.

c) To embed a type checker into the rules of RTPA parser as special semantic actions.

d) To load the parser rules of RTPA to ANTLR in order to generate the RTPA parser (including the lexer and the type checker) in executable Java classes.

It is noteworthy that, although there is no clearly drawn boundary between syntactical and semantic analyses, it is generally classified that language entities and properties expressible by context-free grammars are syntactic issues; otherwise, they are semantic issues.

6.5 Semantics of Programming Languages

Studies on software semantics have been recognized as one of the key areas in the development of fundamental theories for computer science and software engineering [Hoare, 1969; Gries, 1981; McDermid, 1991; Slonneg and Kurts, 1995; Bjoner, 2000]. The semantics of a programming language is the behavioral meanings that constitute what a syntactically correct instructional statement in the language is supposed to do during run-time. The development of formal semantic theories of programming is one of the pinnacles of computing and software engineering [Pagan, 1981; Meyer, 1990; Gunter, 1992; Louden, 1993; Bjoner, 2000].

In semantics analyses, the instructions shared by all programming languages can be classified into three types: a) *Internal operations* such as memory manipulation and assignment instructions; b) *Basic control structures* such as the if-then-else and while-do instructions; and c) *External operations* onto the *environment*, such as input/output, event handling, and human-machine interactive instructions.

Definition 6.46 The *semantics* of a program in a given programming language is the logical consequences of an execution of the program that results in the changes of values of a finite set of variables and/or the embodiment of computing behaviors in the underlying computing environment.

This section introduces existing semantic theories of programming languages such as target semantics, operational semantics, denotational semantics, axiomatic semantics, and algebraic semantics. Then, it demonstrates how the semantics of a program in a given programming language is expressed and embodied on the basis of syntactic analysis. It also shows that the existing semantic notations and methodologies are inadequate to express some important instructions, complex control structures, and the real-time environments at run-time. This leads to the development of the deductive semantics as described in Section 6.5.3.

6.5.1 TAXONOMY OF SEMANTICS

Basic semantics of a programming language can be described by its *behavioral equivalence* to another language, such as a natural language or languages of the target machines. Semantics can also be described by a set of predefined executable functions in machine languages. Another approach to specify the semantics of a programming language is by mathematical definitions known as formal semantics.

A number of formal semantics, such as the *operational* [Wegner, 1972; Ollongren, 1974; Marcotty and Ledgard, 1986; Wikstrom, 1987], *denotational* [Scott and Strachey, 1971; Jones, 1980; Scott, 1982; Bjorner and Jones, 1982; Schmidt, 1988/94/96], *axiomatic* [Hoare, 1969; Dijktra, 1975/76; Gries, 1981], *algebraic* [Goguen et al., 1977/96; Guttag and Horning, 1978], and *deductive semantics* [Wang, 2006a], have been proposed in the last three decades for defining and interpreting the meanings of programs and programming languages. The following subsections describe the formal approaches to specification and analysis of program semantics.

6.5.1.1 Target Semantics

The most basic and simplest semantics of programming languages is the target semantics, which maps the equivalent behaviors of a given statement in the target-machine's language. Target semantics is typical semantics adopted in early stages of programming technologies.

Definition 6.47 *Target semantics* is an equivalent semantics that adopts a target-machine's language to interpret the behavioral meaning of a program in a programming language.

The most typical target language is assembly language. Because machine languages are system dependent, there are two drawbacks in using target semantics. The first drawback is that it is not rigorous and cannot be formally defined and described in order to facilitate machine-based semantic analyses. Another is the low efficiency in applications because the semantics of a given programming language has to be mapped into multiple target languages. Theses reasons motivated the studies on theories of formal semantics of programming languages and software systems.

A common approach towards the establishment of formal semantics is to develop suitable and generic abstract models of the target machines, supplemented with the formal description of the abstract syntactic rules of the programming languages. Once the target machines can be abstracted by unified mathematic models, formal semantics such as the operational, denotational, axiomatic, and algebraic semantics may be developed as shown in the following subsections.

6.5.1.2 Operational Semantics

Definition 6.48 *Operational semantics* adopts a virtual machine, whose operations are well defined, to describe the semantics of a program in a specific programming language by its *equivalent behaviors* implemented on the virtual machine.

The foundation of operational semantics is based on virtual machine theory [McDermid, 1991]. Virtual machines have been discussed in Section 5.6.1. A typical virtual machine for embodying operational semantics of an arbitrary program is called a *reduction machine* [Louden, 1993]. The reduction machine is used to reduce the given program to values inside the machine and its environment by a finite set of permissible operations.

6.5.1.3 Denotational Semantics

Definition 6.49 *Denotational semantics* adopts functions to describe the semantics of a programming language, in which the *functions* map semantics values into syntactically legal program constructs.

The foundation of denotational semantics is based on recursive function theory [Scott and Strachey, 1971; Jones, 1980; Scott, 1982; Bjorner and Jones, 1982; Schmidt, 1988/94/96]. Denotational semantics is considered a well defined semantics among the existing ones for expressing the meaning of computational instructions in a programming language. In denotational semantics, instructions in a program can be translated into a set of functions based on rigorously defined methodologies.

6.5.1.4 Axiomatic Semantics

Definition 6.50 *Axiomatic semantics* adopts effective assertions to describe the semantics of a programming language, in which the assertions of effects for executing an instruction are deduced to the values of data objects manipulated by the instruction.

The foundation of axiomatic semantics is based on predicate logic [Hoare, 1969; Dijktra, 1975/76; Gries, 1981], where assertions play an important role in axiomatic semantics. Because logical axioms are used in the assertions for denoting program semantics, this method gains the name as axiomatic semantics.

Definition 6.51 An *assertion A* is a logical statement about the predicate behavior *Q* and its initial assumptions *P* of a given instruction *S* at any given point of a program during run-time, which can be examined as true or false, i.e.:

$$A \triangleq \{P\}\ S\ \{Q\} \tag{6.33}$$

where *P* is called the *precondition* and *Q* the *postcondition*.

However, *P* and *Q* are not always specifiable, because some of the operations at run-time are unpredictable or indeterministic, such as event dispatching, parallel mechanisms, and dynamic memory allocations. Nevertheless, assertions have been adopted in a number of modern programming languages, such as C++ and Java. Assertions have also been found useful in formal-specification-based software testing [Yao and Wang, 2004].

Assertions play an important role in correction proving for formal specifications of software systems. As C.A.R. Hoare wrote: "Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs [Hoare, 1969]."

6.5.1.5 Algebraic Semantics

Definition 6.52 *Algebraic semantics* adopts abstract algebra to describe the semantics of a programming language, in which data objects and operations are defined by algebraic axioms and deduced by abstract algebraic laws.

The foundation of algebraic semantics is based on *abstract algebras* [Goguen et al., 1977/96; Guttag and Horning, 1978]. A well-known application of algebraic semantics is the definitions and descriptions of ADTs. Algebraic semantics are capable to deduce the semantics of data objects and imposed operations on abstract types, sorts, and mathematical entities. However, it can not reduce the semantics onto concrete data entities and complex architectures and processes.

6.5.1.6 Deductive Semantics

The aforementioned classic formal semantics were oriented on a certain set of software behaviors that are limited by the models of their semantic environments. The mathematical models of the target machines and the semantic environments in conventional semantics seem to be inadequate to deal with the semantics of complex programming requirements, and to express some important instructions, complex control structures, and the real-time environments at run-time. For supporting systematical and machine enabled semantic analysis and code generation in software engineering, the *deductive semantics* will be introduced that provides a systematic semantic analysis methodology.

Definition 6.53 *Deductive semantics* is a formal software semantics that deduces the semantics of a program in a given programming language from a generic abstract semantic function to the concrete semantics, which are embodied onto the changes of status of a finite set of variables constituting the semantic environment of computing.

The theoretical foundations of deductive semantics are based on *process algebra* and Boolean partial differentials [Wang, 2006a]. Based on the mathematical models and architectural properties of a program at different composing levels as described in Section 5.5, deductive models of software semantics, semantic environment, and semantic matrix will be formally defined in Section 6.5.3. Properties of software semantics and relations between the software behavioral space and semantic environment will be discussed. The deductive semantic rules of RTPA are presented in Section 6.6, which serve both as a comprehensive case study for verifying the expressive and analytic capacity of deductive semantics and as the completion of RTPA as a rigorously defined software engineering notation system.

6.5.2 DENOTATIONAL SEMANTICS

As that of natural languages, the semantic analysis of programs is syntax-directed, in which the semantic definitions are based on a context-free grammar or a set of BNF rules. The border between syntactic and semantic analyses lies at the interfaces of format and meaning, structure and behavior, and static grammar and dynamic implementation. Therefore, it can be perceived that semantics of languages concern everything not specified by the grammar in term of the BNF rules.

Denotational semantics provides a rigorous approach to semantic analysis of programs and programming languages. This subsection uses the sample language SPL as given in Fig. 6.6 and explained in Section 6.3.6 to examine how semantics of programs expressed in this language may be elicited and comprehended formally by using the method of denotational semantics.

6.5.2.1 Syntactic and Semantic Domains of Denotational Semantics

Before proceeding to the description of denotational semantics, the concept of semantic functions and their syntactic and semantic domains are introduced as a preparation.

Definition 6.54 A *semantic function SF* of a programming language is a function *f* that maps a set of syntactic constructs *C* onto a set of associated semantic values in *V*, i.e.:

$$SF = f\colon C \rightarrow V \tag{6.34}$$

Example 6.22 A semantic function of expression evaluation *VAL* that associates an integer value in ℤ to an integer arithmetic expression *E* can be denoted by:

$$VAL = f_{val}\colon E \rightarrow \mathbb{Z}$$

The semantic functions provide a set of deduction rules for a language. Based on those rules, semantics of programs written in the language can be derived onto the values of all variables in a given environment.

Definition 6.55 The *domain* of a semantic function, *C*, is a *syntactic domain*.

Definition 6.56 The *codomain* of a semantic function, *V*, is a *semantic domain*.

Example 6.23 For the above semantic function $VAL = f_{val}\colon E \rightarrow \mathbb{Z}$, its syntactic domain is the domain of *E*, i.e., all syntactically correct expressions according to the given grammar. Its semantic domain is the domain of ℤ, i.e., all binding values of each syntactically correct expression.

In semantic analysis, the semantics of each statement in a program has to be deducted onto the values of all variables or identifiers. Since the identifiers operated by a statement are stored in a physical memory location, semantics of a statement is eventually embodied as the current values of all identifiers that are affected by a given statement. More exactly, semantics of a statement are the changes of the values of all identifiers after the execution of the statement. Hence, the semantic environment of a program in a given programming language is the collection of all values of identifiers stored in predefined memory locations.

Definition 6.57 A *semantic environment* Θ of a programming language is a logical model of a set of identifiers and their values bound in pairs, i.e.:

$$\begin{aligned}\Theta &= f: I \to \mathbb{Z} \\ &= \{\mathop{R}_{k=1}^{\#I} (i_k, v_k)\} \\ &= \{(i_1, v_1), (i_2, v_2), \ldots, (i_{\#I}, v_{\#I})\}\end{aligned} \tag{6.35}$$

where *I* is a set of identifier, $\mathbb{Z}$ a set of integer, and #*I* the number of elements in *I*.

According to Definition 6.57, an *empty environment* Θ_0 is an empty set or a set of arbitrary numbers of undefined identifiers without binding value, i.e.:

$$\begin{aligned}\Theta_0 &= \{(\varnothing, \perp)\} \\ &= \perp\end{aligned} \tag{6.36}$$

where ⊥, called the *bottom*, is used to denote an undefined value.

The evaluation of a syntactic domain (*SD*), such as expressions *E*, statements *S*, or a program *P*, in the presence of an environment Θ can be denoted by:

$$SD \,||\, \Theta \tag{6.37}$$

where || represents a parallel relation between an *SD* and its underlay environment Θ. In other words, any change in *SD* will result in a corresponding change of value for the related identifier(s) in Θ.

For example, according to Eq. 6.37, $P||\Theta$, $S||\Theta$, or $E||\Theta$ represent the evaluation of a program, statement, or expression in the given environment Θ, respectively. When there is no confusion, the context $||\Theta$ may be omitted.

6.5.2.2 Description of Syntactic Domains of the Sample Language SPL

In denotational semantics, the syntactic domain of a language is defined as a set of pairwise relations between a syntactic variable and a syntactic entity. Then, the grammar rules for each of the syntactic variables are defined by an abstract syntax.

As given in Example 6.15, the syntactic domain *SD* of the sample language SPL is defined below:

$$
\begin{aligned}
SD = \{&(\text{P}, \ \text{<program>}),\\
&(\text{L}, \ \text{<statement list>}),\\
&(\text{S}, \ \text{<statement>}),\\
&(\text{E}, \ \text{<expression>}),\\
&(\text{I}, \ \text{<identifier>}),\\
&(\text{A}, \ \text{<letter>}),\\
&(\text{N}, \ \text{<number>}),\\
&(\text{D}, \ \text{<digit>})\} \qquad (6.38)
\end{aligned}
$$

On the basis of the defined *SD*, the abstract syntactic model of SPL can be described in Fig. 6.13, where a terminal symbol '*t*' is enclosed in quotes to denotes that it is different from the operation or value that *t* represents.

```
P ::=  L
L ::=  L_1 ';' L_2
     | S
S ::=  I ':=' E ';'
     | 'if' E 'then' L_1 ';'
     | 'if' E 'then' L_1 'else' L_2 ';'
     | 'while' E 'do' L ';'
E ::=  E_1 '+' E_2
     | E_1 '-' E_2
     | E_1 '*' E_2
     | '(' E_1 ')'
     | I
     | N
I ::= I A | A
A ::= 'a' | 'b' | … | 'z'
N ::= N D | D
D ::= '0' | '1' | … | '9'
```

Figure 6.13 The abstract syntax model of SPL

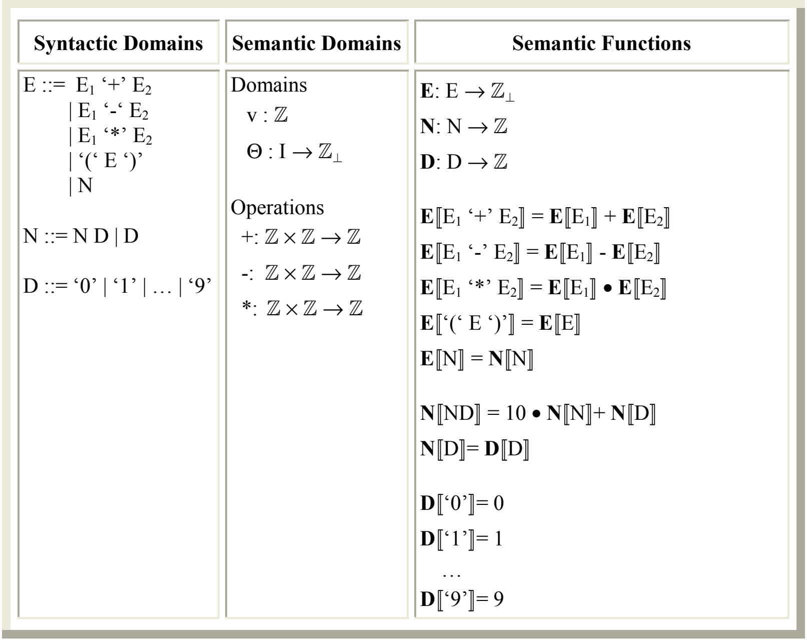
6.5.2.3 Semantic Analysis using Denotational Semantics

The sample language SPL can be divided into three parts: integer arithmetic expressions, assignments, and control constructs. This subsection analyzes those syntactic domains, their corresponding semantic functions, and the semantic meaning of each syntactic domain that may be deducted from the semantic functions. The denotational semantics approach will be used in the analyses.

6.5.2.3.1 Semantics of Integer Arithmetic Expressions

Denotational semantics can be described by a set of semantic functions or rules; each of them maps a syntactic domain into a semantic domain. The semantics of the integer arithmetic expressions of SPL in denotational semantics can be described by a set of semantic functions defined on given syntactic and semantic domains as shown in Table 6.6.

Table 6.6
Semantic Analysis of Integer Arithmetic Expressions



Syntactic Domains	Semantic Domains	Semantic Functions
E ::= E_1 ‘+’ E_2 | E_1 ‘-‘ E_2 | E_1 ‘*’ E_2 | ‘(‘ E ‘)’ | N N ::= N D | D D ::= ‘0’ | ‘1’ | … | ‘9’	Domains $v : \mathbb{Z}$ $\Theta : I \to \mathbb{Z}_\perp$ Operations $+: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $-: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $*: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$\mathbf{E}: E \to \mathbb{Z}_\perp$ $\mathbf{N}: N \to \mathbb{Z}$ $\mathbf{D}: D \to \mathbb{Z}$ $\mathbf{E}[\![E_1 \text{ ‘+’ } E_2]\!] = \mathbf{E}[\![E_1]\!] + \mathbf{E}[\![E_2]\!]$ $\mathbf{E}[\![E_1 \text{ ‘-’ } E_2]\!] = \mathbf{E}[\![E_1]\!] - \mathbf{E}[\![E_2]\!]$ $\mathbf{E}[\![E_1 \text{ ‘*’ } E_2]\!] = \mathbf{E}[\![E_1]\!] \bullet \mathbf{E}[\![E_2]\!]$ $\mathbf{E}[\![\text{‘(‘ } E \text{ ‘)’}]\!] = \mathbf{E}[\![E]\!]$ $\mathbf{E}[\![N]\!] = \mathbf{N}[\![N]\!]$ $\mathbf{N}[\![ND]\!] = 10 \bullet \mathbf{N}[\![N]\!] + \mathbf{N}[\![D]\!]$ $\mathbf{N}[\![D]\!] = \mathbf{D}[\![D]\!]$ $\mathbf{D}[\![\text{‘0’}]\!] = 0$ $\mathbf{D}[\![\text{‘1’}]\!] = 1$ … $\mathbf{D}[\![\text{‘9’}]\!] = 9$

The syntactic domain of SPL, *SD* = {*E, N, D*}, is specified in the left column of Table 6.6. It encompasses three syntactic sub-domains, *E* <Expression>, *N* <Number>, and *D* <Digit> as defined in EBNF.

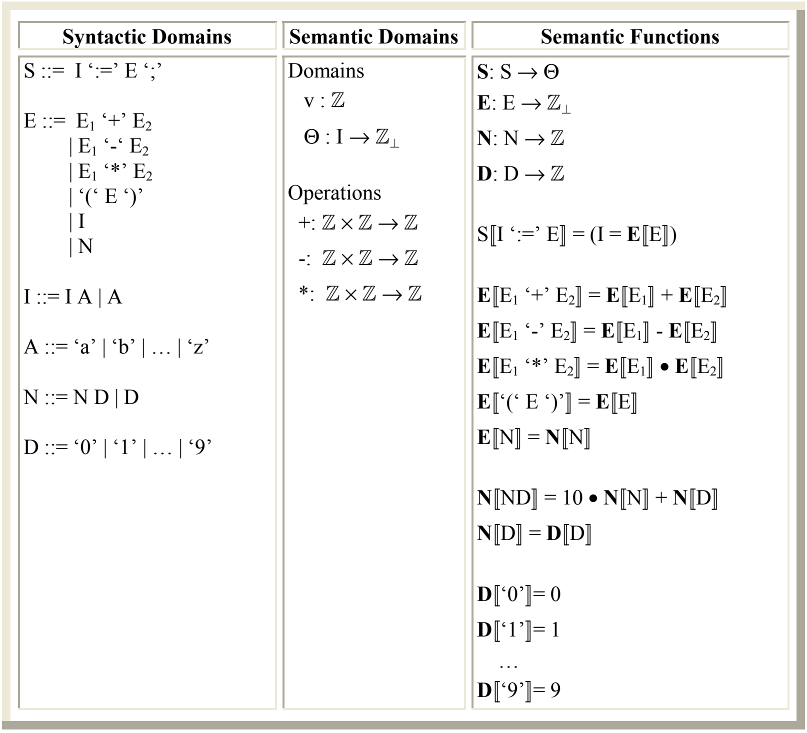
In the semantic domains, the domains of variables and the environment, as well as their allowable operations, are defined in the middle column of Table 6.6. A generic variable *v* is introduced in the simple type integer $\mathbb{Z}$ for all identifiers in SPL. The environment Θ is defined as a function that maps a set of identifiers I into $\mathbb{Z}$, which is undefined ($\perp$) initially. On the basis of the declarations of types of variables, three operations, +, _, *, are defined as functions that operate two integer variables and result in an integer value.

In the semantic function column of Table 6.6, there are two types of semantic functions: the generic and the derived semantic functions. The three generic semantic functions, **E**, **N**, and **D**, are schemas of functions that map a corresponding syntactic domain into a value in $\mathbb{Z}$**.** On the basis of the generic functional schemas, a set of 17 derived semantic functions are defined for each of the corresponding production rules as defined in the syntactic domains.

6.5.2.3.2 Semantics of Assignments

This subsection extends the SPL to cover the semantics of the assignment statement in denotational semantics as shown in Table 6.7.

Table 6.7
Semantic Analysis of Assignments



Syntactic Domains	**Semantic Domains**	**Semantic Functions**
S ::= I ‘:=’ E ‘;’ E ::= E_1 ‘+’ E_2 | E_1 ‘-‘ E_2 | E_1 ‘*’ E_2 | ‘(‘ E ‘)’ | I | N I ::= I A | A A ::= ‘a’ | ‘b’ | … | ‘z’ N ::= N D | D D ::= ‘0’ | ‘1’ | … | ‘9’	Domains $v : \mathbb{Z}$ $\Theta : I \to \mathbb{Z}_\perp$ Operations $+: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $-: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $*: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$\mathbf{S}: S \to \Theta$ $\mathbf{E}: E \to \mathbb{Z}_\perp$ $\mathbf{N}: N \to \mathbb{Z}$ $\mathbf{D}: D \to \mathbb{Z}$ S⟦I ‘:=’ E⟧ = (I = **E**⟦E⟧) **E**⟦E_1 ‘+’ E_2⟧ = **E**⟦E_1⟧ + **E**⟦E_2⟧ **E**⟦E_1 ‘-’ E_2⟧ = **E**⟦E_1⟧ - **E**⟦E_2⟧ **E**⟦E_1 ‘*’ E_2⟧ = **E**⟦E_1⟧ • **E**⟦E_2⟧ **E**⟦‘(‘ E ‘)’⟧ = **E**⟦E⟧ **E**⟦N⟧ = **N**⟦N⟧ **N**⟦ND⟧ = 10 • **N**⟦N⟧ + **N**⟦D⟧ **N**⟦D⟧ = **D**⟦D⟧ **D**⟦‘0’⟧= 0 **D**⟦‘1’⟧= 1 … **D**⟦‘9’⟧= 9

The syntactic domains *SD* = {*E, N, D*} in Table 6.6 are extended by *S* <Statement>, *I* <Identifier>, and *A* <Letter> with corresponding EBNF deduction rules. Therefore, the syntactic domains of assignment statements for semantic analysis include six domains, i.e., *SD* = {*S, E, I, A, N, D*} as shown in Table 6.7.

The semantic domains and the environment Θ remain the same for interpreting the results of semantic analysis.

The new generic semantic function ***S*** maps an assignment statement into the environment Θ, i.e., into an integer identifier in type $\mathbb{Z}$. The new derived semantic function of assignment, $\mathbf{S}[\![\mathrm{I}\ \text{‘:=’}\ \mathrm{E}]\!] = (\mathrm{I} = \mathbf{E}[\![\mathrm{E}]\!])$, denotes that an assignment $\mathbf{S}[\![\mathrm{I}\ \text{‘:=’}\ \mathrm{E}]\!]$ in the environment Θ is an evaluation of the expression *E*, **E**, in Θ, and transfers the value of *E* to the identifier *I*.

6.5.2.3.3 Semantics of Branch Statements

This subsection extends the SPL to cover the semantics of the branch statement in denotational semantics as shown in Table 6.8.

Table 6.8
Semantic Analysis of Branch Construct

Syntactic Domains	Semantic Domains	Semantic Functions
$\mathrm{L} ::= \ \mathrm{L}_1$ ‘;’ L_2 | S S ::= ‘if’ E ‘then’ L_1 ‘else’ L_2 ‘;’ E ::= E_1 ‘+’ E_2 | E_1 ‘-‘ E_2 | E_1 ‘*’ E_2 | ‘(‘ E ‘)’ | I | N	Domains $\mathrm{v} : \mathbb{Z}$ $\Theta : \mathrm{I} \to \mathbb{Z}_\perp$ $\mathsf{T}: \mathbb{Z} \mid \mathsf{T} = 1$ $\mathsf{F}: \mathbb{Z} \mid \mathsf{F} = 0$ Operations $+: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $-: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $*: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$\mathbf{E}: \mathrm{E} \to \mathbb{Z}_\perp$ $\mathbf{L}: \mathrm{L} \to \Theta$ $\mathbf{S}: \mathrm{S} \to \Theta$ $\mathbf{L}[\![\mathrm{L}_1\ \text{‘;’}\ \mathrm{L}_2]\!] = \mathbf{L}[\![\mathrm{L}_2]\!] \circ \mathbf{L}[\![\mathrm{L}_1]\!]$ $\mathbf{L}[\![\mathrm{S}]\!] = \mathbf{S}[\![\mathrm{S}]\!]$ $\mathbf{S}[\![$‘if’ E ‘then’ L_1 ‘else’ L_2 ’;’ $]\!]$ = if ($\mathbf{E}[\![\mathrm{E}]\!] = \mathsf{T}$ then $\mathbf{L}[\![\mathrm{L}_1]\!]$ else $\mathbf{L}[\![\mathrm{L}_2]\!]$

The syntactic domains in Table 6.8 are extended by adding a <statement-list> *L*, i.e., *SD* = {*S, E, I, A, N, D*} ∪ {*L*}.

The environment Θ in the semantic domain remains unchanged. However, two integer constants **T** and **F** are introduced with the value ‘1’ and ‘0’ respectively. The reason for defining these two Boolean constants as integer is that Θ and all variables *v* in SPL have been restricted in $\mathbb{Z}$ for simplicity in analysis.

In the semantic functions, a statement-list **L** for a single statement *S*, $\mathbf{L}[\![S]\!]$, is interpreted as a derived function of an ordinary statement $\mathbf{S}[\![S]\!]$. Another semantic function $\mathbf{L}[\![L_1\ \text{‘;’}\ L_2]\!] = \mathbf{L}[\![L_2]\!] \circ \mathbf{L}[\![L_1]\!]$ denotes that two sequential statement-lists L_1 followed by L_2 are semantically equivalent to the execution of L_1 and then L_2, which is denoted as a concatenation of functions $L_2 \circ L_1$.

The semantic function of a branch statement $\mathbf{S}[\![$*'if' E 'then' L_1 'else' L_2 ';'*$]\!]$ interprets the branch operation as the choice of execution of two optional statement-lists on the conditional expression *E*. When the evaluation of *E*, i.e., the resulted value $\mathbf{E}[\![E]\!] = 1$ (**T**), L_1 will be executed; otherwise, L_2 will be executed.

6.5.2.3.4 Semantics of While-Loop Statements

This subsection extends the SPL to cover the semantics of the while-loop statement in denotational semantics as shown in Table 6.9.

Table 6.9
Semantic Analysis of While Loop Construct

Syntactic Domains	Semantic Domains	Semantic Functions
L ::= L_1 ';' L_2 | S S ::= 'while' E 'do' L ';' E ::= E_1 '+' E_2 | E_1 '-' E_2 | E_1 '*' E_2 | '(' E ')' | I | N	Domains $v : \mathbb{Z}$ $\Theta : I \to \mathbb{Z}_\perp$ $\mathbf{T}: \mathbb{Z} \mid \mathbf{T} = 1$ $\mathbf{F}: \mathbb{Z} \mid \mathbf{F} = 0$ Operations $+: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $-: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ $*: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$\mathbf{E}: E \to \mathbb{Z}_\perp$ $\mathbf{L}: L \to \Theta$ $\mathbf{S}: S \to \Theta$ $\mathbf{L}[\![L_1$ ';' $L_2]\!] = \mathbf{L}[\![L_2]\!] \circ \mathbf{L}[\![L_1]\!]$ $\mathbf{L}[\![S]\!] = \mathbf{S}[\![S]\!]$ $\mathbf{S}[\![$'while' E 'do' L ';'$]\!]$ $= \overset{\mathbf{F}}{\underset{E=\mathbf{T}}{R}}\mathbf{L}\ \mathbf{L}[\![L]\!]$

Note that the semantics of the while loop, according to the definitions of the generic iteration representation as discussed in Section 5.4.2, is described as a *repeat function* that evaluates *E* , if it is true then execute *L* whenever *E* remains true, i.e.:

$$\mathbf{S}[\![\text{'while' } E \text{ 'do' } L \text{ ';'}]\!] = \overset{\mathbf{F}}{\underset{E=\mathbf{T}}{R}} \mathbf{L}[\![L]\!] \tag{6.39}$$

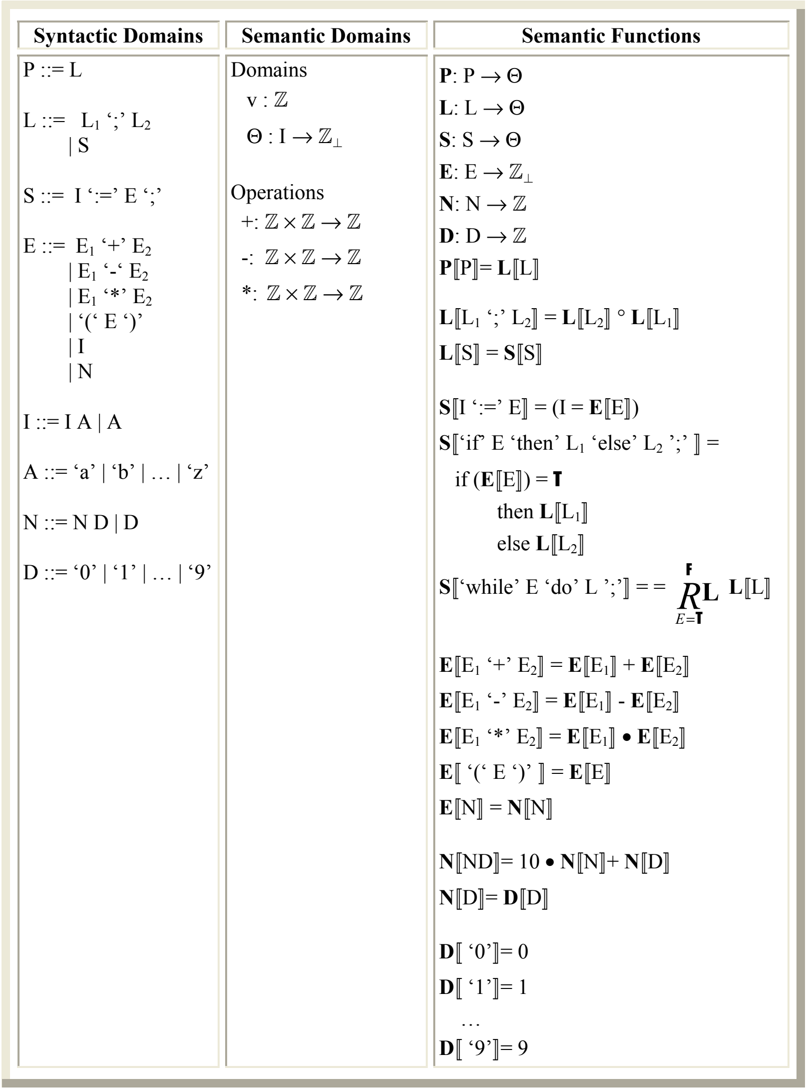
Eq. 6.39 is equivalent to the conventional representation of the while loop by recursive if-then-else structures in the literature [Wirth, 1976; Louden, 1993] as described below:

$$\begin{aligned}
&\mathbf{S}[\![\text{'while' } E \text{ 'do' } L \text{ ';'}]\!] = \\
&\quad \mathbf{S}[\![\textit{if } \mathbf{E}[\![E]\!] = \mathbf{T} \\
&\qquad\quad \textit{then } \mathbf{L}[\![L;\, S]\!] \\
&\qquad\quad \textit{else } \Theta \\
&\quad]\!]
\end{aligned} \tag{6.40}$$

6.5.2.4 Semantics of Programs in SPL

On the basis of the semantic analysis methods developed in Section 6.5.2.3, the semantics of a complete program can be derived in denotational semantics. This subsection describes the semantics of a whole program in SPL that covers all syntactic domains discussed earlier with new extensions of domains of a program *P* and statement-list *L*, as shown in Table 6.10.

Table 6.10
Semantic Analysis of a Whole Program in SPL



<table>
<tr><th>Syntactic Domains</th><th>Semantic Domains</th><th>Semantic Functions</th></tr>
<tr>
<td>P ::= L
L ::= L_1 ‘;’ L_2
| S
S ::= I ‘:=’ E ‘;’
E ::= E_1 ‘+’ E_2
| E_1 ‘-‘ E_2
| E_1 ‘*’ E_2
| ‘(‘ E ‘)’
| I
| N
I ::= I A | A
A ::= ‘a’ | ‘b’ | … | ‘z’
N ::= N D | D
D ::= ‘0’ | ‘1’ | … | ‘9’</td>
<td>Domains
$v : \mathbb{Z}$
$\Theta : I \to \mathbb{Z}_\perp$
Operations
$+: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$-: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$*: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$</td>
<td>$\mathbf{P}: P \to \Theta$
$\mathbf{L}: L \to \Theta$
$\mathbf{S}: S \to \Theta$
$\mathbf{E}: E \to \mathbb{Z}_\perp$
$\mathbf{N}: N \to \mathbb{Z}$
$\mathbf{D}: D \to \mathbb{Z}$
$\mathbf{P}⟦P⟧= \mathbf{L}⟦L⟧$
$\mathbf{L}⟦L_1$ ‘;’ $L_2⟧ = \mathbf{L}⟦L_2⟧ \circ \mathbf{L}⟦L_1⟧$
$\mathbf{L}⟦S⟧ = \mathbf{S}⟦S⟧$
$\mathbf{S}⟦$I ‘:=’ E$⟧$ = (I = $\mathbf{E}⟦E⟧$)
$\mathbf{S}⟦$‘if’ E ‘then’ L_1 ‘else’ L_2 ’;’ $⟧$ =
if ($\mathbf{E}⟦E⟧$) = $\mathbf{T}$
then $\mathbf{L}⟦L_1⟧$
else $\mathbf{L}⟦L_2⟧$
$\mathbf{S}⟦$‘while’ E ‘do’ L ’;’$⟧$ = = $\overset{\mathbf{F}}{\underset{E=\mathbf{T}}{R}}\mathbf{L}$ $\mathbf{L}⟦L⟧$
$\mathbf{E}⟦E_1$ ‘+’ $E_2⟧ = \mathbf{E}⟦E_1⟧ + \mathbf{E}⟦E_2⟧$
$\mathbf{E}⟦E_1$ ‘-’ $E_2⟧ = \mathbf{E}⟦E_1⟧ - \mathbf{E}⟦E_2⟧$
$\mathbf{E}⟦E_1$ ‘*’ $E_2⟧ = \mathbf{E}⟦E_1⟧ \bullet \mathbf{E}⟦E_2⟧$
$\mathbf{E}⟦$ ‘(‘ E ‘)’ $⟧ = \mathbf{E}⟦E⟧$
$\mathbf{E}⟦N⟧ = \mathbf{N}⟦N⟧$
$\mathbf{N}⟦ND⟧= 10 \bullet \mathbf{N}⟦N⟧+ \mathbf{N}⟦D⟧$
$\mathbf{N}⟦D⟧= \mathbf{D}⟦D⟧$
$\mathbf{D}⟦$ ‘0’$⟧$= 0
$\mathbf{D}⟦$ ‘1’$⟧$= 1
…
$\mathbf{D}⟦$ ‘9’$⟧$= 9</td>
</tr>
</table>

It is noteworthy that the sample language SPL analyzed in this subsection is quite simple. Although the basic semantic deduction rules for arithmetic, assignment statements, and the branch and while loop constructs have been covered, complex statements and program constructs such as I/O environment, recursive and interrupt constructs, as well as event handling have not been touched. They are remaining as important topics in programming semantic studies yet to be explored.

6.5.3 DEDUCTIVE SEMANTICS

Deduction is an inference process that discovers new knowledge or derives a specific conclusion based on generic premises such as abstract rules or principles. The nature of semantics of a given programming language is its computational meanings or embodied behaviors expressed by an instruction in the language. Because the carriers of software semantics are a finite set of variables declared in a given program, program semantics can be reduced onto the changes of values of these variables over time. In order to provide a rigorous mathematical treatment of both the abstract and concrete semantics of software, a new type of formal semantics known as the deductive semantics is developed [Wang, 2006a].

Deductive semantics as given in Definition 6.53 is a formal software semantics that deduces the semantics of a program in a given programming language from a generic abstract semantic function to the concrete semantics, which are embodied onto the changes of status of a finite set of variables constituting the semantic environment of computing. Deductive semantics can be used to define both abstract and concrete semantics of large-scale software systems, facilitate software comprehension and recognition, support tool development, enable semantics-based software testing and verification, and explore the semantic complexity of software systems.

This subsection develops the mathematical models of deductive semantics and elicits the fundamental properties of software semantics. The deductive models of semantics, semantic function, and semantic environment at various composing levels of programs are introduced. Properties of software semantics and relationships between the software behavioral space and the semantic environment are studied. New methods such as the semantic differential and semantic matrix are developed to facilitate deductive semantic analyses from a generic semantic function to a specific semantic matrix, and from semantics of statements to those of processes and programs. The establishment of the deductive semantic rules of RTPA will be described in Section 6.6, where deductive semantics of a comprehensive set of software processes is modeled.

6.5.3.1 The Mathematic Model of Software Semantics

A *semantic environment* of a program in a given programming language is a logical model of a finite set of identifiers and their values changing over time along the execution of the program. The semantic environment constituting the behaviors of software is inherently a three dimensional structure known as the operations, memory space, and time.

According to Theorem 3.10, the *behavioral space* Ω of a program executed on a certain machine is a finite set of variables operated in a 3-D state space determined by a triple, $\Omega \triangleq (OP, T, S)$, where *OP* is a finite set of operations, *T* is a finite set of discrete steps of program execution, and *S* is a finite set of memory locations or their logical representations by identifiers of variables.

According to Definition 6.45, the set of *variables* of a program, *S,* plays an important role in semantic analysis, because they are the objects of software behavioral operations and the carriers of program semantics. Variables can be classified as *free* and *system* variables. The former are user defined and the latter are language provided. From a functional point of view, variables can be classified into *object representatives, control variables, result containers,* and *address locaters*. The life spans or scopes of variables can be categorized as *persistent, global, local,* and *temporal*. The persistent variables are those that their life span are longer than the program that generates them, such as data in a database or files in distributed networks.

A new mathematical operator introduced in deductive semantics is the *partial differential of sets* on the basis of Boolean differential, which is used to facilitate the instantiation of abstract semantics into concrete ones.

Definition 6.58 Given two sets *X* and *V, X* ⊆ *V*, a *partial differential* of *X* on *V* with elements *x*, *x* ∈ *X*, denoted by ∂V/∂x, is an elicitation of interested elements from *V* as specified in *X*, i.e.:

$$\begin{aligned} \frac{\partial V}{\partial x} &= X \cap V, \quad x \in X \subseteq V \\ &= X \end{aligned} \tag{6.41}$$

The partial differential of sets can be easily extended to double, triple, or, more generally, multiple partial differentials as defined below.

Definition 6.57 A *multiple partial differential* of $X_1, X_2, \ldots,$ and X_n on *V* with elements $x_1 \in X_1, x_2 \in X_2, \ldots,$ and $x_n \in X_n$, denoted by $\frac{\partial^n}{\partial x_1\, \partial x_2 \ldots \partial x_n} V$, is a Cartesian product of all partial differentials that select interested elements from V as specified in $X_1, X_2, \ldots,$ and X_n, respectively, i.e.:

$$\frac{\partial^{\mathrm{n}}}{\partial x_1\, \partial x_2 \ldots \partial x_n} V = X_1 \times X_2 \times \ldots \times X_n \tag{6.42}$$

where $X_1, X_2, \ldots, X_n \subseteq V$.

For example, $\frac{\partial^2}{\partial \mathrm{x}\, \partial \mathrm{y}} V = X \times Y,\ x \in X \wedge y \in Y \wedge X, Y \subseteq V$ and $\frac{\partial^3}{\partial \mathrm{x}\, \partial \mathrm{y}\, \partial z} V =$ $X \times Y \times Z,\ x \in X \wedge y \in Y \wedge z \in Z \wedge X, Y, Z \subseteq V$.

On the basis of the definitions of software behavioral space and partial differential of sets, the semantic environment of software can be introduced as follows.

Definition 6.60 The *semantic environment* Θ of a program on a certain target machine is its run-time behavioral space $\Omega(OP, T, S)$ projected onto the Cartesian plane determined by T and S, i.e.:

$$\begin{aligned} \Theta &= \frac{\partial^2 \Omega}{\partial t\, \partial s},\ t \in T \wedge s \in S \\ &= \frac{\partial^2 \Omega}{\partial t\, \partial s}(OP, T, S) \\ &= T \times S \end{aligned} \tag{6.43}$$

where, T is a finite set of discrete steps of program execution, S is a finite set of memory locations or their logical representations by identifiers of variables.

As indicated in Definition 6.60, the semantic environment of a program is a dynamic entity over time, because following each execution of a statement in the program, the semantic environment Θ, particularly the set of values V of the variables S may be changed as a result of the operation of the statement.

A generic semantic function is developed below, which can be used to derive a specific and concrete semantic function for a given statement, process, or program at different composing levels by mathematical deduction.

Definition 6.61 A *semantic function* of a program $\wp$, $f_\theta(\wp)$, is a function that maps the semantic environment Θ into a finite set of values V

determined by a Cartesian product on a finite *set of executing steps T* and a finite *set of variables S*, i.e.:

$$f_\theta(\wp) = f\!: T \times S \to V$$

$$= \left(\begin{array}{ccccc} & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\ \mathbf{t_0} & \perp & \perp & \cdots & \perp \\ \mathbf{t_1} & v_{11} & v_{12} & & v_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{t_n} & v_{n1} & v_{n1} & \cdots & v_{nm} \end{array} \right) \qquad (6.44)$$

where $T = \{t_0, t_1, ..., t_n\}$, $S = \{s_1, s_2, ..., s_m\}$, and V is a finite set of values $v(t_i, s_j)$, $0 \le i \le n$, and $1 \le j \le m$.

In Eq. 6.44, all values of $v(t_i, s_j)$ at t_0 are undefined for a program as denoted by the bottom symbol ⊥, i.e., $v(0,\ s_j) =\ \perp,\ 1 \le j \le m$. However, for a statement or a process, it is usually true that $v(0, s_j) \ne\ \perp$ dependent on the context of previous statement(s) or the initialization of the system.

According to Definition 6.61, the semantic environment and the domain of a semantic function can be illustrated by a semantic diagram [Wang, 2006a] as described below.

Definition 6.62 A *semantic diagram* is a sub-Cartesian-plane in the semantic environment Θ that forms the domain of the semantic function for a composed process P with $f_\theta(P) = f\!: T_P \times S_P \to V_P$.

The semantic diagram $f_\theta(P)$ as defined in Definition 6.62 can be illustrated in Fig. 6.14, where S_P is the set of variables of process P.

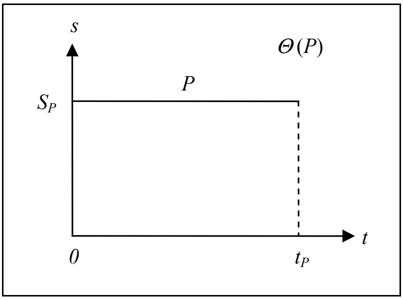


Figure 6.14 The semantic diagram of a process

The semantic diagram can be used to analyze complex semantic relations, and to demonstrate semantic functions and their semantic environments. Observing Fig. 6.14, the flowing properties of semantic function for composed processes can be derived.

Corollary 6.4 The variables of two arbitrary processes P and Q, S_P and S_Q, in the semantic environment Θ possess the following properties:

a) The *entire set of variables*: $S = S_p \cup S_Q$ (6.45)

b) *Global variables*: $S_G \subseteq S_p \cap S_Q$ (6.46)

c) *Local variables*: $S_L = S - S_G$, $S_L \subseteq S_p \oplus S_Q$,
where $S_{Lp} = S_L \setminus S_Q$ and $S_{Lq} = S_L \setminus S_p$ (6.47)

6.5.3.2 Deductive Semantics of Programs at Different Levels of Compositions

It is noteworthy that deductive semantics introduces only a universal semantic function as given in Definition 6.61, rather than adopting multiple concrete semantic functions as the conventional approaches do. In deductive semantics, any particular concrete semantic function is a deduced instantiation of the universal abstract semantic function. This is why it is named deductive semantics, and this avoids the trouble in other exhaustive approaches where a new semantic function has to be particularly defined from time to time whenever additional instruction is introduced in a given language.

According to the architectural model of programs as described in Section 5.5.1, the semantics of a program in a given language can be described and analyzed at various composition levels, such as those of *statement*, *process*, and *system* from the bottom up.

Definition 6.63 The *semantics of a statement p*, *θ(p)*, on a given semantic environment Θ is a double partial differential of the semantic function $f_\theta(p)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}\theta(p) &= \frac{\partial^2}{\partial t\,\partial s} f_\theta(p)\\ &= \underset{i=0}{\overset{\#T(p)}{R}}\ \underset{j=1}{\overset{\#S(p)}{R}} v_p(t_i, s_j)\\ &= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{\#\{s_1, s_2, ..., s_m\}}{R}} v_p(t_i, s_j)\end{aligned}$$

$$= \left(\begin{array}{cccc} & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\ \mathbf{t_0} & v_{01} & v_{02} & \cdots & v_{0m} \\ (\mathbf{t_0}, \mathbf{t_1}] & v_{11} & v_{12} & \cdots & v_{1m} \end{array} \right) \qquad (6.48)$$

where *t* denotes the discrete time immediately before and after the execution of *p* during (t_0, t_1], and # is the *cardinal calculus* that counts the number of elements in a given set, i.e., $n = \#T(p)$ and $m{=}\#S(p)$.

In Definition 6.63, the first partial differential selects all related variable *S*(*p*) of the statement *p* from Θ. The second partial differential selects a set of discrete steps of *p*'s execution *T*(*p*) from Θ. According to Definition 6.63, the semantics of a statement can be deduced onto a semantic function that results in a 2-D matrix with the changes of values of all variables over time of program execution.

On the basis of Definitions 6.61 and 6.63, semantics of any statements in a given programming language can be analyzed using Eq. 6.48 via a deductive process.

Example 6.24 Analyze the semantics of Statement 3, p_3, in the following program entitled *sum*.

```
void sum;
   {
    (0)   int x, y, z;
    (1)   x = 8;
    (2)   y = 2;
    (3)   z := x + y;
   }
```

According to Definition 6.63, the semantics of Statement p_3 is as follows:

$$\begin{aligned} \theta(p_3) &= \frac{\partial^2}{\partial t\ \partial s} f_\theta(p_3) \\ &= \mathop{R}\limits_{i=2}^{3} \mathop{R}\limits_{j=1}^{\#S(p_3)} v_{p_3}(t_i, s_j) \\ &= \mathop{R}\limits_{i=2}^{3} \mathop{R}\limits_{j=1}^{\#\{x,y,z\}} v_{p_3}(t_i, s_j) \\ &= \left(\begin{array}{cccc} & \mathbf{x} & \mathbf{y} & \mathbf{z} \\ \mathbf{t}_2 & 8 & 2 & \perp \\ (\mathbf{t}_2, \mathbf{t}_3] & 8 & 2 & 10 \end{array} \right) \end{aligned} \qquad (6.49)$$

This example shows how the concrete semantics of a statement can be derived on the basis of the generic and abstract semantic function of deductive semantics.

In semantic analysis, the changed part of the semantic environment Θ, known as the semantic effect as defined below, is particularly interested, which is the embodiment of software semantics.

Definition 6.64 The *semantic effect* of a statement *p*, *θ** (*p*), is the resulting changes of values of variables by its semantic function *θ*(*p*) during the time interval immediately before and after the execution of *p*, $\Delta t = (t_i, t_{i+1}]$, i.e.:

$$\begin{aligned}\theta^*(p) &= \mathop{R}_{j=1}^{\#S(p)} (v_p(t_i, s_j) \oplus v_p(t_{i+1}, s_j)) \\ &= \mathop{R}_{j=1}^{\#S(p)} < v_p(t_i, s_j) \rightarrow v_p(t_{i+1}, s_j) \mid v_p(t_i, s_j) \neq v_p(t_{i+1}, s_j) >\end{aligned} \tag{6.50}$$

where → denotes a transition of values for a given variable.

Example 6.25 For the same statement p_3 as shown in Example 6.24, determine its semantic effect $\theta^*(p_3)$.

According to Eq. 6.50, the semantic effect $\theta^*(p_3)$ is:

$$\begin{aligned}\theta^*(p_3) &= \mathop{R}_{j=1}^{\#S(p_3)} < v_{p_3}(t_2, s_j) \rightarrow v_{p_3}(t_3, s_j) \mid v_{p_3}(t_2, s_j) \neq v_{p_3}(t_3, s_j) > \\ &= \mathop{R}_{j=1}^{\#(x,y,z)} < v_{p_3}(t_2, s_j) \rightarrow v_{p_3}(t_3, s_j) \mid v_{p_3}(t_2, s_j) \neq v_{p_3}(t_3, s_j) > \\ &= \{< v_{p_3}(t_2, z) = \perp \rightarrow v_{p_3}(t_3, z) = 10 >\}\end{aligned}$$

It can be seen in Examples 6.24 and 6.25 that deductive semantics can be used not only to describe the *abstract* and *concrete* semantics of programs, but also to elicit and highlight their semantic effects.

According to Theorem 5.7 as well as Definitions 5.51 and 5.53, a program or a process is composed by individual statements with given rules of compositions. Therefore, the definitions and mathematical models of semantics at the statement level can be extended onto the higher levels of program hierarchy systematically.

Definition 6.65 The *semantics of a process P*, *θ(P)*, on a given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(P) &= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P) \\
&= \overset{n-1}{\underset{k=1}{R}} \{[\frac{\partial^2}{\partial t\,\partial s} f_\theta(P_k)]\; r_{kl}\; [\frac{\partial^2}{\partial t\,\partial s} f_\theta(P_l)]\}, l = k+1 \\
&= \overset{n-1}{\underset{k=1}{R}} \{[\overset{\#T(P_k)}{\underset{i=0}{R}} \overset{\#S(P_k)}{\underset{j=1}{R}} v_{P_k}(t_i, s_j)]\; r_{kl}\; [\overset{\#T(P_l)}{\underset{i=0}{R}} \overset{\#S(P_l)}{\underset{j=1}{R}} v_{P_l}(t_i, s_j)]\} \\
&= \begin{pmatrix}
\mathbf{V}_{\mathbf{P_1}} & & & & \mathbf{V_G} \\
& \mathbf{V}_{\mathbf{P_2}} & & & \mathbf{V_G} \\
& & \ddots & & \vdots \\
& & & \mathbf{V}_{\mathbf{P_{n-1}}} & \mathbf{V_G}
\end{pmatrix}
\end{aligned}
\qquad (6.51)
$$

where $\mathbf{V}_{\mathbf{P}_k}$, $1 \le k \le n\text{-}1$, is a set of values of local variables that belongs to processes P_k, and $\boldsymbol{V_G}$ is a finite set of values of global variables.

On the basis of Definition 6.65, the semantics of a program at the top-level composition can be deduced to the combination of semantics of a set of processes, each of which can be further deduced to the composition of all statements' semantics as described below.

Definition 6.66 The *semantics of a program* $\wp$, $\theta(\wp)$, on a given semantic environment Θ, is a combination of the semantic functions of all processes $\theta(P_k)$, $1 \le k \le n$, i.e.:

$$
\begin{aligned}
\theta(\wp) &= \overset{\#K(\wp)}{\underset{k=1}{R}} \frac{\partial^2}{\partial t\,\partial s} f_\theta(\wp) \\
&= \overset{\#K(\wp)}{\underset{k=1}{R}} \theta(P_k) \\
&= \overset{\#K(\wp)}{\underset{k=1}{R}} [\overset{\#T(P_k)}{\underset{i=0}{R}} \overset{\#S(P_k)}{\underset{j=1}{R}} v_{P_k}(t_i, s_j)]
\end{aligned}
\qquad (6.52)
$$

where $\#K(\wp)$ is the number of processes or components in the program.

It is noteworthy that Eq. 6.52 will usually result in a very large matrix of semantic space, which can be quantitatively predicated as follows.

Definition 6.67 The *semantic space of a program* $S_\Theta(\wp)$ is a product between the number of variables $\#S(\wp)$ and the number of executing steps $\#T(\wp)$, i.e.:

$$\begin{aligned} S_{\Theta}(\wp) &= \#S(\wp) \bullet \#T(\wp) \\ &= \sum_{\text{k=1}}^{\#K(\wp)} \#S(\wp_k) \bullet \sum_{\text{k=1}}^{\#K(\wp)} \#T(\wp_k) \end{aligned} \tag{6.53}$$

The semantic space of a program provides a useful measure for software complexity. Due to the tremendous size of the semantic space, both program composition and comprehension are innately a hard problem in terms of complexity and cognitive difficulty.

6.5.3.3 Properties of Software Semantics

Observing the formal definitions and mathematical models of deductive semantics developed in previous sections, a number of common properties of software semantics may be elicited, which are useful for explaining the fundamental characteristics of software semantics.

One of the most interesting characteristics of program semantics is its invariance against different executing speeds as described in the following theorem.

The 19th Law of Software Engineering

Theorem 6.2 The *asynchronicity of program semantics* states that the semantics of a relatively timed program is invariant with the changes of executing speed, as long as any absolute time constraint is met.

Theorem 6.2 states that, for most nonreal-time or relatively timed programs, different executing speeds or simulation paces will not alter the semantics of the software system. This explains why a programmer may simulate the run-time behaviors of a given program that may be executing at a speed of up to 10^9 times faster than that of human beings. It also explains why computers with different system clock frequencies may correctly run the same program and obtain the same behavior.

A fundamental question in programming language and software engineering theories is what the *least complete set of instructions* for programming is. As discussed in Sections 5.2.1 and 5.4, the meta instructions shared by all programming languages can be classified into three categories: a) *Internal operations*, such as memory manipulation and assignments; b) *External operations,* such as input/output, event handling, and human-machine interactions; and c) *Basic control structures* (BCS's) [Wang, 2005a/06c/06h/06f], such as the jump, branch, and iteration constructs as summarized in Tables 5.2, 5.14, and 5.15. Based on the above exploration, the sets of sufficient meta instructions and their algebraic compositional rules have been elicited in Theorems 4.6 and 4.7, respectively. According to these theorems, the questions on the least complete set of instructions in programming can be formally answered below.

The 20th Law of Software Engineering

Theorem 6.3 The *least complete set of instructions* in programming states that a program is *composable* with sufficient descriptive power in a given language *iff* both the sufficient sets of meta instructions ($\mathfrak{P}$, Theorem 4.6) and compositional rules ($\mathfrak{R}$, Theorem 4.7) are rigorously defined.

Theorem 6.3 indicates that the necessary and sufficient conditions of program *compositionality* in a given language are that all the meta instructions (Table 4.8) and the fundamental BCS's (Table 5.16) must be implemented in the language. In case of nonreal-time programming languages, the requirement for the four special BCS's, BCS's #10 through #13, may be waived. However, it is helpful to be aware of the whole set of software compositional rules for both ordinary and real-time software systems.

It is noteworthy that some of the BCS's as shown in Table 5.16 were used to be treated as basic instructions rather than compositional rules in conventional programming and formal techniques. However, according to Theorems 4.3, 4.8, and 6.3, there is a need to distinguish the semantic roles of statements (the *minimum semantic unit* of a language) and BCS's (the compositional rules of the language).

Definition 6.68 The *behavior* of a computational statement is a set of observable actions or changes of status of objects operated by the statement.

According to Theorem 3.10, the behavioral space of software Ω is three dimensional; while as given in Definition 6.58, the semantic environment Θ is two dimensional. Therefore, to a certain extent, semantic analysis is a projection of the 3-D software behaviors into the 2-D semantic environment Θ as shown in Fig. 6.15.

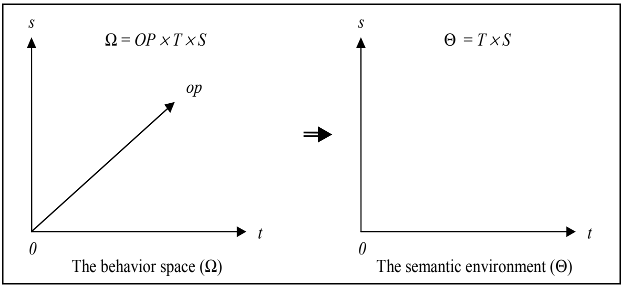


Figure 6.15 Relationship between software behavior space and the semantic environment

6.6 Semantics of RTPA

The theory of deductive semantics developed in Section 6.5.3 will be systematically applied to formally and rigorously describe the semantics of the RTPA *meta processes* and the *process relations* (operations) [Wang, 2002a/02b/03c/06a/07a]. This section extends the coverage of semantic rules of programming languages to a complete set of features that encompasses both basic computing operations and their algebraic composition rules. Because RTPA is a mathematical modeling language based on process algebra that covers a comprehensive set of computing and programming requirements as summarized in Section 4.5.1 and Theorem 6.3, any formal semantics that is capable to process RTPA is powerful enough to express the semantics of any programming language.

6.6.1 SEMANTICS OF RTPA META PROCESSES

Meta processes of RTPA are most fundamental computational operations elicited from basic computing requirements, based on them complex processes can be composed. RTPA identified 17 meta processes such as assignment, system control, event/time handling, memory, and I/O manipulation. The meta processes and their syntaxes have been given in Table 4.8 with detailed descriptions in Section 4.6.4 [Wang, 2002a].

As shown in Table 4.8, each meta process is a basic operation on one or more operands such as variables, memory elements, or I/O ports. Based on Definition 6.63, the deductive semantics of the set of RTPA meta processes will be defined in the following subsections, except that of the system process, §, which will be presented in Section 6.3.3.

6.6.1.1 The Assignment Process

Definition 6.69 The *semantics of the assignment process*, θ(y**RT** := x**RT**), in the given semantic environment Θ is a double partial differential of the semantic function f_θ(y**RT** := x**RT**) on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}
\theta(\mathrm{y}\mathbf{RT} := \mathrm{x}\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\mathrm{y}\mathbf{RT} := \mathrm{x}\mathbf{RT}) \\
&= \underset{i=0}{\overset{\#T(\mathrm{y}\mathbf{RT}\,:=\,\mathrm{x}\mathbf{RT})}{R}}\ \underset{j=1}{\overset{\#S(\mathrm{y}\mathbf{RT}\,:=\,\mathrm{x}\mathbf{RT})}{R}}\ v(t_i, s_j) \\
&= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{2}{R}}\ v(t_i, s_j) \\
&= \begin{pmatrix} & \mathrm{x}\mathbf{RT} & \mathrm{y}\mathbf{RT} \\ \mathbf{t_0} & x\mathbf{RT} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT} & x\mathbf{RT} \end{pmatrix}
\end{aligned} \qquad (6.54)$$

where the size of the matrix is #*T* • #*S*.

6.6.1.2 The Evaluation Process

Definition 6.70 The semantics of the *evaluation process*, θ(�*exp*$\mathbb{T}$ → $\mathbb{T}$), in the given semantic environment Θ is a double partial differential of the semantic function f_θ(θ(�*exp*$\mathbb{T}$ → $\mathbb{T}$) on the sets of variables *S* and executing steps *T* in the following two forms, i.e.:

$$\begin{aligned}
\theta(\text{�}(exp\mathbf{BL}) \rightarrow \mathbf{BL}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}(exp\mathbf{BL}) \rightarrow \mathbf{BL}) \\
&= \underset{i=0}{\overset{\#T(\text{�}exp\mathbf{BL}\rightarrow\mathbf{BL})}{R}}\ \underset{j=1}{\overset{\#S(\text{�}exp\mathbf{BL}\rightarrow\mathbf{BL})}{R}}\ v(t_i, s_j) \\
&= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{2}{R}}\ v(t_i, s_j) \\
&= \begin{pmatrix} & exp\mathbf{B} & \text{�}(exp\mathbf{BL})\mathbf{BL} \\ (\mathbf{t_0}, \mathbf{t_1}] & \delta(exp\mathbf{BL})\mathbf{BL} & \perp \\ (\mathbf{t}_1, \mathbf{t}_2] & \mathbf{T} & \mathbf{T} \\ (\mathbf{t}_1, \mathbf{t}_{2'}] & \mathbf{F} & \mathbf{F} \end{pmatrix}
\end{aligned} \qquad (6.55a)$$

or

$$\begin{aligned}
\theta(\text{�}exp\mathbb{T} \to \mathbb{T}) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{�}exp\mathbb{T} \to \mathbb{T}) \\
&= \mathop{R}_{i=0}^{\#T(\text{�}exp\mathbb{T}\to\mathbb{T})} \; \mathop{R}_{j=1}^{\#S(\text{�}exp\mathbb{T}\to\mathbb{T})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{2} v(t_i, s_j) \\
&= \begin{pmatrix} & exp\mathbb{T} & \text{�}(exp\mathbb{T})\mathbb{T} \\ (\mathbf{t}_0, \mathbf{t}_1] & \delta(exp\mathbb{T})\mathbb{T} & \perp \\ (\mathbf{t}_1, \mathbf{t}_2] & n\mathbb{T} & n\mathbb{T} \end{pmatrix}
\end{aligned} \tag{6.55b}$$

where $\text{�}_{\mathbb{T}}(exp\mathbf{BL})$ is the *Boolean* evaluation function on *exp***BL** that results in **T** or **F**.

In general, $\text{�}_{\mathbb{T}}(exp\mathbb{T})$ is the *cardinal or numerical* evaluation function on $exp\mathbb{T}$ that results in $\mathbb{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{B}\}$ as given in Definition 4.88a through 4.88e, respectively.

6.6.1.3 The Addressing Process

Definition 6.71 The semantics of the *addressing* process, $\theta(id\mathbf{S} \Rightarrow ptr\mathbf{P})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(id\mathbf{S} \;\Rightarrow ptr\mathbf{P})$ on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}
\theta(id\mathbf{S} \Rightarrow ptr\mathbf{P}) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\mathrm{id}\mathbf{S} \Rightarrow \mathrm{ptr}\mathbf{P}) \\
&= \mathop{R}_{i=0}^{\#T(\mathrm{id}\mathbf{S}\,\Rightarrow\,\mathrm{ptr}\mathbf{P})} \; \mathop{R}_{j=1}^{\#S(\mathrm{id}\mathbf{S}\,\Rightarrow\,\mathrm{ptr}\mathbf{P})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{2} v(t_i, s_j) \\
&= \begin{pmatrix} & \mathbf{idS} & \mathbf{ptrP} \\ \mathbf{t}_0 & id\mathbf{S} & \perp \\ (\mathbf{t}_0, \mathbf{t}_1] & id\mathbf{S} & \pi(id\mathbf{S})\mathbf{H} \end{pmatrix}
\end{aligned} \tag{6.56}$$

where $\pi(id\mathbf{S})\mathbf{H}$ is a function as given in Definition 4.41 that associates a declared identifier *id***S** to its hexadecimal memory address located by the pointed *ptr***P**.

6.6.1.4 The Memory Allocation Process

Definition 6.72 The semantics of the *memory allocation process*, $\theta(id\mathbf{S} \Leftarrow \text{MEM}[ptr\mathbf{P}]\mathbf{RT})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(id\mathbf{S} \Leftarrow \text{MEM}[ptr\mathbf{P}]\mathbf{RT})$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(\text{id}\mathbf{S} \Leftarrow \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{id}\mathbf{S} \Leftarrow \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT}) \\
&= \mathop{R}_{i=0}^{\#T(\text{id}\mathbf{S}\Leftarrow\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})} \ \mathop{R}_{j=1}^{\#S(\text{id}\mathbf{S}\Leftarrow\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{3} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{idS} & \mathbf{ptrP} & \mathbf{MEMRT} \\
\mathbf{t_0} & id\mathbf{S} & \perp & \perp \\
(\mathbf{t_0}, \mathbf{t_1}] & id\mathbf{S} & \pi(id\mathbf{S})\mathbf{H} & \text{MEM}[ptr\mathbf{P}]\mathbf{RT}
\end{array} \right)
\end{aligned}
\tag{6.57}
$$

where $\pi(id\mathbf{S})\mathbf{H}$ is a mapping function as given in Definition 4.41 that associates an identifier $id\mathbf{S}$ to a memory block starting at a hexadecimal address located by the pointed $ptr\mathbf{P}$. The ending address of the allocated memory block, $ptr\mathbf{P}$+size($\mathbf{RT}$)-1, is dependent on a machine implementation of the size of a given variable in type $\mathbf{RT}$.

6.6.1.5 The Memory Release Process

Definition 6.73 The semantics of the *memory release process*, $\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT}) \\
&= \mathop{R}_{i=0}^{\#T(id\mathbf{S}\nLeftarrow\text{MEM}[\perp]\mathbf{RT})} \ \mathop{R}_{j=1}^{\#S(id\mathbf{S}\nLeftarrow\text{MEM}[\perp]\mathbf{RT})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{3} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{idRT} & \mathbf{ptrP} & \mathbf{MEMRT} \\
\mathbf{t_0} & (id\mathbf{S}) & \pi(id\mathbf{S})\mathbf{H} & \text{MEM}(ptr\mathbf{P})\mathbf{RT} \\
(\mathbf{t_0}, \mathbf{t_1}] & \perp & \perp & \perp
\end{array} \right)
\end{aligned}
\tag{6.58}
$$

6.6.1.6 The Read Process

Definition 6.74 The semantics of the *read process*, $\theta(\mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT} \gtrdot x\mathbf{RT})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT} \gtrdot x\mathbf{RT})$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \mathrm{x}\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\;\partial s} f_\theta(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \mathrm{x}\mathbf{RT}) \\
&= \mathop{R}_{i=0}^{\#T(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \mathrm{x}\mathbf{RT})} \;\; \mathop{R}_{j=1}^{\#S(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \mathrm{x}\mathbf{RT})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{3} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{ptrP} & \mathbf{MEM(ptrP)RT} & \mathbf{xRT} \\
\mathbf{t_0} & ptr\mathbf{P} & \perp & \perp \\
\mathbf{(t_0, t_1]} & ptr\mathbf{P} & \mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT} & \mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT}
\end{array}\right)
\end{aligned} \tag{6.59}$$

6.6.1.7 The Write Process

Definition 6.75 The semantics of the *write process*, $\theta(\mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT} \lessdot x\mathbf{RT})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\mathrm{MEM}[ptr\mathbf{P}]\mathbf{RT} \lessdot x\mathbf{RT})$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \lessdot \mathrm{x}\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\;\partial s} f_\theta(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \lessdot \mathrm{x}\mathbf{RT}) \\
&= \mathop{R}_{i=0}^{\#T(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \lessdot \mathrm{x}\mathbf{RT})} \;\; \mathop{R}_{j=1}^{\#S(\mathrm{MEM}[\mathrm{ptr}\mathbf{P}]\mathbf{RT} \lessdot \mathrm{x}\mathbf{RT})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{3} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{xRT} & \mathbf{ptrP} & \mathbf{MEM[ptrP]RT} \\
\mathbf{t_0} & x\mathbf{RT} & \perp & \perp \\
\mathbf{(t_0, t_1]} & x\mathbf{RT} & ptr\mathbf{P} & x\mathbf{RT}
\end{array}\right)
\end{aligned} \tag{6.60}$$

6.6.1.8 The Input Process

Definition 6.76 The semantics of the *input* process, $\theta(\mathrm{PORT}[ptr\mathbf{P}]\mathbf{RT} \mid\!\gtrdot x\mathbf{RT})$, in the given semantic environment Θ is a double partial differential of the semantic function f_θ(PORT[*ptr*$\mathbf{P}$]$\mathbf{RT}$ $|\!>$ *x*$\mathbf{RT}$) on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}
\theta(\text{PORT[ptr}\mathbf{P}]\mathbf{RT}\,|\!>\text{x}\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{PORT[ptr}\mathbf{P}]\mathbf{RT}\,|\!>\text{x}\mathbf{RT}) \\
&= \mathop{R}\limits_{i=0}^{\#T(\text{PORT[ptr}\mathbf{P}]\mathbf{RT}|>\text{x}\mathbf{RT})}\ \mathop{R}\limits_{j=1}^{\#S(\text{PORT[ptr}\mathbf{P}]\mathbf{RT}|>\text{x}\mathbf{RT})} v(t_i,s_j) \\
&= \mathop{R}\limits_{i=0}^{1}\mathop{R}\limits_{j=1}^{3} v(t_i,s_j) \\
&= \left(\begin{array}{cccc}
 & \textbf{ptrP} & \textbf{PORT[ptrP]RT} & \textbf{xRT} \\
\mathbf{t_0} & ptr\mathbf{P} & \perp & \perp \\
(\mathbf{t_0},\mathbf{t_1}] & ptr\mathbf{P} & \text{PORT}[ptr\mathbf{P}]\mathbf{RT} & \text{PORT}[ptr\mathbf{P}]\mathbf{RT}
\end{array}\right)
\end{aligned} \tag{6.61}$$

6.6.1.9 The Output Process

Definition 6.77 The semantics of the *output process*, θ(*x*$\mathbf{RT}$ $|\!<$ PORT[*ptr*$\mathbf{P}$]$\mathbf{RT}$), in the given semantic environment *Θ* is a double partial differential of the semantic function f_θ(*x*$\mathbf{RT}$ $|\!<$ PORT[*ptr*$\mathbf{P}$]$\mathbf{RT}$) on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}
\theta(\text{x}\mathbf{RT}\,|\!<\text{PORT[ptr}\mathbf{P}]\mathbf{RT}) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{x}\mathbf{RT}\,|\!<\text{PORT[ptr}\mathbf{P}]\mathbf{RT}) \\
&= \mathop{R}\limits_{i=0}^{\#T(\text{x}\mathbf{RT}|<\text{PORT[ptr}\mathbf{P}]\mathbf{RT})}\ \mathop{R}\limits_{j=1}^{\#S(\text{x}\mathbf{RT}|<\text{PORT[ptr}\mathbf{P}]\mathbf{RT})} v(t_i,s_j) \\
&= \mathop{R}\limits_{i=0}^{1}\mathop{R}\limits_{j=1}^{3} v(t_i,s_j) \\
&= \left(\begin{array}{cccc}
 & \textbf{xRT} & \textbf{ptrP} & \textbf{PORT[ptrP]RT} \\
\mathbf{t_0} & x\mathbf{RT} & \perp & \perp \\
(\mathbf{t_0},\mathbf{t_1}] & x\mathbf{RT} & ptr\mathbf{P} & x\mathbf{RT}
\end{array}\right)
\end{aligned} \tag{6.62}$$

6.6.1.10 The Timing Process

Definition 6.78 The semantics of *the timing process*, θ(@*t*$\mathbf{TM}$ $\underline{\underline{@}}$ §*t*$\mathbf{TM}$), in the given semantic environment Θ is a double partial differential of the semantic function f_θ(@*t*$\mathbf{TM}$ $\underline{\underline{@}}$ §*t*$\mathbf{TM}$) on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{@}}\S\mathrm{t}\mathbf{TM}) &\triangleq \frac{\partial^2}{\partial t\;\partial s} f_\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{@}}\S\mathrm{t}\mathbf{TM}) \\
&= \mathop{R}_{i=0}^{\#T(@\,\mathrm{t}\mathbf{TM}\underline{\underline{@}}\S\mathrm{t}\mathbf{TM})}\;\mathop{R}_{j=1}^{\#S(@\,\mathrm{t}\mathbf{TM}\underline{\underline{@}}\S\mathrm{t}\mathbf{TM})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1}\mathop{R}_{j=1}^{2} v(t_i, s_j) \\
&= \begin{pmatrix} & \S\mathrm{t}\mathbf{TM} & @\mathrm{t}\mathbf{TM} \\ \mathbf{t_0} & \S\mathrm{t}\mathbf{TM} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & \S\mathrm{t}\mathbf{TM} & \S\mathrm{t}\mathbf{TM} \end{pmatrix}
\end{aligned}
\qquad (6.63)
$$

where **TM** represents the three timing types as shown in Table 4.7, i.e. **TM** = {**yy:MM:dd**, **hh:mm:ss:ms**, **yy:MM:dd:hh:mm:ss:ms**}.

6.6.1.11 The Duration Process

Definition 6.79 The semantics of the *duration process*, $\theta(@t\mathbf{TM} \underline{\underline{\Delta}}\ \S t\mathbf{TM}{+}\Delta d\mathbf{Z})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(@t\mathbf{TM} \underline{\underline{\Delta}}\ \S t\mathbf{TM}{+}\Delta d\mathbf{N})$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM}{+}\Delta \mathrm{d}\mathbf{Z}) &\triangleq \frac{\partial^2}{\partial t\;\partial s} f_\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM}{+}\Delta \mathrm{d}\mathbf{Z}) \\
&= \mathop{R}_{i=0}^{\#T(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM})}\;\mathop{R}_{j=1}^{\#S(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM})} v(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1}\mathop{R}_{j=1}^{3} v(t_i, s_j) \\
&= \begin{pmatrix} & \S\mathrm{t}\mathbf{TM} & \Delta \mathrm{d}\mathbf{N} & @\mathrm{t}\mathbf{TM} \\ \mathbf{t_0} & \S\mathrm{t}\mathbf{TM} & \Delta \mathrm{d}\mathbf{N} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & \S\mathrm{t}\mathbf{TM} & \Delta \mathrm{d}\mathbf{N} & \S\mathrm{t}\mathbf{TM} + \Delta d\mathbf{N} \end{pmatrix}
\end{aligned}
\qquad (6.64)
$$

where **TM** = {**yy:MM:dd**, **hh:mm:ss:ms**, **yy:MM:dd:hh:mm:ss:ms**}.

6.6.1.12 The Increase Process

Definition 6.80 The semantics of the *increase process*, $\theta(\uparrow(x\mathbf{RT}))$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\uparrow(x\mathbf{RT}))$ on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}\theta(\uparrow(\mathrm{x}\mathbf{RT})) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\uparrow(\mathrm{x}\mathbf{RT}))\\ &= \mathop{R}\limits_{i=0}^{\#T(\uparrow(\mathrm{x}\mathbf{RT}))}\ \mathop{R}\limits_{j=1}^{\#S(\uparrow(\mathrm{x}\mathbf{RT}))} v(t_i, s_j)\\ &= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{1} v(t_i, s_j)\\ &= \begin{pmatrix} & \mathbf{xRT}\\ \mathbf{t_0} & x\mathbf{RT}\\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT}+1\end{pmatrix}\end{aligned} \qquad (6.65)$$

where the run-time type $\mathbf{RT} = \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TM}\}$.

6.6.1.13 The Decrease Process

Definition 6.81 The semantics of the *decrease process*, $\theta(\downarrow(x\mathbf{RT}))$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\downarrow(x\mathbf{RT}))$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}\theta(\downarrow(\mathrm{x}\mathbf{RT})) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\downarrow(\mathrm{x}\mathbf{RT}))\\ &= \mathop{R}\limits_{i=0}^{\#T(\downarrow(\mathrm{x}\mathbf{RT}))}\ \mathop{R}\limits_{j=1}^{\#S(\downarrow(\mathrm{x}\mathbf{RT}))} v(t_i, s_j)\\ &= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{1} v(t_i, s_j)\\ &= \begin{pmatrix} & \mathbf{xRT}\\ \mathbf{t_0} & x\mathbf{RT}\\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT}\text{ - }1\end{pmatrix}\end{aligned} \qquad (6.66)$$

where the run-time type $\mathbf{RT} = \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TM}\}$.

6.6.1.14 The Exception Detection Process

Definition 6.82 The semantics of the *exception detection process*, $\theta(!(@)e\mathbf{S})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(!(@)e\mathbf{S}))$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(!(@e\mathbf{S}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(!(@e\mathbf{S})\\
&= \mathop{R}\limits_{i=0}^{\#T(!(@e\mathbf{S})}\ \mathop{R}\limits_{j=1}^{\#S(!(@e\mathbf{S})} v(t_i,s_j)\\
&= \mathop{R}\limits_{i=0}^{1}\mathop{R}\limits_{j=1}^{3} v(t_i,s_j)\\
&= \left(\begin{array}{cccc}
 & \mathbf{@eS} & \mathbf{ptrP} & \mathbf{PORT(ptrP)S}\\
\mathbf{t_0} & @e\mathbf{S} & \perp & \perp\\
(\mathbf{t_0},\mathbf{t_1}] & @e\mathbf{S} & ptr\mathbf{P} & @e\mathbf{S}
\end{array}\right)
\end{aligned} \tag{6.67}$$

Eq. 6.67 indicates that the semantics of exception detection is the output of a string $@e\mathbf{S}$ to a designated port PORT[$ptr\mathbf{P}$]$\mathbf{S}$, where the pointer $ptr\mathbf{P}$ points to a CRT or a printer. Therefore, the semantics of exception detection can be described based on the semantics of output as defined in Definition 6.77, i.e.:

$$\theta((!(@e\mathbf{S})) = \theta(@e\mathbf{S} \mid\!< \text{PORT}[ptr\mathbf{P}]\mathbf{S}) \tag{6.68}$$

6.6.1.15 The Skip Process

Definition 6.83 The semantics of the *skip process*, $\theta(\otimes)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\otimes)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(\otimes) &\triangleq \theta(P^k \curvearrowright P^{k-1})\\
&= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P^k \curvearrowright P^{k-1})\\
&= \mathop{R}\limits_{i=0}^{\#T(P^k \curvearrowright P^{k-1})}\ \mathop{R}\limits_{j=1}^{\#S(P^k \curvearrowright P^{k-1})} v(t_i,s_j)\\
&= \mathop{R}\limits_{i=0}^{1}\mathop{R}\limits_{j=1}^{2} v(t_i,s_j)\\
&= \left(\begin{array}{ccc}
 & \mathbf{S_{p^{k-1}}} & \mathbf{S_{p^k}}\\
\mathbf{t_0} & S_{P^{k-1}} & S_{P^k}\\
(\mathbf{t_0},\mathbf{t_1}] & S_{P^{k-1}} \setminus S_{P^k} & \perp
\end{array}\right)
\end{aligned} \tag{6.69}$$

where P^k is a process P at a given embedded layer k in a program with P^0 at the outermost layer, and $\curvearrowright$ denotes the jump process relation and its semantics will be formally defined in Section 6.6.2.2.

According to Definition 6.83, the skip process ⊗ has no semantic effect on the current process P^k at the given embedded layer k in a program, such as a branch, loop, or function. However, it do have semantic effect on internal control structures that redirects the system to jump to execute an upper-layer process P^{k-1} in the embedded hierarchy. Therefore, skip is also known as *exit* or *break* in programming languages.

6.6.1.16 The Stop Process

Definition 6.84 The semantics of the *stop process*, $\theta(\boxtimes)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\boxtimes)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(\boxtimes) &\triangleq \theta(P \curvearrowright \S) \\
&= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P \curvearrowright \S) \\
&= \mathop{R}\limits_{i=0}^{\#T(P\curvearrowright\S)} \; \mathop{R}\limits_{j=1}^{\#S(P\curvearrowright\S)} v(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{2} v(t_i, s_j) \\
&= \left(\begin{array}{ccc} & \mathbf{S}_\S & \mathbf{S_P} \\ \mathbf{t_0} & S_\S & S_P \\ (\mathbf{t_0}, \mathbf{t_1}] & S_\S \setminus S_P & \perp \end{array}\right)
\end{aligned} \tag{6.70}$$

where the stop process ☒ does nothing but returns the control of execution to the system. The semantics of jump, $\curvearrowright$, will be formally described in Section 6.6.2.2.

6.6.2 SEMANTICS OF RTPA PROCESS RELATIONS

Section 6.6.1 presented formal definitions of the meta processes of RTPA for software system modeling. According to Theorem 6.3, via the composition of multiple meta processes by the 17 process relations, complex architectures and behaviors of software systems, in the most complicated case, a real-time system, can be sufficiently described [Wang, 2002a].

Detailed descriptions of syntaxes of RTPA process relations have been described in Table 4.9. On the basis of Definition 6.65, the semantics of the RTPA process relations will be formally defined and analyzed in the following subsections, except that of the three process dispatch relations which will be presented in Section 6.6.3.

6.6.2.1 The Sequential Process Relation

Definition 6.85 The semantics of the *sequential relations* of processes, $\theta(P \to Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \to Q)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(P \to Q) &\triangleq \frac{\partial^2}{\partial t\, \partial s} f_\theta(P \to Q) \\
&= \frac{\partial^2}{\partial t\, \partial s} f_\theta(P) \to \frac{\partial^2}{\partial t\, \partial s} f_\theta(Q) \\
&= \mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j) \to \mathop{R}_{i=0}^{\#T(Q)} \mathop{R}_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}_{i=0}^{\#T(\overset{\frown}{PQ}} \mathop{R}_{j=1}^{\#S(P \cup Q)} v(t_i, s_j) \\
&= \left(\begin{array}{cccc} & \mathbf{s_P} & \mathbf{s_Q} & \mathbf{s_{PQ}} \\ \mathbf{t_0} & \perp & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & - & V_{1PQ} \\ (\mathbf{t_1}, \mathbf{t_2}] & - & V_{2Q} & V_{2PQ} \end{array} \right) \\
&= \begin{pmatrix} \mathbf{V_P} & & \mathbf{V_{PQ}} \\ & \mathbf{V_Q} & \mathbf{V_{PQ}} \end{pmatrix}
\end{aligned}
\tag{6.71}
$$

where $\overset{\frown}{PQ}$ indicates a concatenation of these two processes over time, and in the simplified notation of the matrix, $V_P = v(t_P, s_P)$, $0 \le t_P \le n_P$, $1 \le s_P \le m_P;$ $V_Q = v(t_Q, s_Q)$, $0 \le t_Q \le n_Q$, $1 \le s_Q \le m_Q;$ and $V_{PQ} = v(t_{PQ}, s_{PQ})$, $0 \le t_{PQ} \le n_{PQ}$, $1 \le s_{PQ} \le m_{PQ}$.

In Eq. 6.71, the first partial differential selects a set of related variables in the sequential processes *P* and *Q*, $S(P \cup Q)$. The second partial differential selects a set of time moments $T(\overset{\frown}{PQ})$. The semantic diagram of the sequential process relation as defined in Eq. 6.71 is illustrated in Fig. 6.16 in the semantic environment Θ .

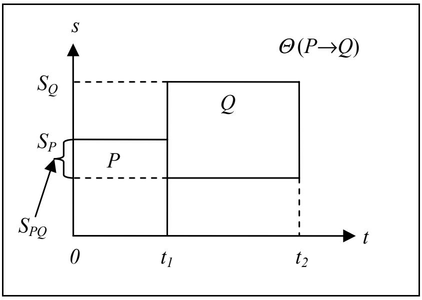


Figure 6.16 The semantic diagram of the sequential process relation

The following example shows the physical meaning of Eq. 6.71 and how the abstract syntaxes and their implied meanings are embodied onto the objects (variables) and their dynamic values in order to obtain the concrete semantics in deductive semantics.

Example 6.26 Analyze the semantics of two simple sequential processes *P* and *Q* in the following program:

```
void sequential_sum;
      {
        (0)      int x, y, z;
       {// P
        (1)      x = 2;
        (2)      y = 8;
        (3)      z := x + y;
       }
       {// Q
        (4)      z := x + y + z;
        (5)      print z;
       }
      }
```

According to Definition 6.85, the semantics of the above program can be analyzed as follows:

$$\theta(P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_5) = \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_5)$$
$$= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_0) \rightarrow \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_1) \rightarrow \ldots \rightarrow \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_5)$$

$$
\begin{aligned}
&= \mathop{R}\limits_{i=0}^{\#T(P_0)} \mathop{R}\limits_{j=1}^{\#S(P_0)} v_{P_0}(t_i, s_j) \to \mathop{R}\limits_{i=0}^{\#T(P_1)} \mathop{R}\limits_{j=1}^{\#S(P_1)} v_{P_1}(t_i, s_j) \to \ldots \\
&\quad \to \mathop{R}\limits_{i=0}^{\#T(P_5)} \mathop{R}\limits_{j=1}^{\#S(P_5)} v_{P_5}(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{\#T(\widehat{P_0\ P_1\ \ldots P_5})} \quad \mathop{R}\limits_{j=1}^{\#S(P_0 \cup P_1\ \cup \ldots \cup P_5)} \quad v(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{5} \mathop{R}\limits_{j=1}^{4} v(t_i, s_j) \\
&= \left(\begin{array}{c|cccc}
 & \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{PORT[CRTP]N} \\
\mathbf{t_0} & \perp & \perp & \perp & \perp \\
(\mathbf{t_0}, \mathbf{t_1}] & 2 & \perp & \perp & \perp \\
(\mathbf{t_1}, \mathbf{t_2}] & 2 & 8 & \perp & \perp \\
(\mathbf{t_2}, \mathbf{t_3}] & 2 & 8 & 10 & \perp \\
(\mathbf{t_3}, \mathbf{t_4}] & 2 & 8 & 20 & \perp \\
(\mathbf{t_4}, \mathbf{t_5}] & 2 & 8 & 20 & 20
\end{array} \right)
\end{aligned}
\tag{6.72}
$$

where PORT[*CRT***P**]**N** denotes a system monitor of type **N** located by the pointer *CRT***P**; the semantics of *P* and *Q* are shown in the intervals [t$_0$, t$_3$] and (t$_3$, t$_5$], respectively.

6.6.2.2 The Jump Process Relation

Definition 6.86 The semantics of the *jump relations* of processes, $\theta(P \curvearrowright Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \curvearrowright Q)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(P \curvearrowright Q) &\triangleq \frac{\partial^2}{\partial t\, \partial s} f_\theta(P \curvearrowright Q) \\
&= \frac{\partial^2}{\partial t\, \partial s} f_\theta(P) \curvearrowright \frac{\partial^2}{\partial t\, \partial s} f_\theta(Q) \\
&= \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j) \curvearrowright \mathop{R}\limits_{i=0}^{\#T(Q)} \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{\#T(\widehat{PQ})} \mathop{R}\limits_{j=1}^{\#S(P \cup Q)} v(t_i, s_j)
\end{aligned}
$$

$$= \left(\begin{array}{lcccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{addrH} \\ [\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & \perp & V_{1PQ} & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & - & - & - & \pi(Q\mathbf{S})\mathbf{H} \\ (\mathbf{t_2}, \mathbf{t_3}] & - & V_{3Q} & V_{3PQ} & \end{array} \right) \qquad (6.73)$$

where $\pi(Q\mathbf{S})\mathbf{H}$ is a system addressing function of the system that directs the program control flow to execute the new process Q, which is physically located in a different memory address at $addr\mathbf{H} = \pi(Q\mathbf{S})\mathbf{H}$.

The semantic diagram of the jump process relation as defined in Eq. 6.73 is illustrated in Fig. 6.17 in the semantic environment Θ.

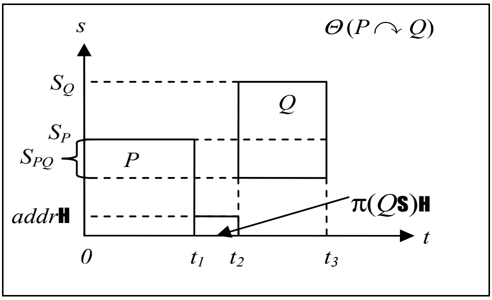


Figure 6.17 The semantic diagram of the jump process relation

The jump process relation is an important process relation that forms a fundamental part of many other processes and constructs. For instance, the jump process relation has been applied in expressing the semantics of the *skip* and *stop* processes in Section 4.1.

6.6.2.3 The Branch Process Relation

Definition 6.87 The semantics of the *branch relations* of processes, θ(�*exp*BL = T → *P* | �~ → *Q*), abbreviated by $\theta(P \mid Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \mid Q)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(\text{�}exp\textbf{RT} \to P \mid \text{�}\sim \to Q) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}exp\textbf{RT} \to P \mid \text{�}\sim \to Q)\\
&= \text{�}exp\textbf{BL} \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P)\\
&\quad \mid \text{�}\sim \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)\\
&= \text{�}exp\textbf{BL} \to \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j)\\
&\quad \mid \text{�}\sim \to \mathop{R}\limits_{i=0}^{\#T(Q)} \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j)\\
&= \left(\begin{array}{ccccc}
 & \textbf{exp}\textbf{BL} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}}\\
(\mathbf{t_0}, \mathbf{t_1}] & \delta(exp\textbf{BL}) & \perp & \perp & \perp\\
(\mathbf{t_1}, \mathbf{t_2}] & \textbf{T} & V_{2P} & - & V_{2PQ}\\
(\mathbf{t_1}, \mathbf{t_{2'}}] & \textbf{F} & - & V_{3Q} & V_{3PQ}
\end{array}\right)
\end{aligned}
\tag{6.74}
$$

where δ(*exp***BL**) is the evaluation function on the value of *exp***BL**, δ(*exp***BL**) ∈ {**T**, **F**}.

The semantic diagram of the branch process relation as defined in Eq. 6.74 is illustrated in Fig. 6.18 in the semantic environment Θ(*θ*(�*exp***BL** = **T** → *P* | �~ → *Q*).

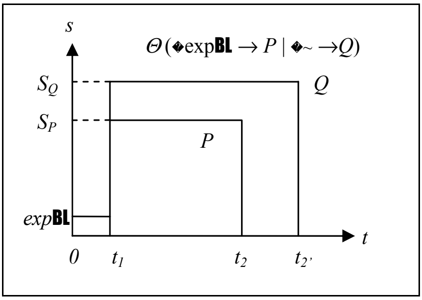


Figure 6.18 The semantic diagram of the branch process relation

6.6.2.4 The Switch Process Relation

Definition 6.88 The semantics of the *switch relations* of processes, *θ*(�*exp*$_i$**RT** → P$_\text{i}$ | �~ → ∅), abbreviated by *θ*(P$_\text{i}$ | ∅), in the given semantic environment Θ is a double partial differential of the semantic function *f*$_\theta$(P$_\text{i}$ | ∅) on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(\text{�}\mathit{exp}_{\mathrm{i}}\mathbf{RT} \to \mathrm{P}_{\mathrm{i}} \mid \text{�}\sim \to \otimes) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}\mathit{exp}_{\mathrm{i}}\mathbf{RT} \to \mathrm{P}_{\mathrm{i}} \mid \text{�}\sim \to \otimes) \\
&= \text{�}\mathit{exp}\mathbf{RT} = \mathit{0} \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_{\mathit{0}}) \\
&\quad \mid \ldots \\
&\quad \mid \mathit{exp}\mathbf{RT} = n-1 \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_{n-1}) \\
&\quad \mid \mathit{exp}\mathbf{RT} = n \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(\otimes) \\
&= \text{�}\mathit{exp}\mathbf{RT} = \mathit{0} \to \mathop{R}_{i=0}^{\#T(P_{\mathit{0}})} \mathop{R}_{j=1}^{\#S(P_{\mathit{0}})} v_{P_0}(t_i, s_j) \\
&\quad \mid \ldots \\
&\quad \mid \mathit{exp}\mathbf{RT} = n-1 \to \mathop{R}_{i=0}^{\#T(P_{n-1})} \mathop{R}_{j=1}^{\#S(P_{n-1})} v_{P_{n-1}}(t_i, s_j) \\
&\quad \mid \mathit{exp}\mathbf{RT} = n \to \varnothing \\
&= \left(\begin{array}{cccccc}
 & \mathbf{exp}\mathbf{RT} & \mathbf{S}_{\mathbf{P_0}} & \cdots & \mathbf{S}_{\mathbf{P_{n\text{-}1}}} & \mathbf{S}_{\mathbf{G}} \\
[\mathbf{t}_0, \mathbf{t}_1] & \delta(\mathrm{exp}\mathbf{RT}) & \perp & \cdots & \perp & \perp \\
(\mathbf{t}_1, \mathbf{t}_{\mathbf{2_0}}] & \mathit{0} & V_{2_{\mathit{0}} P} & \cdots & - & V_G \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
(\mathbf{t}_1, \mathbf{t}_{\mathbf{2_{n\text{-}1}}}] & n-1 & - & \cdots & V_{2_{n-1} P} & V_G \\
(\mathbf{t}_1, \mathbf{t}_{\mathbf{2_n}}] & n & - & \cdots & - & V_G
\end{array}\right)
\end{aligned}
\tag{6.75}
$$

where V_G is a set of global variables shared by P_0, P_1, and $P_{n\text{-}1}$.

The semantic diagram of the switch process relation as defined in Eq. 6.75 is illustrated in Fig. 6.19 in the semantic environment $\Theta($�$\mathit{exp}_i\mathbf{RT} \to \mathrm{P}_{\mathrm{i}} \mid$ �$\sim \to \varnothing)$.

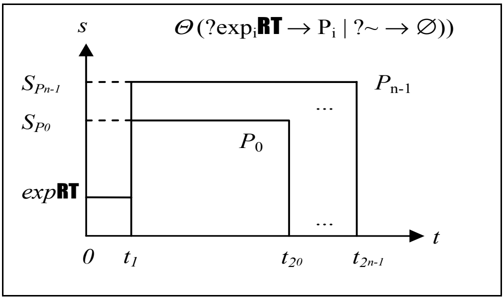


Figure 6.19 The semantic diagram of the switch process relation

6.6.2.5 The While-Loop Process Relation

Definition 6.89 The semantics of the *while-loop* relations of processes, $\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(P))$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(P))$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(P)) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(P)) \\
&= \overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(\frac{\partial^2}{\partial t\ \partial s} f_\theta(P)) \\
&= \overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(\overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v_P(t_i, s_j)) \\
&= \left(\begin{array}{ccc}
 & \exp\mathbf{BL} & \mathbf{S_P} \\
[\mathbf{t_0},\mathbf{t_1}] & \delta(\exp\mathbf{BL}) & \perp \\
(\mathbf{t_1},\mathbf{t_2}] & \mathbf{T} & V_P \\
(\mathbf{t_1},\mathbf{t_{2'}}] & \mathbf{F} & \otimes \\
\vdots & \vdots & \vdots \\
(\mathbf{t_3},\mathbf{t_4}] & \delta(\exp\mathbf{BL}) & - \\
(\mathbf{t_4},\mathbf{t_5}] & \mathbf{T} & V_P \\
(\mathbf{t_4},\mathbf{t_{5'}}] & \mathbf{F} & \otimes
\end{array}\right)
\end{aligned}
\tag{6.76}
$$

where ∅ denotes exit, and δ(*exp***BL**) is the evaluation function on the Boolean expression, δ(*exp***BL**) ∈ {**T**, **F**}.

The semantic diagram of the while-loop process relation as defined in Eq. 6.76 is illustrated in Fig. 6.20 in the semantic environment Θ.

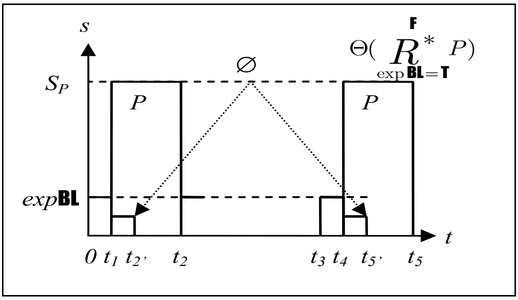

s
$\Theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*} P)$
∅
S_P
P
P
*exp***BL**
t
0 t_1 $t_{2'}$ t_2 t_3 t_4 $t_{5'}$ t_5

Figure 6.20 The semantic diagram of the while-loop process relation

6.6.2.6 The Repeat-Loop Process Relation

Definition 6.90 The semantics of the *repeat-loop* relations of processes, $\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{+}(P))$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{+}(P))$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{+}(P)) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{+}(P)) \\
&= \overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{+}(\frac{\partial^2}{\partial t\ \partial s} f_\theta(P)) \\
&= P \rightarrow \overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}(\mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j)) \\
&= \left(\begin{array}{ccc}
 & \exp\mathbf{BL} & \mathbf{S_P} \\
[\mathbf{t_0},\mathbf{t_1}] & \perp & V_P \\
(\mathbf{t_1},\mathbf{t_2}] & \delta(\exp\mathbf{BL}) & - \\
(\mathbf{t_2},\mathbf{t_3}] & \mathbf{T} & V_P \\
(\mathbf{t_2},\mathbf{t_{3'}}] & \mathbf{F} & \otimes \\
\vdots & \vdots & \vdots \\
(\mathbf{t_4},\mathbf{t_5}] & \delta(\exp\mathbf{BL}) & - \\
(\mathbf{t_5},\mathbf{t_6}] & \mathbf{T} & V_P \\
(\mathbf{t_5},\mathbf{t_{6'}}] & \mathbf{F} & \otimes
\end{array}\right)
\end{aligned}
\qquad (6.77)
$$

The semantic diagram of the repeat-loop process relation as defined in Eq. 6.77 is illustrated in Fig. 6.21 in the semantic environment Θ.

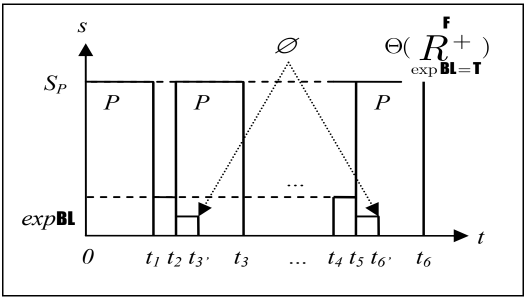


Figure 6.21 The semantic diagram of the repeat-loop process relation

6.6.2.7 The For-Loop Process Relation

Definition 6.91 The semantics of the *for-loop* relations of processes, $\theta(\mathop{R}\limits_{i\mathbf{N}=1}^{n} \mathrm{P(i)})$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(\mathop{R}\limits_{i\mathbf{N}=1}^{n} \mathrm{P(i)})$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(\mathop{R}\limits_{i\mathbf{N}=1}^{n}\mathrm{P(i)}) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\mathop{R}\limits_{i\mathbf{N}=1}^{n}\mathrm{P(i)}) \\
&= \mathop{R}\limits_{k\mathbf{N}=1}^{n} (\frac{\partial^2}{\partial t\ \partial s} f_\theta(P_i)) \\
&= \mathop{R}\limits_{k\mathbf{N}=1}^{n} (\mathop{R}\limits_{i=0}^{\#T(P_k)} \ \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)) \qquad (6.78) \\
&= \left(\begin{array}{ccc}
 & k\mathbf{N} & \mathbf{S_P} \\
[\mathbf{t_0}, \mathbf{t_1}] & 1 & \perp \\
(\mathbf{t_1}, \mathbf{t_2}] & 1 & V_P \\
\vdots & \vdots & \vdots \\
(\mathbf{t_{n-2}}, \mathbf{t_{n-1}}] & n & - \\
(\mathbf{t_{n-1}}, \mathbf{t_n}] & n & V_P
\end{array} \right)
\end{aligned}
$$

The semantic diagram of the for-loop process relation as defined in Eq. 6.78 is illustrated in Fig. 6.22 in the semantic environment Θ.

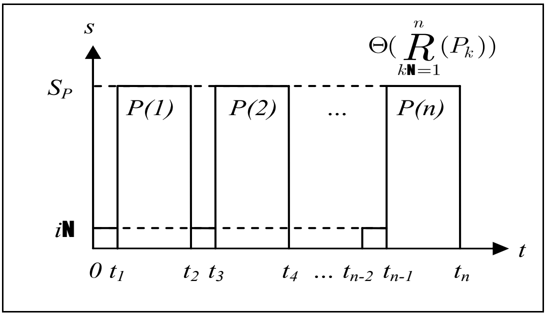


Figure 6.22 The semantic diagram of the for-loop process relation

6.6.2.8 The Function Call Process Relation

Definition 6.92 The semantics of the *function call relations* of processes, $\theta(P \rightarrowtail Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \rightarrowtail Q)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(P \rightarrowtail Q) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \rightarrowtail Q) \\
&= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \ \rightarrowtail\ \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q) \\
&= \underset{i=0}{\overset{\#T(P)}{R}}\ \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j) \ \rightarrowtail\ \underset{i=0}{\overset{\#T(Q)}{R}}\ \underset{j=1}{\overset{\#S(Q)}{R}} v_Q(t_i, s_j) \\
&= \underset{i=0}{\overset{\#T([t_0,t_1] \frown (t_1,t_2] \frown (t_2,t_3])}{R}}\ \underset{j=1}{\overset{\#S(P \cup Q)}{R}} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\
\mathbf{t_0} & \perp & \perp & \perp \\
\mathbf{(t_0,t_1]} & V_{1P} & - & V_{1PQ} \\
\mathbf{(t_1,t_2]} & - & v_{2Q} & V_{2PQ} \\
\mathbf{(t_2,t_3]} & V_{3P} & - & V_{3PQ}
\end{array} \right)
\end{aligned} \tag{6.79}$$

The semantic diagram of the procedure call process relation as defined in Eq. 6.79 is illustrated in Fig. 6.23 in the semantic environment Θ.

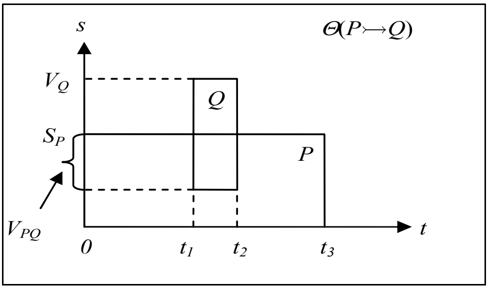


Figure 6.23 The semantic diagram of the function call process relation

6.6.2.9 The Recursive Process Relation

Definition 6.93 The semantics of the *recursive relations* of processes, $\theta(P \circlearrowleft P)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \circlearrowleft P)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(P \circlearrowleft P) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \circlearrowleft P) \\
&= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \circlearrowleft \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \\
&= \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v(t_i, s_j) \circlearrowleft \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v(t_i, s_j) \\
&= \left(\begin{array}{cc}
 & \mathbf{S_P} \\
[\mathbf{t_0}, \mathbf{t_1}] & V_{P^n} \\
(\mathbf{t_1}, \mathbf{t_2}] & V_{P^{n-1}} \\
\vdots & \vdots \\
(\mathbf{t_3}, \mathbf{t_4}] & V_{P^0} \\
\vdots & \vdots \\
(\mathbf{t_5}, \mathbf{t_6}] & V_{P'^{n-1}} \\
(\mathbf{t_6}, \mathbf{t_7}] & V_{P'^n}
\end{array} \right)
\end{aligned} \tag{6.80}$$

The semantic diagram of the recursive process relation as defined in Eq. 6.80 is illustrated in Fig. 6.24 in the semantic environment Θ.

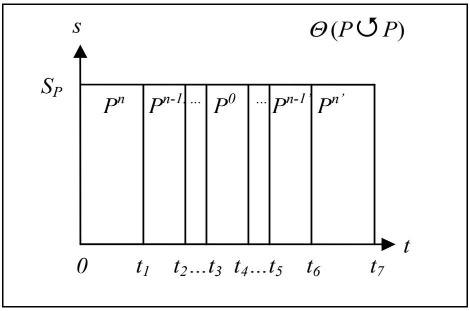


Figure 6.24 The semantic diagram of the recursive process relation

6.6.2.10 The Parallel Process Relation

Definition 6.94 The semantics of the *parallel relations* of processes, $\theta(P \parallel Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \parallel Q)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(P \parallel Q) &\triangleq \frac{\partial^2}{\partial t\, \partial s} f_\theta(P \parallel Q) \\
&= \frac{\partial^2}{\partial t\, \partial s} f_\theta(P) \parallel \frac{\partial^2}{\partial t\, \partial s} f_\theta(Q) \\
&= \mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j) \parallel \mathop{R}_{i=0}^{\#T(Q)} \mathop{R}_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}_{i=0}^{\max(\#T(P),\#T(Q))} \mathop{R}_{j=1}^{\#S(P \cup Q)} v(t,s) \\
&= \left(\begin{array}{cccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\ \mathbf{t_0} & V_{0P} & V_{0Q} & V_{0PQ} \\ (\mathbf{t_0},\mathbf{t_1}] & V_{1P} & V_{1Q} & V_{1PQ} \\ (\mathbf{t_1},\mathbf{t_2}] & - & V_{2Q} & V_{2PQ} \end{array} \right)
\end{aligned} \tag{6.81}$$

where $t_2 = max(\#T(P), (\#T(Q))$ is the synchronization point between two parallel processes.

The semantic diagram of the parallel process relation as defined in Eq. 6.81 is illustrated in Fig. 6.25 in the semantic environment Θ.

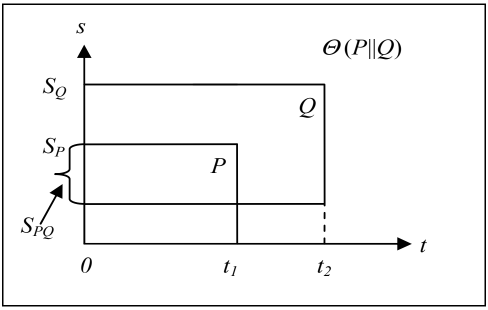


Figure 6.25 The semantic diagram of the parallel process relation

It is noteworthy that parallel processes *P* and *Q* are interlocked. That is, they should start and end at the same time. In case $t_1 \neq t_2$, the process completed earlier should wait for the completion of the other. The second condition between parallel processes is that the shared resources, in particular variables, memory space, ports, and devices, should be protected. That is, when a process operates on a shared resource, it is locked to the other process until the operation is completed. A variety of interlocking and synchronization techniques, such as *semaphores, mutual exclusions,* and *critical regions*, have been proposed in real-time system techniques [Liu, 2000; McDermid, 1991].

6.6.2.11 The Concurrent Process Relation

Definition 6.95 The semantics of the *concurrent relations* of processes, $\theta(P \oiint Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \oiint Q)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(P \oiint Q) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(P \oiint Q) \\
&= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P) \oiint \frac{\partial^2}{\partial t\,\partial s} f_\theta(Q) \\
&= \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j) \oiint \mathop{R}\limits_{i=0}^{\#T(Q)} \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}\limits_{i=0}^{\max(\#T(P),\#T(Q))} \mathop{R}\limits_{j=1}^{\#S(P \cup Q)} v(t_i, s_j) \\
&= \left(\begin{array}{ccccc}
 & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{comRT} \\
\mathbf{t_0} & V_{0P} & V_{0P} & V_{0P} & V_{0com} \\
\mathbf{(t_0,t_1]} & V_{1P} & - & V_{1PQ} & V_{1com} \\
\mathbf{(t_1,t_2]} & V_{2P} & V_{2Q} & V_{2PQ} & V_{2com} \\
\mathbf{(t_2,t_3]} & V_{3P} & - & V_{3PQ} & V_{3com}
\end{array} \right)
\end{aligned}
\qquad (6.82)
$$

where *com***RT** is a set of inter-process communication variables that are used to synchronize *P* and *Q* executing on different machines based on independent system clocks.

The semantic diagram of the concurrent process relation as defined in Eq. 6.82 is illustrated in Fig. 6.26 in the semantic environment Θ.

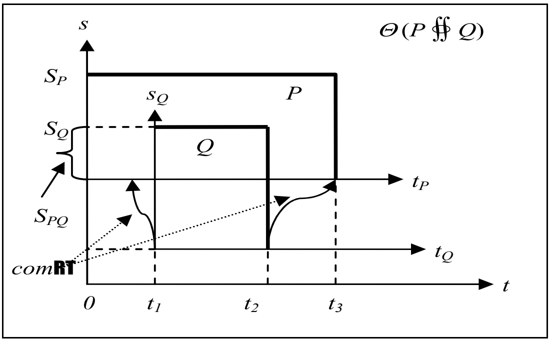


Figure 6.26 The semantic diagram of the concurrent process relation

6.6.2.12 The Interleave Process Relation

Definition 6.96 The semantics of the *interleave relations* of processes, $\theta(P \,|||\, Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \,|||\, Q)$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(P \,|||\, Q) &\triangleq \frac{\partial^2}{\partial t\, \partial s} f_\theta(P \,|||\, Q) \\
&= \frac{\partial^2}{\partial t\, \partial s} f_\theta(P) \,|||\, \frac{\partial^2}{\partial t\, \partial s} f_\theta(Q) \\
&= \mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j) \,|||\, \mathop{R}_{i=0}^{\#T(Q)} \mathop{R}_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}_{i=0}^{\#T([\mathrm{t}_0,\mathrm{t}_1]\frown(\mathrm{t}_1,\mathrm{t}_2]\frown(\mathrm{t}_2,\mathrm{t}_3]\frown(\mathrm{t}_3,\mathrm{t}_4]\frown(\mathrm{t}_4,\mathrm{t}_5])} \mathop{R}_{j=1}^{\#S(P\cup Q)} v(t_i, s_j) \\
&= \left(\begin{array}{cccc}
 & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\
\mathbf{t_0} & V_{0P} & V_{0Q} & V_{0PQ} \\
\mathbf{(t_0, t_1]} & V_{1P'} & - & V_{1PQ} \\
\mathbf{(t_1, t_2]} & - & V_{2Q'} & V_{2PQ} \\
\mathbf{(t_2, t_3]} & V_{3P''} & - & V_{3PQ} \\
\mathbf{(t_3, t_4]} & - & V_{4Q''} & V_{4PQ} \\
\mathbf{(t_4, t_5]} & V_{5P'''} & - & V_{5PQ}
\end{array} \right)
\end{aligned}
\qquad (6.83)
$$

The semantic diagram of the interleave process relation as defined in Eq. 6.83 is illustrated in Fig. 6.27 in the semantic environment Θ.

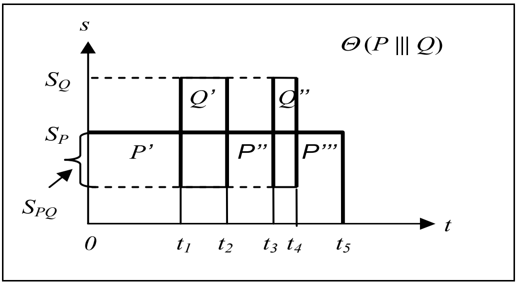


Figure 6.27 The semantic diagram of the interleave process relation

6.6.2.13 The Pipeline Process Relation

Definition 6.97 The semantics of the *pipeline relations* of processes, $\theta(P >> Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P >> Q)$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(P \gg Q) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \gg Q) \\
&= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \gg \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q) \\
&= \mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j) \gg \mathop{R}_{i=0}^{\#T(Q)} \mathop{R}_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}_{i=0}^{\#T(\widehat{PQ})} \mathop{R}_{j=1}^{\#S(P \cup Q)} v(t_i, s_j) \\
&= \left(\begin{array}{cccc} & \mathbf{S_P} & \mathbf{S_{P_o}} = \mathbf{S_{Q_i}} & \mathbf{S_Q} \\ \mathbf{t_0} & V_{0P} & V_{0PQ} & V_{0Q} \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & V_{1PQ} & - \\ (\mathbf{t_1}, \mathbf{t_2}] & - & V_{2PQ} & V_{2Q} \end{array} \right)
\end{aligned}
\qquad (6.84)
$$

where $\mathbf{S_{P_o}}$ and $\mathbf{S_{Q_i}}$ denote a set of n one-to-one connections between the outputs of P and inputs of Q, respectively, as follows:

$$
\mathop{R}_{k=0}^{n-1} (P_o(i) = Q_i(i)) \qquad (6.85)
$$

The semantic diagram of the pipeline process relation as defined in Eq. 6.84 is illustrated in Fig. 6.28 in the semantic environment Θ.

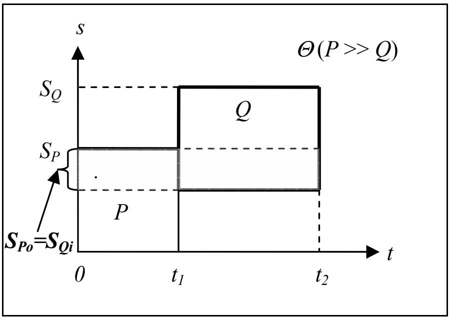


Figure 6.28 The semantic diagram of the pipeline process relation

6.6.2.14 The Interrupt Process Relation

Definition 6.98 The semantics of the *interrupt relations* of processes, $\theta(P \lightning Q)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P \lightning Q)$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(P \lightning Q) &\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(P \lightning Q) \\
&= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P) \lightning \frac{\partial^2}{\partial t\,\partial s} f_\theta(Q) \\
&= \mathop{R}_{i=0}^{\#T(P)} \mathop{R}_{j=1}^{\#S(P)} v_P(t_i, s_j) \lightning \mathop{R}_{i=0}^{\#T(Q)} \mathop{R}_{j=1}^{\#S(Q)} v_Q(t_i, s_j) \\
&= \mathop{R}_{i=0}^{\#T(P' {}^\frown Q {}^\frown P'')} \mathop{R}_{j=1}^{\#S(P \cup Q)} v(t_i, s_j) \\
&= \left(\begin{array}{ccccc}
 & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{int}\odot \\
[\mathbf{t_0}, \mathbf{t_1}] & V_{1P'} & \perp & V_{1PQ} & \perp \\
(\mathbf{t_1}, \mathbf{t_2}] & - & - & V_{2PQ} & V_{2\mathbf{int}\odot} \\
(\mathbf{t_2}, \mathbf{t_3}] & - & V_{3Q} & V_{3PQ} & - \\
(\mathbf{t_3}, \mathbf{t_4}] & - & - & V_{4PQ} & V_{4\mathbf{int}'\odot} \\
(\mathbf{t_4}, \mathbf{t_5}] & V_{5P''} & - & V_{5PQ} & -
\end{array} \right)
\end{aligned}
\qquad (6.86)
$$

The semantic diagram of the interrupt process relation as defined in Eq. 6.86 is illustrated in Fig. 6.29 in the semantic environment Θ. In Fig. 6.29, $C(\mathbf{int}\odot)$ and $C'(\mathbf{int}'\odot)$ are the interrupt and interrupt-return points, respectively.

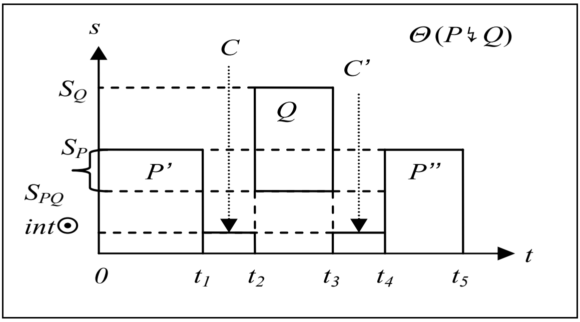


Figure 6.29 The semantic diagram of the interrupt process relation

6.6.3 SEMANTICS OF SYSTEM AND SYSTEM PROCESS DISPATCHING

The generic mathematical model of program systems has been modeled in Theorem 5.7 in Section 5.5.1. According to Theorem 5.7 and Definitions 6.63 and 6.65, the semantics of system at the top level of a program can be deduced onto a dispatch mechanism of a finite set of processes based on time-, event-, and interrupt-dispatching.

6.6.3.1 The System Process

Definition 6.99 The semantics of the *system process* in RTPA, §, is an abstract logical model of the executing platform with a set of parallel dispatched processes based on internal system clock, external events, and system interrupts, i.e.:

$$
\begin{aligned}
\theta(\S) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\S) \\
&= \frac{\partial^2}{\partial t\ \partial s} f_\theta\{ \mathop{R}_{i\mathbf{N}=0}^{n_e\mathbf{N}-1} (@e_i\mathbf{S} \hookrightarrow P_i) \\
&\qquad\qquad\qquad || \mathop{R}_{j\mathbf{N}=0}^{n_t\mathbf{N}-1} (@t_j\mathbf{TM} \hookrightarrow P_j) \\
&\qquad\qquad\qquad || \mathop{R}_{k\mathbf{N}=0}^{n_{\text{int}}\mathbf{N}-1} (@int_k\mathbf{S} \hookrightarrow P_k) \\
&\qquad\qquad\qquad \}
\end{aligned}
$$

$$\begin{aligned} = \underset{SysShuntDown\textbf{BL}=\textbf{F}}{\overset{\textbf{T}}{R}} \{ & \underset{i\textbf{N}=0}{\overset{n_e\textbf{N}-1}{R}}(@e_i\textbf{S} \hookrightarrow P_i) \\ & || \underset{j\textbf{N}=0}{\overset{n_t\textbf{N}-1}{R}}(@t_j\textbf{TM} \hookrightarrow P_j) \\ & || \underset{k\textbf{N}=0}{\overset{n_{\text{int}}\textbf{N}-1}{R}}(@int_k\textbf{S} \hookrightarrow P_k) \\ & \} \end{aligned} \tag{6.87}$$

where the semantics of the parallel relations has been given in Definition 6.94, and those of the system dispatch processes will be described in the following subsections.

6.6.3.2 The Time-Driven Dispatching Process Relation

Definition 6.100 The semantics of the *time-driven dispatching relations* of processes, $\theta(@t_k\textbf{TM} \hookrightarrow_t \text{P}_k)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(@t_k\textbf{TM} \hookrightarrow_t \text{P}_k)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned} \theta(@t_k\textbf{TM} \hookrightarrow_k \text{P}_k) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@t_k\textbf{TM} \hookrightarrow_k \text{P}_k) \\ &= \underset{k=1}{\overset{n}{R}}(@t_k\textbf{TM} \rightarrow \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_k)) \\ &= \underset{k=1}{\overset{n}{R}}(@t_k\textbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_k)}{R}}\ \underset{j=1}{\overset{\#S(P_k)}{R}} v_{P_k}(t_i, s_j)) \\ &= \ @t_1\textbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_1)}{R}}\ \underset{j=1}{\overset{\#S(P_1)}{R}} v_{P_1}(t_i, s_j) \\ &\quad\ | \ldots \\ &\quad\ | @t_n\textbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_n)}{R}}\ \underset{j=1}{\overset{\#S(P_n)}{R}} v_{P_n}(t_i, s_j) \\ &= \left(\begin{array}{lcccc} & @t_k\textbf{TM} & \textbf{S}_{\textbf{P}_1} & \ldots & \textbf{S}_{\textbf{P}_n} \\ [\textbf{t}_0, \textbf{t}_1] & \delta(@t_k\textbf{TM}) & \perp & \cdots & \perp \\ (\textbf{t}_1, \textbf{t}_2] & @t_1 & V_{P_1} & \cdots & - \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\textbf{t}_1, \textbf{t}_\textbf{n}] & @t_n & - & \cdots & V_{P_n} \end{array} \right) \end{aligned} \tag{6.88}$$

where �$(@t_k\textbf{TM})$ = �$(@t_k\textbf{N})$ is the evaluation function as defined in Eq. 4.88b.

The semantic diagram of the time-driven dispatching process relation as defined in Eq. 6.88 is illustrated in Fig. 6.30 in the semantic environment Θ.

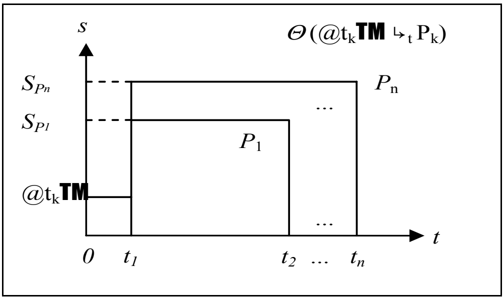


Figure 6.30 The semantic diagram of time-driven dispatch relation

6.6.3.3 The Event-Driven Dispatching Process Relation

Definition 6.101 The semantics of the *event-driven dispatching relations* of processes, $\theta(@e_k\mathbf{S} \hookrightarrow_e P_k)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(@e_k\mathbf{S} \hookrightarrow_e P_k)$ on the sets of variables S and executing steps T, i.e.:

$$\begin{aligned}
\theta(@e_k\mathbf{S} \hookrightarrow_e P_k) &\triangleq \frac{\partial^2}{\partial t\, \partial s} f_\theta(@e_k\mathbf{S} \hookrightarrow_e P_k) \\
&= \mathop{\mathcal{R}}_{k=1}^{n} (@e_k\mathbf{S} \rightarrow \frac{\partial^2}{\partial t(P_k)\partial s(P_k)} f_\theta(P_k)) \\
&= \mathop{\mathcal{R}}_{k=1}^{n} (@e_k\mathbf{S} \rightarrow \mathop{R}_{i=0}^{\#T(P_k)} \mathop{R}_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)) \\
&= @e_1\mathbf{S} \rightarrow \mathop{R}_{i=0}^{\#T(P_1)} \mathop{R}_{j=1}^{\#S(P_1)} v_{P_1}(t_i, s_j) \\
&\quad | \ldots \\
&\quad | @e_n\mathbf{S} \rightarrow \mathop{R}_{i=0}^{\#T(P_n)} \mathop{R}_{j=1}^{\#S(P_n)} v_{P_n}(t_i, s_j) \\
&= \left(\begin{array}{ccccc}
 & @e_k\mathbf{S} & \mathbf{S}_{\mathbf{P_1}} & \ldots & \mathbf{S}_{\mathbf{P_n}} \\
[\mathbf{t_0}, \mathbf{t_1}] & \delta(@e_k\mathbf{S}) & \perp & \cdots & \perp \\
(\mathbf{t_1}, \mathbf{t_2}] & @e_1 & V_{P_1} & \cdots & - \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(\mathbf{t_1}, \mathbf{t_n}] & @e_n & - & \cdots & V_{P_n}
\end{array} \right)
\end{aligned} \tag{6.89}$$

The semantic diagram of the event-driven process relation as defined in Eq. 6.89 is illustrated in Fig. 6.31 in the semantic environment Θ.

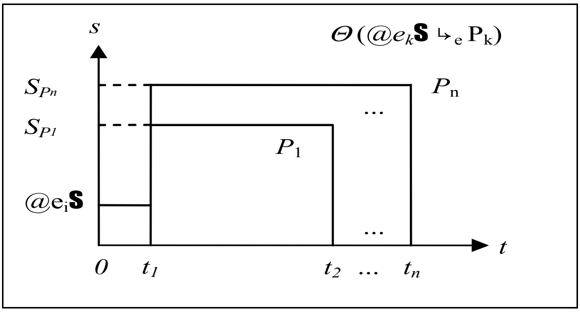


Figure 6.31 The semantic diagram of the event-driven dispatch relation

6.6.3.4 The Interrupt-Driven Dispatching Process Relation

Definition 6.102 The semantics of the *interrupt-driven dispatching relations* of processes, $\theta(@int_k\mathbf{S} \hookrightarrow_i \mathrm{P}_i)$, in the given semantic environment Θ is a double partial differential of the semantic function $f_\theta(@int_k\mathbf{S} \hookrightarrow_i \mathrm{P}_i)$ on the sets of variables S and executing steps T, i.e.:

$$
\begin{aligned}
\theta(@int_k\mathbf{S} \hookrightarrow_i \mathrm{P}_k) &\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@int_k\mathbf{S} \hookrightarrow_i \mathrm{P}_k) \\
&= \mathop{R}\limits_{k=1}^{n}(@int_k\mathbf{S} \rightarrow \frac{\partial^2}{\partial t(P_k)\partial s(P_k)} f_\theta(P_k)) \\
&= \mathop{R}\limits_{k=1}^{n}(@int_k\mathbf{S} \rightarrow \mathop{R}\limits_{i=0}^{\#T(P_k)} \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)) \\
&= \ @int_1\mathbf{S} \rightarrow \mathop{R}\limits_{i=0}^{\#T(P_1)} \mathop{R}\limits_{j=1}^{\#S(P_1)} v_{P_1}(t_i, s_j) \\
&\quad\ | \ldots \\
&\quad\ | @int_n\mathbf{S} \rightarrow \mathop{R}\limits_{i=0}^{\#T(P_n)} \mathop{R}\limits_{j=1}^{\#S(P_n)} v_{P_n}(t_i, s_j) \\
&= \left(\begin{array}{ccccc}
 & \boldsymbol{@int_k\mathbf{S}} & \mathbf{S_{P_1}} & \ldots & \mathbf{S_{P_n}} \\
\mathbf{[t_0,t_1]} & \delta(@int_k\mathbf{S}) & \perp & \cdots & \perp \\
\mathbf{(t_1,t_2]} & @e_1 & V_{P_1} & \cdots & - \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\mathbf{(t_1,t_n]} & @e_n & - & \cdots & V_{P_n}
\end{array}\right)
\end{aligned} \tag{6.90}
$$

The semantic diagram of the interrupt-driven process relation as defined in Eq. 6.90 is illustrated in Fig. 6.32 in the semantic environment Θ.

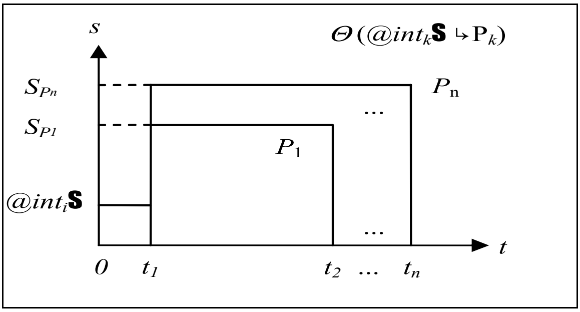


Figure 6.32 The semantic diagram of the interrupt-driven dispatch relation

Semantics plays an important role in language processing, formal methods, and software engineering theories. This section has presented a rigorous treatment of RTPA deductive semantics, which enables a new approach towards deductive reasoning of software semantics at all composing levels of program hierarchy. Deductive semantics has greatly simplified the description and analysis of the semantics of complicated software systems implemented in programming languages or specified in formal notations. Deductive semantics can be used to define both abstract and concrete semantics of large-scale software systems, facilitate software comprehension and recognition, support tool development, enable semantics-based software testing and verification, and explore the semantic complexity of software systems.

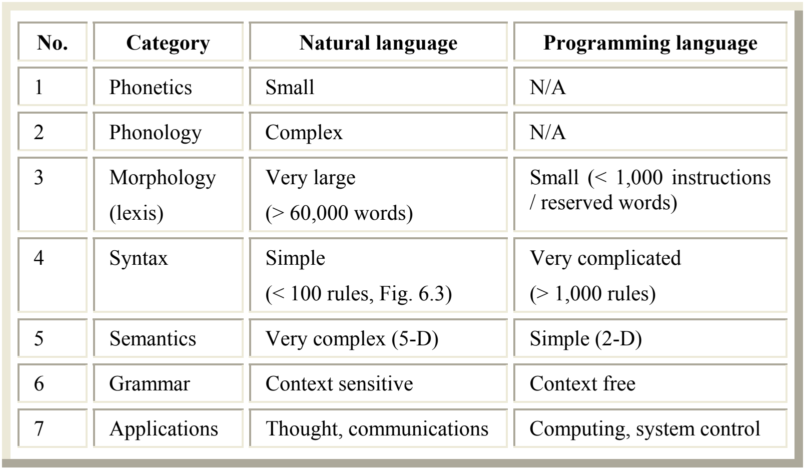
6.7 Linguistic Perceptions on Software Engineering

Because software engineering is the application of information technologies in communicating between professionals and customers, architects and software engineers, programmers and computers, as well as computing systems and their environments, linguistics and formal language theories play important roles in software engineering. This section takes a comparative approach to explore the common and special characteristics and features of natural and programming languages. It analyzes how the formal language theories extends the study of natural languages, and how linguistics may improve the understanding of programming languages and their work products – software.

6.7.1 COMPARATIVE ANALYSIS OF NATURAL AND PROGRAMMING LANGUAGE THEORIES

It is interesting to compare the features of programming languages and those of natural ones. A summary of the comparative analysis of programming and natural languages is provided in Table 6.11 on the basis of the discussions in preceding sections. Intuitively, it is expected that a programming language would be a small subset of natural languages. Surprisingly, this hypothesis is only partially true at the morphology (lexicon) and semantic levels. However, the syntax of programming languages is far more complicated than those of natural languages.

Table 6.11
Comparative Analysis of Natural and Programming Language Theories



No.	Category	Natural language	Programming language
1	Phonetics	Small	N/A
2	Phonology	Complex	N/A
3	Morphology (lexis)	Very large (> 60,000 words)	Small (< 1,000 instructions / reserved words)
4	Syntax	Simple (< 100 rules, Fig. 6.3)	Very complicated (> 1,000 rules)
5	Semantics	Very complex (5-D)	Simple (2-D)
6	Grammar	Context sensitive	Context free
7	Applications	Thought, communications	Computing, system control

It can also be seen in Table 6.11 that the semantics of programming languages is much simpler than that of natural languages, which is determined by the basic objectives of applications that should be suitable for limited machine intelligence. However, for achieving such simple and precise semantics in programming languages, a very complex and rigorous syntax and grammatical rules have to be adopted.

More generally, it is noteworthy that there is no clear-cutting between syntax and semantics in both natural and programming languages as stated in

Theorem 6.1. In other words, syntactic and semantic rules are equivalent and interchangeable in linguistics. A simple syntax will require for a complex semantics, while a complex syntax will result in a simple semantics.

6.7.2 PRINCIPLES OF PROGRAMMING LANGUAGE DESIGN

A variety of programming languages have been designed and proposed in the last five decades, either procedural or object-oriented, assemble or high-level, general purpose or special usages [Louden, 1993]. For example, FORTRAN was seen as focused on execution efficiency, COBLE was emphasized on natural language like readability, Pascal was to provide a structured language for maintaining complexity, and C++ and Java are designed for object-orientation and more structural encapsulation of language components. All of them may be perceived as instances derived on the basis of the formal language theory discussed in previous subsections.

A number of common principles shared in programming language design can be elicited from existing and historical programming languages [Hoare, 1966/73; Wirth, 1974; Horowitz, 1984; Mitchell, 1996]. This subsection describes the basic principles and generic criteria for programming language design, which may be used to evaluate and appreciate the features of different existing and future programming languages.

6.7.2.1 Abstraction and Complexity Control

Abstraction is a primitive design principle of programming languages. It is also a basic engineering principle for controlling design complexity in software engineering, since any professional language itself is an abstract symbolic system for describing and exchanging notions. The abstractive power of programming languages helps programmers to denote real-world applications with mathematical-based architectural and behavioral models, which can then be embodied by executable code.

6.7.2.2 Efficiency

Efficiency is a ubiquitous requirement for programming languages. Efficiency can be classified into programming efficiency, language processor implementation efficiency, as well as target code time and memory efficiency.

Programming efficiency is the innate characteristic of programming language that depends on the *expressiveness*, *writeability*, and *readability* of the language. These features involve both the express power on data architectures and behavioral processes. A structural or object-oriented methodology for building complex structures from basic components and architectures are also necessary.

Language processor implementation efficiency refers to the implementability and complexity of compilers or interpreters for a given language. Some special grammar rules and run-time supporting features of programming languages may dramatically increase the implement complexity and efficiency.

Target code efficiency is closely linked with tool implementing technologies and their efficiency. The basic criteria for code efficiency can be evaluated by time, space, or both. Usually the time and space efficiencies of code may be contradictory to each other in a given environment. Therefore, tradeoffs between them are often required, and almost all modern compilers provide options on optimizing a required feature during compiling.

6.7.2.3 Expressivity

Expressivity is the principle for language descriptivity and its preciseness. Expressivity of languages can be classified as those for architecture and data manipulation, behaviors and processes manipulation, and I/O and environment interaction manipulation. A common model for expressivity of programming languages developed in RTPA is the expressions of *object architectures* and *behavioral processes* [Wang, 2002a]. The former can be described generically by CLMs, and the latter can be described by a set of 17 meta processes and their algebraic relations. Since both CLMs and processes of RTPA are defined on the basis of a small set of algebraic laws and operators, the system models specified in RTPA are both precise and expressive.

6.7.2.4 Simplicity

Simplicity is the principles for enabling writeability, readability, and efficiency. The successful story of RISC (Reduced Instruction Set Computing) technology in microinstruction level [Marshall, 1989; Bhandarhar and Clark, 1991] has proven that a smaller and essential instruction set can be more efficient in computing and programming. According to Theorem 4.8, an important finding of the work in developing RTPA [Wang, 2002a] is that the basic set of behavioral instructions for any programming language includes only 17 meta processes and 17 process relations, where a meta process is a basic instruction in computing. Therefore, simplicity is an important design principle of programming languages that drive language designers and researchers to seek the core and essential expressive constructs in computing and programming.

6.7.2.5 Uniformity

Uniformity is a principle for language instructions' consistency and generality of appearance and behaviors. Similar instructions should adopt similar appearance and behavior; dissimilar instructions should not be easily confused. Basic control structures should be implemented in unified syntax and implies consistent semantics.

6.7.2.6 Orthogonality

Orthogonality is a principle of language design which requires that all language constructs should behave the same in any context. The orthogonality of programming languages enables different constructs to be freely composed in applications with a predictable behavior. For example, all instructions or data objects should behave independently and context-freely in either sequential or embedded program components.

6.7.2.7 Comprehensibility and Readability

Comprehensibility is the feature of how understandable of a program and its readability. With the increasing complexity of large-scale system development, code reviews, and legacy system reengineering, comprehensibility or readability has gained more and more attention than writeability among language designers and researchers in software engineering. Hence, C.A.R. Hoare asserted that "The readability of programs is immeasurably more important than their writeability [Hoare, 1973]."

6.7.3 CHARACTERISTICS OF PROGRAMMING LANGUAGES

The architectural characteristics of programming and natural languages have been contrasted in Table 6.11. This subsection reviews the basic requirements for programming and the characteristics of programming languages from a linguistic perspective.

6.7.3.1 Fundamental Requirements for Programming

The basic elements for computing have been identified in Section 5.2.1. For supporting and implementing the basic computing requirements, the necessary expressive power of programming languages is as follows:

- Arithmetical operations
- Logic operations

- Data and memory manipulations
- Inputs/outputs manipulations
- Events timing and processing
- Interrupt and parallel dispatching of processes

Observing Table 6.3 it can be seen that although natural languages can be rich, complex, and powerfully descriptive, they share the common and basic mechanisms, such as '*to be* (|=),' '*to have* (|⊂),' and '*to do* (|>).' Programming languages presented a comprehensive set of instructions on describing system actions and behaviors, such as the sequential, branch, iterative, recursive, concurrent, parallel, and interruptive process relations [Wang, 2005a/06c/06h/06f]. However, 'to be (|=)' is not adequately represented in programming languages. This results in a vital weakness and a lot of ambiguity in programming languages in describing architectures of software and its components.

The 17 meta processes of RTPA and their composition rules known as the 17 process relations provide a sufficient and comprehensive descriptive power. According to Theorem 4.8, only binary orthogonal combination of the meta processes and process relations may result in up to 2,312 programming instructions, which is rich enough than any programming language to describe architectures and behaviors of software systems.

6.7.3.2 Characteristics of Programming Languages

The characteristics of programming languages, perceived from a linguistic point of view, can be described as follows:

- An artificial language
- Limited alphabet and grammar
- To be learnt as a second (or *n*th, *n* > 2) language
- Only a written language
- No tenses and timing
- No person (presumed 'I' or the computer)
- Context free
- Objects are typed
- Language and tool integration
- Designed to manipulate abstract objects

It is noteworthy that there is a cognitive and expressive gap between the context-free programming languages and the context-sensitive natural languages. Therefore, a fundamental issue is how a real-world problem and its solution(s) in a context-sensitive manner may be described by a relatively inadequate programming language. Because the context cannot be freely removed without loss of useful information, programming languages have to imply the context of a program in data objects and related semantic environment. This forms a fundamental constraint on automatic code generation by machines, because no machine can recognize implied and inexplicitly expressed semantics in a program. This is one of the key reasons for why software engineering is still a labor (programmer) intensive discipline. RTPA presents a context-expressive software notation system, where the programming context or the semantic environment for a given problem is explicitly described by a system architectural model with encapsulated data objects in terms of a set of CLMs. Based on RTPA, intelligent and automatic code generation systems have been successfully implemented for software engineering as presented in Section 15.4.2.

6.8 Summary

This chapter has demonstrated that **formal language theories** play an important role in computing theories, without it computing and software engineering theories are not complete. **Language** is an oral and/or written symbolic system for thought, self-expression, and communication. **Linguistics** is the discipline that studies human or natural languages. This chapter has extended linguistics to **programming languages** and **professional notation systems** known as formal languages.

A comparative approach has been adopted to explore the common and special characteristics of human and programming languages. This chapter has analyzed not only how linguistics may improve the understanding of programming languages and their work products – software, but also how formal language theories extend the study of natural languages.

This chapter has explored the linguistics foundations of software engineering and analyzed the expressive means and their rigorous treatment in software engineering. Classic thought in linguistics, such as syntaxes, semantics, grammars, and linguistic analyses, has been reviewed. Formal treatment of language elements and their compositions from the bottom up have been described. Syntaxes and semantics of programming languages and their analyses have been presented. Semantics of RTPA have been formally described on the basis of deductive semantics. Comparative analyses of natural and programming languages, as well as linguistics perceptions on software engineering, have been discussed. As a result, the **linguistic foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Linguistics Foundations of Software Engineering*, readers have achieved the following strategic aims with the knowledge architecture as summarized below.

Chapter 6. Linguistics Foundations of SE

■ Fundamentals of Linguistics

- Taxonomy of linguistics
- Syntaxes
- Semantics
- Grammars
- Formal analysis of syntaxes

■ Formal Language Theory

- Alphabets
- Strings
- Expressions
- Grammar theories
- Languages
- BNF and EBNF

■ Syntax of Programming Languages

- Lexical analyses
- Syntax definitions and descriptions
- Syntactical analyses
- Syntactical analyses of RTPA

■ Semantics of Programming Languages

- Taxonomy of semantics
  - Target semantics
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics
  - Algebraic semantics
  - Deductive semantics
- Denotational semantics

  - Syntactic and semantic domains of denotational semantics
  - Description of syntactic domains of SPL
  - Semantic analysis using denotational semantics
  - Semantics of programs in SPL

- Deductive semantics
  - The mathematical model of software semantics
  - Deductive semantics of programs at different levels of compositions
  - Properties of software semantics

■ Semantics of RTPA

- Semantics of RTPA meta processes
  - The assignment process
  - The evaluation process
  - The addressing process
  - The memory allocation process
  - The memory release process
  - The read process
  - The write process
  - The input process
  - The output process
  - The timing process
  - The duration process
  - The increase process
  - The decrease process
  - The exception detection process
  - The skip process
  - The stop process

- Semantics of RTPA process relations
  - The sequential process relation
  - The jump process relation
  - The branch process relation
  - The switch process relation
  - The while-loop process relation
  - The repeat-loop process relation
  - The for-loop process relation
  - The function call process relation
  - The recursive process relation
  - The parallel process relation
  - The concurrent process relation
  - The interleave process relation
  - The pipeline process relation
  - The interrupt process relation

- Semantics of system and process dispatching
    - The system process
    - The time-driven dispatch process relation
    - The event-driven dispatch process relation
    - The interrupt-driven dispatch process relation

■ Linguistic Perceptions on Software Engineering

- Comparative analysis of natural and programming languages
- Principles of programming language design
- Characteristics of programming languages

SIGNIFICANT FINDINGS OF THIS CHAPTER

• The need centered in software engineering is to efficiently facilitate **communications** among multiple stakeholders, such as those between *professionals* and *customers, architects* and *software engineers, programmers* and *computers,* as well as *computing systems* and *their environments*. Therefore, **linguistics and formal language** theories play important roles in computing and software engineering.

• The **basic function of languages** is both to *communicate information* and to *express* abstract human *behaviors*.

• It is recognized that the ways to **express** human and system behaviors can be classified into three categories: **to *be*, to *have*,** and **to *do*** in natural languages. All **mathematical means** and forms, in general, are abstract description and manipulation of these three categories of human and system behaviors and common rules.

• **Natural languages** are *context-sensitive*, while programming languages are *context-free*. Therefore, the **descriptive power** of programming languages is inherently limited than that of the needs for expressing and solving natural-world problems.

• Most fundamental problems in computing and software engineering may stem from the **removal or implication** of the computing environments and data objects. Therefore, a much natural and **context-expressive** programming language and related compiling technology are yet to be sought.

- An important discovery in modern linguistics is the existence of the **universal grammar** (UG) among human languages. UG and grammars of natural languages can be formally described and analyzed using formal language theories. A typical UG is the **Deductive Grammar of English** (DGE) as established in Section 6.2.4.

- Although natural languages can be rich, complex, and powerfully descriptive, their functions can be classified into three **fundamental categories** known as ‘*to be* (|=),’ ‘*to have* (|⊂),’ and ‘*to do* (|>).’

- In software engineering, **formal language theories** are reoriented to express software specification and design notions, rather than focussing on language generation and recognition.

- A **programming language** can be designed and generated from the **bottom up** according to a set of lexes and syntaxes. Reversely, the language can be recognized, analyzed, and reduced from the **top down** via lexical and syntactic analyses. **Software engineering** puts emphases on language *recognition, cognition*, and its *expressiveness* rather than language generation.

- Semantics of a programming language can be described by its **behavioral equivalence** to another language. Semantics can also be described by a set of predefined **executable functions in machine languages**. Another approach to specify the semantics of a programming language is by mathematical definitions known as **formal semantics**.

- According to deductive semantics, the **carriers of software semantics** are a finite set of variables declared in a given program. Therefore, program semantics can be reduced onto the changes of values of these variables.

- The **behavior** of a computational statement is a set of observable actions or changes of status of objects operated by the statement.

- **Semantic analysis** is a deductive process that projects the 3-D software behaviors into the 2-D semantic environment.

- **Programming languages vs. natural languages**: It was expected that a programming language would be a small subset of natural languages. Surprisingly, this hypothesis is only partially true to the *morphology* (*lexicon*) and *semantics*, because the *syntax* of programming languages is far more complicated that those of natural languages.

- **Syntax vs. semantics** of programming languages: Syntactic and semantic rules are *equivalent* and *interchangeable.* A simple syntax requires for a complex semantics, while a complex syntax results in a simple semantics.

- **Deductive semantics** is a formal software semantics that deduces the semantics of a program in a given programming language from a generic abstract semantic function to the concrete semantics, which are embodied onto the changes of status of a finite set of variables constituting the semantic environment of computing.

  - The **advantage of deductive semantics** is that it introduces only **a universal semantic function** rather than adopting multiple concrete semantic functions as the conventional approaches do. In deductive semantics, any particular concrete semantic function is a deduced instantiation of the universal abstract semantic function. This avoids the trouble in other exhaustive approaches where new semantic functions have to be particularly defined whenever additional instructions are introduced in a given language.

  - Deductive semantics can be used not only to describe the **abstract and concrete semantics** of programs, but also to elicit and highlight their **semantic effects**.

- Historically, **language-centered programming** had been the dominant methodology in computing and software engineering. However, this should not be taken for granted as the only approach to software engineering, because the expressive power of programming languages is inadequate to deal with complicated software systems. In addition, the rigorousness and level of abstraction of programming languages are too low in modeling the architectures and behaviors of software systems. Therefore, the recognition of the need for **mathematical modeling** of both **software system architectures** and **static/dynamic behaviors,** as well as the support of **automatic code generation** systems, is profoundly important.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Fundamentals of Linguistics

- **Linguistics** is the discipline that studies the nature and use of languages. The central issue of linguistics is grammar, the rules of a language and how to generate, form, recognize, and interpret the language.

- Linguistics is one of the important foundations of computing and software engineering because languages are the **basic means** of human **communication** and **tools of thinking** and expression.

- The basic function of languages is to express abstract human behaviors and to communicate information. Any human language, natural or artificial, is **a sequential or 1-D symbol stream** of syntactical blocks, which, from top down, are *paragraphs, sentences, phrases, words,* and *letters*.

- **Syntax** is a domain of linguistics that studies sentence formation and structures. Syntax of languages is multi-dimensional (n-D).

- **Lexical elements** in a language can be classified into the categories of *lexical, functional, phrasal,* and *relational*.

- **Semantics** is a domain of linguistics that studies the interpretation of words and sentences, and analysis of their meanings. Semantic analysis and comprehension is a **deductive cognitive process**. The semantics of a sentence is comprehended till all elements of the sentence can be reduced to either a real-world image or a primitive abstract concept, and the logical relations of parts of the sentence are clarified.

- A **grammar** is a set of common rules that integrates phonetics, phonology, morphology, syntax, and semantics of a given language. The grammar for the multi-dimensional syntax of languages is hierarchical and recursive. The basic properties of natural language grammars are *generality, parity, universality, mutability,* and *inaccessibility*.

- The **universal grammar** (UG) is a system of categories, mechanisms, and constraints shared by all human languages. UG is perceived as innate in the brain based on recent neurolinguistic and psycholinguistic studies.

- A paradigm of UG is the rigorous definition of the **English grammar** by the **deductive grammar** (DGE) at the sentence level. Any valid English sentences can be derived on the basis of DGE.

Formal Language Theory

- A **programming language** is a special notation system for describing and specifying instructive computing information on both architectural (data) and behavioral (process) aspects of a software system.

- **Formal languages** are rigorously defined theories and rules of programming languages to specify, analyze, generate, and recognize computational languages. Formal language theories study the language elements such as *alphabets, strings, expressions, languages,* and *grammars*.

- An **alphabet** is a nonempty finite set of symbols or letters.

- A **string** (word) over an alphabet is a finite sequence of symbols defined on the alphabet. The **closure** of an alphabet $\sum^*$ is a power set of the given alphabet.

- An **expression** is a string on an alphabet or a number of strings concatenated by a set of special symbols known as operators.

- A **regular expression** is a special kind of strings consisting of single symbols on a given alphabet, or composed of single symbols by the empty string $\varnothing$, union $\cup$, repeat *, and parentheses ( ).

- A **language** is a set of expressions and strings over an alphabet that are formed following certain properties and rules known as grammar. A **regular language** over an alphabet $\sum$ is a set of regular expressions on $\sum^*$. A language is regular if and only if it is accepted by a finite automaton. A **context-free language** L_f is a language generated by a context-free grammar G_f, i.e., $L_f = L(G_f)$.

- A **production** p is a function that produces an ordered pair (α, β), i.e., p: $\alpha \to \beta$, where α and β is a terminal, nonterminal, or their combinations. A production with all terminals on its RHS is a *final product* with its semantics or physical meaning defined, while a product with at least one nonterminal on its RHS is an *intermediate product* with its semantics pending on further deduction.

- **Chomsky Grammars**: Based on the types of production rules adopted in a grammar, formal grammars can be classified as Type 0 through Type 3 from the bottom up.

  - A *Type 0 grammar,* G_0, is a grammar that has no restrictions on its productions.

  - A *Type 1 grammar,* G_1, is a grammar that satisfies: $\forall p \in G_1, p$: $\alpha \to \varnothing \vee (p\text{: } \alpha \to \beta \Rightarrow |\alpha| \le |\beta|)$.

  - A *Type 2 grammar,* G_2, is a grammar that satisfies: $\forall p \in G_2, p$: $A \to \beta$.

  - A *Type 3 grammar,* G_3, is a grammar that satisfies: $\forall p \in G_3, p$: $s_0 \to \varnothing \vee p\text{: } A \to a \vee p\text{: } A \to aB$ where s_0 is the start symbol and a is a single terminal.

- A **context** of a production is a certain configuration of all symbols in the strings and expressions of a production.

  - A **context-sensitive grammar** G_s is constrained by: $\forall p \in G_s, p$: $\alpha A \alpha' \to \alpha\beta\alpha'$, where $\alpha X \alpha'$ is the context, and X is a nonterminal that can be replaced in the given context.

  - A **context-free grammar** G_f is constrained by: $\forall p \in G_f,\ p$: $A \to \beta$.

  - A **regular grammar** G_r is a grammar that is constrained by: $\forall p \in G_r,\ p$: $S_0 \to \varnothing \vee p$: $A \to a \vee p$: $A \to aB$.

- **Levels of grammars**: A higher level grammar imposes stronger restrictions on its production rules than those of the lower level grammars, i.e., $G_3\ (G_r) \subseteq G_2\ (G_f) \subseteq G_1\ (G_s) \subseteq G_0$.

- An ***LL*(k) grammar** is a class of context-free grammars $G_f = (\sum, s, T, R)$, where the first L defines that the parsing is from *l*eft to right, and the second L specifies that next production is derived by *l*eft-most derivation, and k, $k \geq 1$, denotes that at most k-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.

- An ***LR*(k) grammar** is a class of context-free grammars $G_f = (\sum, s, T, R)$, where the L defines that the parsing is from *l*eft to right, and the R specifies that next production is derived by *r*ight-most derivation in reverse, and k, $k \geq 0$, denotes that at most k-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.

- A **Backus-Naur form** (BNF) is defined by a 5-tuple: $BNF \triangleq (\sum, T, V, P, S)$. An **extended BNF** (EBNF) adopts an extended set of metasymbols $S' = \{\ |, (\ )^*, (\ )^+, [\ ]\}$. BNF/EBNF are a recursive notation for describing the productions of a context-free grammar.

Syntax of Programming Languages

- The **syntactic processing** of programs encompasses lexical and syntactical analyses.

- The **lexical structure** of a programming language is the structures of its lexemes, such as strings or words, known as *tokens* in language processing.

- **Tokens** of a programming language can be classified into three categories that represent program entities of *reserved words, reserved symbols* (operators and separators), and *identifiers* (variables and constants).

- Lexical analyses are conducted by a **scanner** or a **lexical analyzer**.

- **Lexical analysis** breaks down a source code into a finite sequence of individual tokens; for each of them its language property is identified.

- The **syntax** of a programming language is its grammatical rules for constructing legal instructions.

- **Grammar rules** of a language that constrain and direct a syntactic analysis of a parser can be described by BNF, EBNF, syntax diagrams, or RTPA.

- Syntactical analyses are conducted by the **parser** or **syntactical analyzer**.

- **Syntactical analysis** techniques can be classified into *top-down* and *bottom-up parsing* that adopt the LL(k) or LR(k) grammar, respectively.

- **Top-down parsing** is a class of parsing techniques directed by an *LL*(*k*) grammar that matches an input string to a given syntax tree in a preorder, i.e., from the root of the syntax tree to the leftmost nodes.

- **Recursive-descent parsing** is a top-down parsing technique that derives a parsing tree according to a set of left-recursive grammar rules.

- **Predictive parsing** is a restricted form of recursive-descent parsing where the backtracking is eliminated in a top-down parsing by adopting an *LL*(1) grammar.

- **Bottom-up parsing** is a class of parsing techniques that derives a parse tree for an input string from the leaves to the root, in order to reduce the string to the start symbol of production rule.

- **RTPA syntactical analyses**: Most of the RTPA syntax for software system specification and modeling can be specified by a set of about 300 *LL*(*k*) grammar rules in EBNF.

- Special grammar rules of RTPA are described by **syntactic predicate** in the form of <*syntactic entity*> => <production>, which is a selective backtracking to recognize language constructs that cannot be distinguished without seeing all or most of the construct.

- On the basis of the EBNF grammar rules, the **RTPA parser** and **type checker** are implemented using ANTLR.

Semantics of Programming Languages

- The **semantics of a programming language** is the behavioral meanings that constitute what an instructional statement of the language is to do. The **semantics of a program** in a given programming language is the logical consequences of an execution of the program that results in the changes of values of a finite set of variables in the underlying computing environment.

- **Formal semantics** of programming languages can be classified into five categories known as *operational semantics, denotational semantics, axiomatic semantics, algebraic semantics,* and *deductive semantics*.

    - **Operational semantics** adopts a virtual machine, whose operation is well-defined, to describe the semantics of a programming language by its *equivalent behaviors* executing by the virtual machine.

    - **Denotational semantics** adopts functions to describe the semantics of a programming language, in which the *function* describes semantics by associating semantic values to syntactically legal constructs.

    - **Axiomatic semantics** adopt effective assertions to describe the semantics of a programming language, in which the *assertions of effect* by executing an instruction is deduced to the values of data manipulated by the instruction.

    - **Algebraic semantics** adopt abstract algebra to describe the semantics of a programming language, in which data objects and operations are defined by algebraic axioms and deduced by abstract algebraic laws.

- **Deductive semantics** is a formal software semantics that deduces the semantics of a program in a given programming language from a generic abstract semantic function to the concrete semantics, which are embodied onto the changes of status of a finite set of variables constituting the semantic environment of computing.

- A **semantic environment** of a program in a given programming language is a logical model of a finite set of identifiers and their values changing over time along the execution of the program.

- The **behavioral space** Ω of a program executed on a certain machine is a finite set of variables operated in a three-dimensional state space determined by a finite set of operations *O*, a finite set of memory locations or their logical representations by identifiers of variables *S*, and a finite set of discrete steps of program execution *T*.

- The **semantic environment** Θ of a program on a certain target machine is its run-time behavioral space Ω projected onto the Cartesian plane determined by *T* and *S*, i.e., $\Theta = \dfrac{\partial^2 \Omega}{\partial t\ \partial s} = T \times S$.

- A **semantic function** of a program $\wp$, $f_\theta(\wp)$, is a finite set of values *V* determined by a Cartesian product on a finite set of variables *S* and a finite set of executing steps *T*, i.e., $f_\theta(\wp) = f\colon T \times S \to V$, where $T = \{t_0, t_1, ..., t_n\}$, $S = \{s_1, s_2, ..., s_m\}$, and *V* is a set of values $v(t_i, s_j)$, 0 ≤ i ≤ n, and 1 ≤ j ≤ m.

- A **semantic diagram** is a sub-Cartesian-plane in the semantic environment Θ that forms the domain of the semantic function for a composed process *P* with $f_\theta(P) = f\colon T_P \times S_P \to V_P$.

- The **semantics of a statement** *p*, *θ(p)*, in a given semantic environment Θ is a double partial differential of the semantic function $f_\theta(p)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$
\begin{aligned}
\theta(p) &= \frac{\partial^2}{\partial t\ \partial s} f_\theta(p) \\
&= \mathop{R}_{i=0}^{\#T(p)} \mathop{R}_{j=1}^{\#S(p)} v_p(t_i, s_j) \\
&= \mathop{R}_{i=0}^{1} \mathop{R}_{j=1}^{\#\{s_1, s_2, ..., s_m\}} v_p(t_i, s_j) \\
&= \left(\begin{array}{ccccc}
 & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\
\mathbf{t_0} & v_{01} & v_{02} & \cdots & v_{0m} \\
(\mathbf{t_0}, \mathbf{t_1}] & v_{11} & v_{12} & \cdots & v_{1m}
\end{array} \right)
\end{aligned}
$$

- The **semantic effect** of a statement *p*, $\theta^*(p)$, is the resulted changes of values of variables by its semantic function $\theta(p)$ during the time interval immediately before and after the execution of *p*, $\Delta t = (t_i, t_{i+1}]$, i.e.:

$$\begin{aligned}\theta^*(p) &= \mathop{R}_{j=1}^{\#S(p)} (v_p(t_i, s_j) \oplus v_p(t_{i+1}, s_j)) \\ &= \mathop{R}_{j=1}^{\#S(p)} < v_p(t_i, s_j) \rightarrow v_p(t_{i+1}, s_j) \,|\, v_p(t_i, s_j) \neq v_p(t_{i+1}, s_j) >\end{aligned}$$

- The **semantics of a process** *P*, *θ(P)*, in a given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P)$ on the sets of variables *S* and executing steps *T*, i.e.:

$$\begin{aligned}\theta(P) &= \frac{\partial^2}{\partial t\, \partial s} f_\theta(P) \\ &= \mathop{R}_{k=1}^{n-1} \{[\frac{\partial^2}{\partial t\, \partial s} f_\theta(P_k)]\; r_{kl}\; [\frac{\partial^2}{\partial t\, \partial s} f_\theta(P_l)]\}, l = k+1 \\ &= \mathop{R}_{k=1}^{n-1} \{[\mathop{R}_{i=0}^{\#T(P_k)} \mathop{R}_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)]\; r_{kl}\; [\mathop{R}_{i=0}^{\#T(P_l)} \mathop{R}_{j=1}^{\#S(P_l)} v_{P_l}(t_i, s_j)]\} \\ &= \begin{pmatrix} \mathbf{V_{P_1}} & & & & \mathbf{V_G} \\ & \mathbf{V_{P_2}} & & & \mathbf{V_G} \\ & & \ddots & & \vdots \\ & & & \mathbf{V_{P_{n-1}}} & \mathbf{V_G} \end{pmatrix}\end{aligned}$$

where $\mathbf{V}_{\mathbf{P}_k}$, $1 \leq k \leq n\text{-}1$, is a set of values of local variables that belongs to processes P_k, and V_G is a finite set of values of global variables.

- The **semantics of a program** ℘, *θ*(℘), in a given semantic environment Θ, is a combination of the semantic functions of all processes $\theta(P_k)$, $1 \leq k \leq n$, i.e.:

$$\begin{aligned}\theta(\wp) &= \mathop{R}_{k=1}^{\#K(\wp)} \frac{\partial^2}{\partial t\, \partial s} f_\theta(\wp) \\ &= \mathop{R}_{k=1}^{\#K(\wp)} \theta(P_k) \\ &= \mathop{R}_{k=1}^{\#K(\wp)} [\mathop{R}_{i=0}^{\#T(P_k)} \mathop{R}_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)]\end{aligned}$$

- The **semantic space of a program** $S_\Theta(\wp)$ is a product of #*S*(℘) variables and #*T*(℘) executing steps, i.e.:

$$S_\Theta(\wp) = \#S(\wp) \bullet \#T(\wp)$$
$$= \sum_{\text{k=1}}^{\#K(\wp)} \#S(\wp_k) \bullet \sum_{\text{k=1}}^{\#K(\wp)} \#T(\wp_k)$$

- **Usage of deductive semantics**: a) To define both abstract and concrete semantics of large-scale software systems; b) To facilitate software comprehension and recognition; c) To support tool development; d) To enable semantics-based software testing and verification; and e) To explore the semantic complexity of software systems.

- The **semantics** of a program are **invariant** with the changes of executing speed, as long as any absolute time constraint is met.

- A program is **composable** in a given language *iff* both sufficient sets of meta instructions and BCS's are rigorously defined.

- **Semantics of RTPA**: Deductive semantics can be applied to derive and interpret the semantics of the 17 *meta processes* and the 17 *process relations* of RTPA, which covers an essential and sufficient set of fundamental computing requirements in programming and software engineering.

Linguistics Perceptions on Software Engineering

- **Programming languages vs. natural languages**: Although the *lexicon* and *semantics* of a programming language are a small subset of natural languages, the *syntax* of programming languages are far more complicated that those of natural languages.

- **Language design principles** elicited from existing and historical programming languages are *abstraction, efficiency*, *expressivity*, *simplicity*, *uniformity*, *orthogonality, comprehensibility*, and *readability*, which are used to evaluate and appreciate the features of different existing and future programming languages.

- The **limitations** of conventional software specification and compiling technologies make programming hard and complicated, because the objects (O), actions (B), and contexts (T, S) are separated, implied, and inexplicitly expressed. This is the **fundamental constraint** on code generation that can not be implemented by machines automatically, and dominates software engineering as still a **labor-intensive** discipline.

Questions and Research Opportunities

6.1 What are the roles of linguistics and formal language theories in programming and software engineering?

6.2 Discuss why the *language-centered programming* convention in computing and software engineering must not be taken for granted. What would be the alternative approaches to software engineering?

6.3 According to the HAMSD model provided in Theorem 1.4, discuss whether *mathematical modeling* of software system architectures and static/dynamic behaviors, as well as automatic *code generation systems* would be the silver bullets for software engineering.

6.4 It is recognized that the ways to express human and system behaviors can be classified into three categories: to *be*, to *have*, and to *do* in natural languages. All mathematical means and forms, in general, are abstract description and manipulation of these three categories of human and system behaviors and common rules.

Referring to Table 4.6, discuss the compatibility and differences between natural languages, programming languages, and denotational mathematical structures in their expressive power and level of abstraction.

6.5 Referring to Fig. 6.2, how is the deductive process of linguistic analyses conducted in natural languages from the 1-D language sentence to the 5-D semantics?

6.6 What are the Universal Grammar (UG) and its basic properties?

6.7 Why may any valid English sentence be derived by the generic schema on the basis of the Formal English Grammar (FEG)?

6.8 What are the relationship and compatibility between syntaxes and semantics of natural languages? Is the compatibility also applicable to programming languages?

6.9 What are the different orientations of formal language theories for computing and software engineering?

6.10 What is the hierarchical structure of objects under study in formal language from the bottom up?

6.11 What are the restrictions imposed to a general expression in order to obtain a regular expression?

6.12 What is an $LL(k)$ grammar and what does the first L, second L, and k stands for, respectively?

6.13 What is an $LR(k)$ grammar and what does L, R, and k stands for, respectively?

6.14 Summarize the following grammars and their relationships in a table based on Corollaries 6.1 and 6.2:

- *Chomsky grammar* types G_0, G_0, G_0, and G_3;
- The *context-sensitive* grammar G_s, the *context-free* grammar G_f, and the *regular* grammar G_r;
- The $LL(k)$ grammar and the $LR(k)$ grammar

6.15 What are the extensions of EBNF on BNF, and how do these extensions improve the express power of BNF notations?

6.16 Referring to Table 6.5, describe the relationship between EBNF notations and their mathematical semantics in RTPA.

6.17 Summarize the four forms of formal semantics, such as *operational semantics, denotational semantics, axiomatic semantics,* and *algebraic semantics, as well as* their methods and usages.

6.18 What is a semantic function? What are the differences between an abstract and a concrete semantic function?

6.19 What are deductive semantics and its generic abstract semantic function? What are the advantages of deductive semantics against conventional semantic theories?

6.20 Explain the relationship between a semantic environment and a semantic diagram as defined in deductive semantics.

6.21 Figs. 6.2 and 6.15 show the nature of semantic analyses for natural and programming languages, respectively. Analyze the similarity and differences between them.

6.22 Using the following program entitled *sum* as given in Example 6.24, analyze the semantics of Statements (0) through (3) using deductive semantics:

```
void sum;
{
  (0) int x, y, z;
  (1) x := 8;
  (2) y := 2;
  (3) z := x + y;
}
```

6.23 Draw a semantic diagram for the following composed processes which is a sequential relation between *P* and a while-loop for *Q*:

$$\theta(P \rightarrow \overset{\mathbf{F}}{\underset{\exp\mathbf{BL}=\mathbf{T}}{R}}{}^{*}Q)$$

6.24 Theorem 6.3, the *least complete set of instructions in programming*, states that a program is *composable* with sufficient descriptive power in a given language *iff* both the sufficient sets of meta instructions ($\mathfrak{P}$, Theorem 4.6) and compositional rules ($\mathfrak{R}$, Theorem 4.7) are rigorously defined.

Try to prove Theorem 6.3 on the basis of Theorems 5.7 and 4.8.

6.25 What are the syntactic and semantic differences between the roles of statements (meta processes) and BCS's (process relations)?

6.26 Comparatively analyze the linguistic complexities of programming languages and natural languages in the following aspects: *lexis, syntax, semantics, grammar,* and *applications*. Identify in which aspects a programming language would be far more complicated than a natural language.

6.27 Read the following classic article in software engineering:

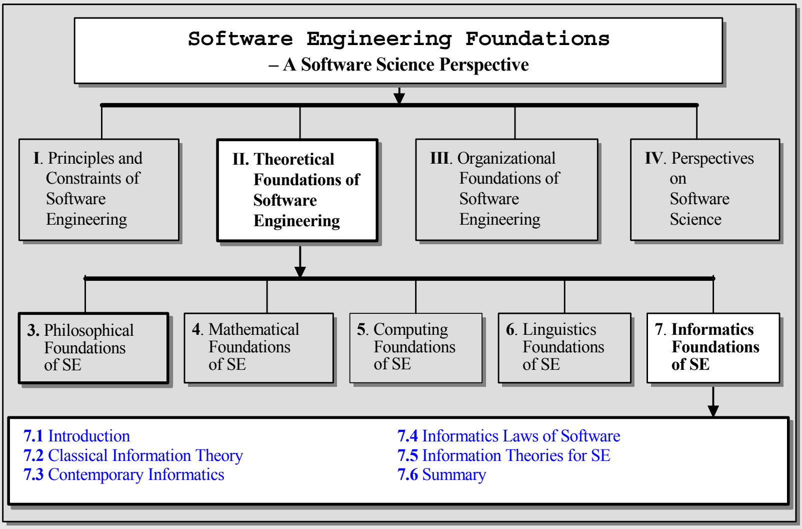
> C.A.R. Hoare (1981), The Emperor's Old Clothes, the The 1980 Turing Award Lecture, *Communications of the ACM*, 24(2), pp. 75-83.

Discuss the following topics in a group:

- About the author.
- What are the 'new clothes' of the emperor in software engineering?
- Identify your own examples of promising technical hoaxes that were popular in software engineering but disappeared shortly.
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 7

INFORMATION SCIENCE FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
3. Philosophical Foundations of SE
4. Mathematical Foundations of SE
5. Computing Foundations of SE
6. Linguistics Foundations of SE
7. Informatics Foundations of SE

7. Information Science Foundations of SE

Knowledge Structure

❍ Classic information theory

- Shannon's definition of information
- The physical meaning of classic information
- Domain of classical information theory
- Subjectivity of classical information theory

❍ Contemporary informatics

- Information: the third essence of nature
- Measurement of information
- From machine informatics to cognitive informatics

❍ Informatics laws of software

- What constrains software?
- Equivalence between information-matter-energy
- Informatics laws and properties of software

❍ Information theories for software engineering

- The informatics metaphor of software
- Informatics laws that constrain software behaviors
- The informatics attributes of software quality

Learning Objectives

- To know the essences of the classical information theory.
- To understand contemporary information theory and the measurement of information.
- To be aware of the emergence of cognitive informatics based on contemporary informatics.
- To understand the 19 informatics laws and fundamental properties of software.
- To understand the cognitive functional complexity of software.
- To be able to apply informatics in software engineering, particularly the information metaphor and informatics laws.

"A little information, when shared, can go a long way."

BBC World (2005)

"A fundamental discovery in computer science is that software as a unique entity is not constrained by any law and principle known in the physical world. However, software obeys the laws of informatics."

"Cumulativeness is the most significant attribute of information that mankind relies on in evolution."

Yingxu Wang (2002)

7.1 Introduction

It is recognized that *matter*, *energy*, and *information* are the three essences of the natural and the abstract worlds according to the Information-Matter-Energy (IME) model [Wang, 2003a/2006a] as presented in Theorem 1.2. In a modern society, information plays more and more important roles because it is the only link between the physical (external) and the abstract (internal) worlds in human life, and the top-level requirement for achieving fundamental human esteems.

Information is the product of either natural or machine intelligence. *Informatics*, the science of information, studies the nature of information, its processing, and ways of transformation between information, matter, and energy. Informatics has developed from the classic information theory [Hartley, 1928; Shannon, 1948/49a/49b/51/59; Shannon and Weaver, 1949; Bell, 1953; Goldman, 1953; Reza, 1961], contemporary informatics [Chaitin, 1977/04; Zhong, 1996; Nielsen and Chuang, 2000; David, 2002; Wang, 2002d/03b], to cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06] in the past half century.

A fundamental discovery in computer science and software engineering was that software as a unique entity is not constrained by any law and principle known in the physical world [McDermid, 1991; Hartmanis, 1994; Wang, 2006a]. Hence, it is curious to query the following:

- What are the constraints that software obeys?

Software in informatics is perceived as instructive and behavioral information. Referring to the abstractive levels of human knowledge and information as given in the HAMSD model (Fig. 1.3) in Section 1.2.4, software is information at Level 4 (special notation systems) and/or Level 5 (mathematics). Therefore, software possesses all the properties of information and may be formally treated by informatics theories.

This chapter attempts to demonstrate that software obeys the laws of informatics [Wang, 2006a/06b/07a]. As a logical consequence, it explores the following important issue:

- What are the laws of informatics that constrain software in software engineering?

In order to answer this fundamental question, this chapter examines the informatics properties and laws of software and software engineering. In the remainder of this chapter, the information science foundations of software engineering will be presented from classic, contemporary, to cognitive informatics. Section 7.2 briefly reviews classic information theories and its perception on information as probability-based properties of signals and channels. Section 7.3 presents contemporary informatics and current perception on information in the Information Technology (IT) and software industries. Section 7.4 explores a comprehensive set of informatics laws that constrain the behaviors of software. Then, Section 7.5 describes information science for software engineering and applications of informatics in software engineering.

7.2 Classic Information Theory

Information theory, system theory, and *cybernetics* were regarded as the major three theories invented in the 1940s, which have greatly influenced related research, industrial applications, and human life since the second half of the 20th century. The classic information theory is regarded to be founded by Claude E. Shannon during 1948-1949 [Shannon, 1948/49a/49b/51/59; Shannon and Weaver, 1949], while the term *information* was first adopted by Hartley in 1928 [Hartley, 1928], and extensively studied by Bell and Goldman in 1953, respectively [Bell, 1953; Goldman, 1953]. Conventional information theory was modeled based on probability theory, and was focused on information transmission rather than information itself [Reza, 1961; Kolmogorov, 1965].

7.2.1 SHANNON'S PERCEPTION ON INFORMATION

In the early 1940s, it was thought that increasing the transmission rate of information over a communication channel increased the probability of errors. Shannon surprised the communication theory community by proving that this was not true as long as the communication rate was below the capacity of a channel, where the channel capacity is constrained by its noise characteristics.

In the classic information theory, Shannon [Shannon, 1948/49a/49b/51/59] defines information as a probabilistic measure of the variability of messages that can be obtained from a message source [Shannon, 1948/59]. In other words, the physical meaning of information is the prediction of variability of any kind of signals that can be sent via transmission channels.

Definition 7.1 *Information* is a weighted probabilistic measure of the variability of messages (signals) that is expected from a message source via a transmission channel.

Definition 7.2 The *information variability* of the *i*th sign in a message, I_i, is determined by its unexpectedness, i.e.:

$$I_i = log_2 \frac{1}{p_i} \quad \text{[bit]} \tag{7.1}$$

where p_i is the probability that the *i*th sign is transmitted. The unit of information is *bit*, shortened from '*binary digit.*'

On the basis of Definition 7.2, the total information variability of a given signal system can be derived below.

Definition 7.3 The *total information variability* transmitted by a source or sender, *I,* is the weighted sum of the probability of all its *n* possible signs I_i, $1 \le i \le n$, known as the alphabet, in the message, i.e.:

$$\begin{aligned} I &= \sum_{i=1}^{n} p_i \bullet I_i \\ &= \sum_{i=1}^{n} p_i \bullet \log_2 \frac{1}{p_i} \\ &= - \sum_{i=1}^{n} p_i \bullet \log_2 p_i \quad \text{[bit]} \end{aligned} \tag{7.2}$$

Example 7.1 For a binary source that has an alphabet of two equally likely signs, i.e., $p_1 = p_2 = 0.5$, its total information variability, *I*, is:

$$\begin{aligned} I &= \sum_{i=1}^{n} p_i \bullet \log_2 \frac{1}{p_i} \\ &= \sum_{i=1}^{2} 0.5 \bullet \log_2 \frac{1}{0.5} \\ &= 2 \bullet 0.5 \\ &= 1 \quad [\text{bit}] \end{aligned} \tag{7.3}$$

The classic information theory perceived that information is any kind of signals that can be sent via transmission channels where the signals' probability is predictable. It is noteworthy in this theory: a) If the statistical probability of any sign in the message is either p_i = 0 or unknown, which is often the case, then there is no definition of information; and b) When p_i = 1, there is no information may be received.

It is noteworthy that for the above binary system, the information variability is always 1 bit. In other words, *I* is a measure of information variability rather than that of information quantity. Therefore, *I* is not proportional to the sizes of messages, according to Eq. 7.2. On the basis of the classical information measurement, no matter how many bit messages have been transmitted, the value of the *I* will not change for a given transmission system. This result may be surprisingly contradictory to the common sense of information in contemporary informatics and in the IT industry.

However, the most important contribution of classic information theory is the identification of the fundamental information unit *bit*. It indicates that the foundation of information is a binary digit. Therefore, any other kind of complex information can then be reduced to the measurement of bit. This forms a common foundation for both contemporary informatics and computer science as discussed in Theorem 5.1.

7.2.2 THE PHYSICAL MEANING OF CLASSIC INFORMATION

An important concept that links the classic information theory with physics is *entropy* [Shannon, 1951; Brillouin, 1953; Cutnell and Johnson, 1998; Witold, 2007b]. Entropy is an inherent property of both concrete and abstract information systems that drifts them to disorder.

7.2.2.1 The Concept of Entropy

Entropy is not a physical entity in the concrete world but a measure of the extent of chaos of a given system. Entropy is the concept studied intensively in thermodynamics [Cutnell and Johnson, 1998].

Definition 7.4 *Entropy* is the extent of the trend of a system towards complete disorder or randomization.

Definition 7.5 The *quantity of information entropy* H_i of a message source is determined by the average weighted information variability I transmitted by the source, i.e.:

$$\begin{aligned} H_i &= I \\ &= - \sum_{i=1}^{n} p_i \bullet \log_2 p_i \quad [\text{bit}] \end{aligned} \tag{7.4}$$

Eq. 7.4 shows that entropy may be measured by the information variability. In other words, the nature of information variability is entropy. Actually, Shannon initially used entropy to denote the information variability of signal systems and channels.

It is found that the maximum entropy of a given source occurs if the probabilities of all signals are equal [Shannon, 1948].

7.2.2.2 The Laws of Thermodynamics

A thermodynamic system is the collection of objects on which attention on energy transformation is being focused with respect to the surrounding environment. Energy is an inherent property of matter and systems, which exists in various forms, such as heat, mechanical (kinetic or potential) energy, chemical energy, and radiant energy,

Definition 7.6 *Thermodynamics* is the branch of physics built upon the fundamental laws obeyed by energy in the forms of heat and work and their transformation.

Thermodynamics treats *temperature* as the statistical measure of the thermal status of a system. A basic principle of *thermal equilibrium* states the status of a system there is no flow of heat within it. Because all forms of energy may be degraded to heat, the rules that apply to heat transformations may be used to describe energy changes and exchanges in systems.

The three laws of thermodynamics are the basic theories that govern exchanges of energy. Although these laws may be expressed in a number of ways, the most common descriptions are provided in this subsection [Cutnell and Johnson, 1998].

Lemma 7.1 *The first law of thermodynamics, conservation of energy,* states that energy can be neither created nor destroyed, so that the total input of energy E_i in any transformation must equal the total output of energy E_o, i.e.:

$$\sum E_i \equiv \sum E_o \tag{7.5}$$

Eq. 7.5 can also be expressed by perceiving the change of internal energy ΔE due to heat Q and work W, i.e.:

$$\Delta E = Q - W \tag{7.6}$$

where a positive or negative Q represents the system gains or loses heat, and a positive or negative W denotes a system does the work or receives the work, respectively.

The second law of thermodynamics that deals with natural tendency of heat can be described as follows [Cutnell and Johnson, 1998].

Lemma 7.2 *The second law of thermodynamics, the heat flow statement,* states that heat flows spontaneously from a substance at a higher temperature to a substance at a lower temperature, and does not flow spontaneously in the reversed direction.

The second law of thermodynamics is the most profound law in all of science, which shows that energy during its forms change tends to become degraded to scattered states in which the capacity for useful work diminishes. Although, the total energy of a system is always conservative according to the first law, the second law reveals that the ability of the energy to be utilized for useful work continuously decreases.

Since entropy is a measure of the disorder, the random property of energy and the second law may be explained by the natural tendency for entropy to increase in a transformation system. In thermodynamics, the thermal entropy is defined as follows.

Definition 7.7 The *thermal entropy* H_t is a function of the state or condition of a system. For a reversible process, the *change in the entropy* ΔH can be defined by the heat Q divided by the temperature T in Kelvins, i.e.:

$$\Delta H_t = \frac{Q}{T} \qquad [J/K] \tag{7.7}$$

It is noteworthy that the unit of thermal entropy H_t in thermodynamics is a Joule per Kelvin [J/K], while in informatics the information entropy H_i is defined as a pure quantity with the unit of bit.

With the introduction of thermal entropy, Lemma 7.2 can be revised as follows [Cutnell and Johnson, 1998].

Corollary 7.1 *The second law of thermodynamics, the heat flow statement,* states that:

(a) Entropy of the universe ΔH_u does not change when a reversible process occurs, i.e.: $\Delta H_u = 0$, and

(b) Entropy of the universe ΔH_u increases when an irreversible process occurs, i.e.: $\Delta H_u > 0$.

Because entropy can be interpreted in terms of order and disorder, when an irreversible process occurs and the entropy of the universe increases, the energy available for doing work decreases.

Lemma 7.3 *The third law of thermodynamics, the state of maximum order*, states that a perfect crystal at a temperature of absolute zero possesses zero entropy, i.e.:

$$\lim_{T \to 0} H_t = 0 \tag{7.8}$$

where the unit of temperature of T is in Kelvin.

The third law may also be described as that it is not possible to lower the temperature of any system to absolute zero in a finite number of steps. The third law of thermodynamics emphasizes the prevalence of disorder in almost all natural states and systems, because ideal crystallization is rarely achievable and the temperature of absolute zero is unattainable.

7.2.2.3 Transformation between Information Entropy and Thermal Entropy

According to the second law of thermodynamics, the information entropy and the thermal entropy in a system is conservative. Hence, an extended view of the second law of thermodynamics can be perceived as follows.

Corollary 7.2 The *extended 2nd law of thermodynamics* states that in any system, the sum of the information entropy H_i and the thermal entropy H_t is a constant, i.e.:

$$k_t\,|H_t\,| + k_i\,H_i = \varepsilon \quad \text{[bit]} \tag{7.9}$$

where $k_t,\ k_i,\ \varepsilon$ are positive constants for a given system, and the unit of k_t is Kelvin per Joule (K/J).

Eq. 7.9 indicates that for the decrease of the thermal entropy of a system, the information entropy has to be increased, and vice versa. Therefore, the information entropy is also perceived as the *negative entropy.*

7.2.3 DOMAIN OF CLASSIC INFORMATION THEORY

Although classic information theory was intended to be applied in a very broad area, the domain of it mainly encompasses *communication* and *coding* theories. The former studies models of communication channels, noises, and signal processing. The latter deal with data encoding, decoding, compression, protection, and encryption.

The structure of the domain of classical information theory can be described as follows:

- Communication theories
- Models of communication channels
- Noise behaviors
- Signal processing
- Coding theories
- Data compression
- Data protection
- Data encryption

Classic information theory is good at answering two fundamental questions in communication theory: a) What is the ultimate transmission rate of communication? and b) What is the maximum rate of data compression?

For the former, Shannon revealed that the ultimate transmission rate of a channel is the maximum channel capacity. For the latter, Shannon answered that the maximum data compression rate is the entropy (information) of the data [Shannon, 1948]. In addition to being the foundation of communication theory, classic information theory has also found a wide range of applications in algorithm complexity analysis in computer science, functional and information sizes measurement in software engineering, and statistic mechanisms in physics.

7.2.4 SUBJECTIVITY OF CLASSIC INFORMATION THEORY

It is noteworthy that classic information theory is not about the measure of information itself rather than its variability or entropy. A dilemma in the conventional information theory is that the measurement of the variability or entropy of information is dependent on the receiver's subjective judgment. According to classic information theory, information is the message that one does not expect and does not know. Therefore, a subjective criterion has been introduced into the objective measurement of information. This results in that the same message represents varying information for different observers depending on their degrees of awareness of the message. Further, whenever one reads the same message later, the information that one may obtain degrades over time because of the loss of uncertainty.

For instance, the following assertions that were thought to be true according to classic information theory may be doubtable:

- If the contents of a message are already known, there is no expected information.
- The same message may have information for some people, and no information for others, dependent on their previous knowledge about the expected message.
- Every time, when one reads the same message, the information that one may get is different and decreasing.
- Information of a message is dependent on the probability distribution of all signs in the alphabet, but not the size of the message.

Information is perceived as entropy of signals on the basis of statistical probability. The subjective nature of entropy deems there is no information if the probability of a sign in a message is 1. In both cases when the probability of signals is indeterminable or 0, the entropy or information is undefined.

Alternative information theories have been proposed to improve the classical theory, such as nonprobability-based theory [Chaitin, 1977/04; Nielsen and Chuang, 2000; David, 2002], decision theory [Berger, 1990;

Edwards and Fasolo, 2001; Carlsson and Turban, 2002; Wang, 2005b/05e; Wang and Ruhe, 2007], and belief theory [Kramosil, 2001].

7.3 Contemporary Informatics

The domain of informatics has been extended in the last decades along with the development in computer science and in the IT industry. Conventional informatics treats *information* as a probabilistic measure of the variability or uncertainty of messages that can be received from a source. It was focused on information transmission rather than information itself. However, contemporary informatics tends to regard information as entities of messages, rather than a measurement of the messages' probability properties as that of the classic information theory [Chaitin, 1977/04; Zhong, 1996; Nielsen and Chuang, 2000; David, 2002; Wang, 2002d/03b; Wang and Patel, 2004]. The new perception is found better to explain the theories and practices in the IT and computer/software industries, because it treats information as any aspect of the natural world that can be abstractly represented and mentally processed [Wang, 2002d/03b].

This section first explores the nature of information in contemporary informatics and the measurement of information. Then, it discusses the transition of information science from machine informatics to cognitive informatics, and from external informatics to internal informatics inside the brain.

7.3.1 INFORMATION: THE THIRD ESSENCE OF NATURE

According to the IME model (Theorem 1.2), information is recognized as the third essence of the natural world supplementing to matter and energy [Wang, 2003a], because the primary function of the human brain is information processing.

It is observed that in applied computing and software sciences as well as in the IT industries, the term information has a much more practical and concrete meaning that focuses on data and knowledge representation, storage and processing. With this orientation, information is regarded as an entity of messages, rather than a measurement or metric of the messages' variability. With this perspective, the definition of information has been shifted from the classical informatics to the contemporary informatics as follows [Wang, 2002d/03b].

Definition 7.8 *Information* in contemporary informatics is defined as any property or attribute of the natural world that can be generally abstracted, quantitatively represented, and mentally processed.

From Definition 7.8 it can be seen that the intension and extension of information have been shifted from the probability of messages to organized data that represent the messages, knowledge, and/or abstracted real-world entities. With this new orientation, information is regarded as an independent and essential entity in modeling the natural world, particularly its abstract aspect.

7.3.2 MEASUREMENT OF INFORMATION

With the new orientation as discussed in Section 7.3.1, information is regarded as an entity of messages, rather than a measurement of the variability of messages. From this perspective, a definition of information can be derived as follows [Wang, 2003b].

Definition 7.9 The *measurement of information*, I_k, is defined by the cost of code to abstractly represent a given size of internal message M in the brain in a digital system based on k, i.e.:

$$\begin{aligned} I_k &= f : M \to S_k \\ &= \lceil \log_k M \rceil \end{aligned} \tag{7.10}$$

where I_k is the content of information in a k-based digital system, and S_k the measurement scale based on k. The unit of I_k is the number of k-based digits.

Eq. 7.10 is a generic measure of information sizes. When a binary digital representation system is adopted, i.e., $k = b = 2$, it becomes the most fundamental one for the meta-level representation of information.

The 19th Principle of Software Engineering

Theorem 7.1 The *primitive form of information* states that the most fundamental form of information that can be represented and processed is binary digit where $k = b = 2$, i.e.:

$$\begin{aligned} I_b &= f : M \to S_b \\ &= \lceil \log_b M \rceil \\ &= \lceil \log_2 M \rceil \quad [\text{bit}] \end{aligned} \tag{7.11}$$

Theorem 7.1 indicates that any form of information in the physical (natural) and abstract (mental) worlds can be unified on the basis of bits. This is the informatics foundation of modern digital computers and natural intelligence.

Note that the *bit* here is a concrete and deterministic unit, and it is no longer probability-based as in conventional information theories [Shannon, 1948; Bell, 1953]. In a certain extent, computer science and engineering is a branch of contemporary informatics that studies machine representation and processing of external information; while cognitive informatics is a branch of contemporary informatics that studies internal information representation and processing in the brain (see Chapter 9).

Example 7.2 According to Eq. 7.10, for given messages M_1 = 2 bits and $M_2 = 2^{30}$ bits, their information contents can be determined, respectively, as follows:

$$\begin{aligned} I_{b1} &= \lceil \log_2 M_1 \rceil \\ &= \lceil \log_2 2 \rceil \\ &= 1 \quad [bit] \end{aligned}$$

and

$$\begin{aligned} I_{b2} &= \lceil \log_2 M_2 \rceil \\ &= \left\lceil \log_2 2^{30} \right\rceil \\ &= 30 \quad [bit] \end{aligned}$$

The results show that messages M_1 and M_2 contain, respectively, 1 bit and 30 bits information. In other words, information I_{b1} = 1 bit and I_{b1} = 30 bits may represent messages in sizes of 2 or 2^{30} bits, respectively.

With the new perception on information according to Definitions 7.8 and 7.9, it is natural and intuitive to perceive IT as any technology that can be used for the processing of information. In the same way, an information system can be defined as an abstract representation system for information elicitation, acquisition, storage, manipulation (adding, deleting, updating), production, presentation, searching, and retrieving.

7.3.3 FROM MACHINE INFORMATICS TO COGNITIVE INFORMATICS

According to the IME model, the information theories discussed in Sections 7.2 and 7.3 so far can be collectively classified as *external* informatics. Complementary to it, there is a whole range of new research areas known as cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06]. The emerging discipline of *cognitive informatics* developed recently forms a profound interdisciplinary study of cognitive and information sciences. Cognitive informatics is a cutting-edge interdisciplinary research area that tackles the fundamental problems of modern informatics, computation, software engineering, artificial intelligence, cognitive science, neuropsychology, and life sciences. Almost all of the hard problems yet to be solved in the above areas share a common root in the understanding of mechanisms of natural intelligence and cognitive processes of the brain.

Cognitive informatics is perceived as a new frontier that explores the internal information processing mechanisms of the brain, and their engineering applications in computing and the IT industry. This subsection briefly introduces the historical development of informatics from the classical information theory and contemporary informatics, to cognitive informatics. The domain of cognitive informatics and its interdisciplinary nature are explored. Further details will be extensively described in Chapter 9.

7.3.3.1 Cognitive Informatics

The development of classical and contemporary informatics, the cross fertilization between computer science, software engineering, cognitive science and neuropsychology, has led to a whole range of extremely interesting new research areas known as cognitive informatics. Cognitive informatics is the transdisciplinary study of cognitive and information sciences that investigates into the internal information processing mechanisms and processes of the natural intelligence – human brains and minds. Cognitive informatics is a branch of information and computer science that studies computing by cognitive methodologies; and studies cognitive science by informatics and computing theories.

The first- and second-generation informatics put emphases on external information processing, which overlook the fundamental fact that human brains are both the original sources and final consumers of information, and any information must be cognized by the brain before it can be understood. This observation leads to the establishment of the third-generation informatics, a term coined as cognitive informatics [Wang, 2002d, Wang et al., 2002a].

In many disciplines of human knowledge, almost all of the hard problems yet to be solved share a common root in the understanding of the mechanisms of the natural intelligence and the cognitive processes of the brain. This leads to the study on cognitive informatics, and the query on the nature of information processing in the brain, such as information acquisition, representation, memory, retrieve, generation, and communication. Via an interdisciplinary approach and with the support of modern information and neural science technologies, mechanisms of the brain and the mind will be systematically explored in cognitive informatics.

The relationship between the internal and external informatics can be illustrated in Fig. 7.1. In cognitive informatics, the brain is perceived as the last thing in the world yet to be explored, where special recursive intelligent power of the brain is required. This makes cognitive informatics unique in distinguishing it from other natural sciences.

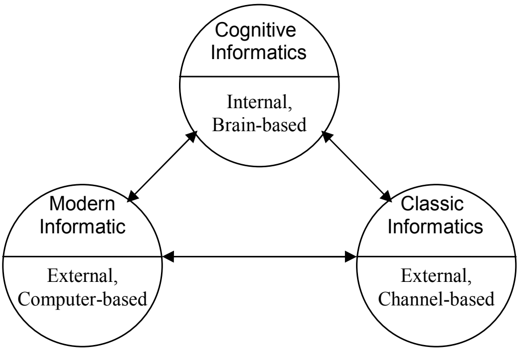


Figure 7.1 Relationship between the three-generation informatics

7.3.3.2 Perspective on Information in Cognitive Informatics

Definition 7.10 *Information* in cognitive informatics is defined as the abstract artifacts and their relations that can be modeled, processed, stored, and processed by human brains.

The measurement of information in cognitive informatics is similar to Definition 7.9 as given in Section 7.3.2. However, the basic unit of information, *bit*, in cognitive informatics corresponds to a single synaptic connection between neurons in the brain [Wang, 2003b; Wang et al., 2003].

Definition 7.11 The *measurement of cognitive information*, *I_c,* is defined by the cost of number of synaptic links to abstractly represent a given size of internal message *X* in the brain in a binary relational system, i.e.:

$$\begin{aligned} I_c &= f : X \to S_c \\ &= \lceil \log_2 X \rceil \qquad [bit] \end{aligned} \tag{7.12}$$

where S_c is the cognitive measurement scale based on number of synapses, and the unit of information, I_c, is a *bit*.

According to Theorem 7.1, the most fundamental form of information that can be represented and processed is bit. Any form of information in the physical (natural) and abstract (mental) worlds can be unified on the basis of bit. Both internal and external information share the same basic type in information representation.

7.3.3.3 The Role of Information in Mankind Evolution

It is recognized that the basic evolutional need of mankind is to preserve both the species' biological traits and the cumulated information/knowledge bases [Wang, 2005f/07a]. For the former, the gene pools are adopted to pass human trait information via DNA from generation to generation. However, for the latter, because acquired knowledge cannot be inherited between generations or individuals, various information means and systems are adopted to pass cumulated human information and knowledge.

Therefore, to a certain extent, mankind relies very much on information for evolution than that of genes, because the basic characteristic of the human brain is information processing. In other words, the ability to cumulate and transfer information from generation to generation plays the vital role in mankind evolution for both individuals and the species. This distinguishes human beings from other species in natural evolution, where the latter cannot systematically pass acquired information from generation to generation in order to enable it to grow cumulatively and exponentially [Wang, 2005f].

Further discussion on the role of information and its accumulation in societies will be discussed in Chapter 13 on sociological foundations of software engineering.

7.4 Informatics Laws of Software

A fundamental finding in computer science and software engineering is that software is not constrained by physical laws and properties because it is not a physical entity at all. Therefore, the study on what constrains software is a fundamental query yet to be explored in software engineering theories. It is also one of the key objectives of this book.

This section explores the informatics properties and laws of software and software engineering. The informatics laws of software may help to understand the nature of the objects studied in software science and engineering, and the unique constraints and methodologies that distinguish software engineering from other engineering disciplines.

7.4.1 EQUIVALENCE BETWEEN I-M-E

Before exploring the informatics properties of software, the equivalence between information, matter, and energy, as well as potential transformability among them, is explained in the following subsections.

7.4.1.1 Equivalence of Matter and Energy

According to the *theory of special relativity* discovered by Albert Einstein (1879 - 1855), one of the most enlightening results in modern physics is that mass and energy are equivalent. Einstein revealed that a moving object obeys the following mass-energy relation.

Lemma 7.4 (Einstein‘s Theory of Special Relativity): The *total energy E* of the moving object is related to its mass *m* and speed *v* by the following equation:

$$E = \frac{mc^2}{\sqrt{1-(v^2/c^2)}} \tag{7.13}$$

where *c* is the speed of light, a universal constant, that is measured to be 199,782,458 m/s.

The discovery on the transformability between matter and energy indicates that a gain or loss of mass can be regarded equally well as a loss or gain of energy, respectively.

A special case of Eq. 7.13 is when the object is at rest, i.e., $v = 0$. Under this condition, the *static energy* of the object E_o is obtained as:

$$E_o = mc^2 \tag{7.14}$$

Eq. 7.14 is Einstein’s then famous equation known as the *mass-energy relation* [Cutnell and Johnson, 1998].

In this sense, matter may be regarded as a special case of energy, and the loss of mass during nuclear reactions is transformed into awesome energy. This explains why the apparent discrepancy between energy input and output in nuclear reactions does not contradict the first law of thermodynamics (Lemma 7.1), because the release of tremendous amounts of energy in nuclear transformations such as fission or fusion is accounted for by the loss of mass during these reactions and the conversion of this mass to energy.

Another special case of Eq. 7.13 is that when the object is traveling at $v = c$. This results in the *maximum energy* of the object $E_{max} = \infty$, which indicates that no object with mass may travel at a speed as the same as light c in a vacuum, because as v approaches c, the kinetic energy becomes infinite. Hence an infinitive amount of work would have to be done to enable the object to reach c.

Inversely, the transformation of energy into matter has also been observed in experimental physics. In which, the pair production of the particles known as an electron and a positron can be generated by a high energy gamma ray when it hits the nucleus of an atom [Cutnell and Johnson, 1998].

7.4.1.2 Transformation between Matter, Energy, and Information

Information in cognitive informatics is deemed as the generic abstract artefacts that can be modeled, processed, and stored by human brains. Theories of cognitive informatics provide a new perception on information and informatics – the science of information – in the following aspects:

- Information is the 3rd essence in modeling the world.
- Any product and/or process of human mental activities results in information.
- Information, matter, and energy may be transferred between each other.
- Software obeys the laws of modern informatics and cognitive informatics.

The perceived transformability among I-M-E can be illustrated by Fig. 7.2, where all generic functions f_1 through f_6 obey the following corollary.

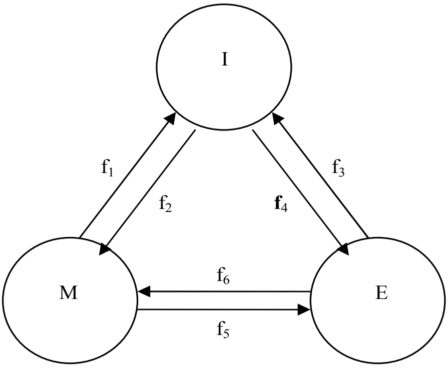


Figure 7.2 The transformability between I-M-E

Corollary 7.3 The *transformability between I-M-E* states that, according to the IME model, the three essences of the world are predicated to be transformable between each other as described by the following generic functions f_1 to f_6:

$$I = f_1\,(M) \qquad (7.15a)$$
$$M = f_2\,(I) \overset{?}{=} f_1^{\,-1}(I) \qquad (7.15b)$$

$$I = f_3\,(E) \qquad (7.15c)$$
$$E = f_4\,(I) \overset{?}{=} f_3^{\,-1}(I) \qquad (7.15d)$$

$$E = f_5\,(M) \qquad (7.15e)$$
$$M = f_6\,(E) = f_5^{\,-1}(E) \qquad (7.15f)$$

where a question mark on an equal sign denotes a hypothesis on the existence of such an inverse function.

Albert Einstein revealed Functions f_5 and f_6, the relationship between matter (m) and energy (E), in the form $E = mC^2$, where C is the speed of light. It is a great curiosity to explore what the remaining relationships and forms of transformation between I-M-E will be. In a certain extent, contemporary informatics is the science to seek possible solutions for f_1 to f_4. A clue to explore the relations and transformability is believed in the understanding of the natural intelligence and its information processing mechanisms [Wang, 2002d/03a/03b].

It is expected that any breakthrough in this area will significantly push forward the development of the next generation theories and technologies in informatics, computing, software, and cognitive sciences.

7.4.2 INFORMATICS LAWS AND PROPERTIES OF SOFTWARE

This subsection explores the informatics laws that constrain software by investigating the properties and principles of cognitive informatics. The properties and laws of information are helpful to explain the nature of information science and IT technology, which are tackling a wide range of fundamental problems in the interdisciplinary area between conventional natural sciences and modern informatics-based sciences, particularly, in the area of computing and software engineering.

A set of 19 informatics properties of software has been identified as follows [Wang, 2006a]:

1) Abstraction
2) Generality
3) Cumulativeness
4) Dependency on cognition
5) Multi-dimensional behavioral space
6) Sharability
7) Physically dimensionless
8) Weightless
9) Transformability between I-M-E
10) Multiple representation forms
11) Multiple carrying media
12) Multiple transmission forms
13) Dependency on media
14) Dependency on energy
15) Wearless and time dependency
16) Conservation of information entropy and thermal entropy
17) Information-based quality attributes
18) Susceptible to distortion
19) Scarcity

The following subsections provide detailed description of these properties of information and their applications in understanding the informatics laws that constrain software, software behaviors, and software engineering processes. Some of them may seem intuitive, but they are so profound in describing the axiomatic theory of informatics foundations of software engineering.

7.4.2.1 Abstraction

Property 7.1 *Abstraction*: Information is abstract artifacts that is elicited from physical entities in the natural world, or is created for representing relations between the entities or the entities and abstract mental objects. Information can be attributes, status, characteristics, structures, and dynamic processes of real-world entities, as well as relations between them. New information may be derived based on existing information and their relations in the abstract world.

Therefore, although it can be recorded, transformed, and communicated, information is the product of the brain and it exists in the abstract world. Theorem 1.2 on the IME model presented in Section 1.2 and the Object-Attribute-Relation (OAR) model of internal information representation in the brain presented in Section 9.4 [Wang, 2007g] provide a generic view about the abstractive property of information and its relationship with the real-world entities.

7.4.2.2 Generality

Property 7.2 *Generality*: According to Property 7.1 (*abstraction*) and the OAR model [Wang, 2007g], it can be derived that sources of information are widely general. Information can be elicited from objects, attributes, and their relations. Any physical entity in the universe is the source of information, and any abstract artifact (object) is the crystallization of information. Therefore, information is formed by the combination between physical entities, abstract objects, and relations between them, i.e.:

- Abstraction of physical entities and their attributes
- Relations between physical entities
- Relations between physical entities and abstract objects
- Relations between abstract objects

Hence, to a certain extent, contemporary informatics studies the sources and initiation of information, as well as the creation and perception of information by human cognitive processes.

7.4.2.3 Cumulativeness

Property 7.3 *Cumulativeness*: The physical world is conservative. According to the natural law of conservation, matter and energy can neither be reproduced nor destroyed (while they may be transformed). However, information is not conservative but cumulative, because information may be created, destroyed, and reproduced. Cumulativeness is the most significant attribute of information that mankind relies on in evolution.

7.4.2.4 Dependency on Cognition

Property 7.4 *Dependency on cognition*: Information should be recognized and consumed by human brains or other intelligent systems by a cognitive process before it can be effectively retained, retrieved, and used. According to the OAR model [Wang, 2007g], information is represented internally by its relations with existing information and knowledge in the brain. Without cognition and comprehension, there is no information and knowledge, also no access and retrieval of them.

7.4.2.5 Multi-Dimensional Behavioral Space

Property 7.5 *Multi-dimensional behavior space*: Information, as that of semantics of natural languages (Lemma 6.2), can be generically modeled by a 5-tuple encompassing the subject (J), behavior (B), object (O), time (T), and space (S), i.e.:

$$I = (J, B, O, T, S) \tag{7.16}$$

where behavior B is a set of observable actions, operations, or changes of status.

When the subject J and object O are obvious or they are implied, the information related to J and O, I_J, can be simplified as a triple as shown below:

$$I_J = (B_J, T, S) \tag{7.17}$$

where B_J is the behavior of J.

Therefore, software as instructive information, I_s, can be modeled in a 3-dimensional behavioral space Ω, i.e.:

$$\begin{aligned} I_s &= \Omega \\ &= (OP, T, S) \end{aligned} \tag{7.18}$$

where *OP* is a finite set of operations or computational behaviors.

7.4.2.6 Sharability

Property 7.6 *Shareability*: Information can be shared and reused by multiple users without loss in quantity and without degradation in quality. Information may be amplified or multiplied by broadcasting. The lossless reuse of existing information will usually result in the creation of new information.

7.4.2.7 Physically Dimensionless

Property 7.7 *Physically dimensionless*: Related to Property 7.1 (*abstraction*), information has no physical size and dimension. No matter how large or small the physical entities, their conceptual abstraction result in the same unit of information. Their abstract representations or the cognitive visual objects occupy a similar sight frame; only the resolutions may vary [Wang, 2003e; Wang and Wang, 2006].

7.4.2.8 Weightless

Property 7.8 *Weightless*: A direct corollary based on Property 7.7 (*physically dimensionless*) is that the weight of information, W_I, is always zero, i.e.:

$$W_I \equiv 0 \tag{7.19}$$

This explains why an empty or full hard disk has the same weight; a blank or recorded tape has no difference in weight; and a memory chip storing all 0, all 1, or any combinations of 0s and 1s has the same weight. This property of information can also explain why one can afford to obtain a PhD degree without feeling any change of the weight of the brain, rather than the changes of its internal configurations.

7.4.2.9 Transformability between I-M-E

Property 7.9 *Transformability between I-M-E*: According to the IME model (Theorem 1.1), the three essences of the world are predicated to be transformable between each other as shown in Fig. 7.2.

In Fig. 7.2, there are six possible relations between the three essences in the natural and information worlds. These relations can be described by the following generic functions f_1 through f_6 as given in Eqs. 7.16a through 7.16f.

7.4.2.10 Multiple Representation Forms

Property 7.10 *Multiple representation forms*: Related to Property 7.1 (*abstraction*) and Property 7.2 (*generality*), it is observed that information can be represented in multiple forms, such as analogue (audio, visual), abstract (written languages and notation systems), and digital.

In the above classification, *digitalization* in information representation is the most generic and fundamental approach. The cognitive foundation of digitalization is that information is represented discretely or granularly in the brain with the basic unit as individual neurons. Therefore, the discrete representability is the foundation of information representation, storage, and processing. It is also the foundation of modern digital multimedia information engineering.

7.4.2.11 Multiple Carrying Media

Property 7.11 *Multiple carrying media*: Parallel to Property 7.10 (*multiple representation forms*), information can be carried by various media, as listed in the following, and their combinations: electronic, electrical, magnetic, optical, mechanic, hydraulic, written, oral, and signs.

It is noteworthy that a certain medium may carry one or more forms of information. Correspondingly, a given form of information may be carried by different media.

7.4.2.12 Multiple Transmission Forms

Property 7.12 *Multiple transmission forms*: In addition to that information may be represented in multiple forms (Property 10) and carried by various media (Property 11), its transmission can be conducted in multiple forms as well. The following is the possible transmission forms of information:

a) Information *passing*:	1 - to - 1	
b) Information *broadcasting*:	1 - to - n	
c) Information *gathering*:	n - to - 1	
d) Information *networking*:	n - to - m	(7.20)

where 1 represents a single information source/receiver, *n* and *m* indicate multiple sources/receivers, and *n* and *m* can be the same.

The fast development of the Internet indicates that the fourth form of information transmission, *information networking*, is the highest form of communication.

7.4.2.13 Dependency on Media

Property 7.13 *Dependency on media*: Information can not exist without a storage medium. The types of media may be organic, physical, chemical, or the combinations of them as described in Property 7.11. Therefore, to some extent, information may be perceived as a change of status of the storage medium or matter.

7.4.2.14 Dependency on Energy

Property 7.14 *Dependency on energy*: All information processing tasks, such as acquisition, storage, retain, retrieve, and refresh, consume certain energy. There is no system that may process information without consuming energy. Therefore, in some extent, information may also be perceived as a change of status of energy on a given medium.

7.4.2.15 Wearless and Time Dependency

Property 7.15 *Wearless and time dependency*: The logic of formal information, such as special notation systems, mathematics, and philosophies as described at the abstract cognitive Levels 4 – 5 according to the HAMSD model in Section 1.2.4, does not wear out. Once the logic of a specific piece of information is true, it is always true and true forever.

However, the timeliness of informal information, as described at the abstract cognitive Levels 1 – 2, is much shorter, i.e., such kind of information may be out of date quickly.

7.4.2.16 Conservation of Information Entropy and Thermal Entropy

Property 7.16 *Conservation of information entropy and thermal entropy*: According to the extended second law of thermodynamics (Corollary 7.2), the sum of the information entropy and the thermal entropy in a given system is conservative. In a physical system, entropy can be reduced by input of *energy* in order to maintain the order of the system. In neural and social systems, the order and the state of organization can be increased by inputting information.

7.4.2.17 Information-Based Quality Attributes

On the basis of the conventional *product-based* metaphor [ISO 9126, 1991], the quality of software is perceived as a collection of external attributes, such as usability, availability, reliability, portability, and maintainability. Quality software is commonly considered as the software that contains fewer bugs. However, the intension of the concept of quality itself has never been properly defined in software engineering.

Property 7.17 *Information-based quality attributes*: To model the quality of software and information, a set of *information-based* quality attributes is identified as follows:

- Completeness
- Correctness
- Consistency
- Properly represented (no mis-interpretation)
- Clearness (no ambiguity)
- Feasible (can be implemented in technical and economical terms)
- Verifiable (attributes specified can be measured)

From this new angle, *software quality* can be defined as the achievement of the above inherent attributes for software architectures, static behaviors, and dynamic behaviors.

Comparing the above two approaches towards software quality, it can be seen that the former is a set of external quality attributes, and the latter is a set of internal ones. The *internal quality attributes* should be focused and controlled first in software engineering. Otherwise, it would be too late to examine the external quality attributes, because this may only be carried out after a software system has already been built.

7.4.2.18 Susceptible to Distortion

Property 7.18 *Susceptible to distortion*: Unlike physical entities, information is more fragile and vulnerable subjected to distortion, decay, and destroy. Therefore, information should be treated more carefully. Fault-tolerant and security techniques should be always adopted in dealing with information distortion.

7.4.2.19 Scarcity

Property 7.19 *Scarcity*: Information scarcity states that information when it is needed is always inadequate, constrained by its availability, awareness, and/or the cost and complexity to thoroughly search, acquire, and comprehend it.

According to the law of universal constraints as given in Theorem 3.1, any theory, method, or technology has its own limitations and constraints. To a certain extent, science and engineering are the searching of the maximum extent of general relations between entities, phenomena, and behaviors under a set of constraints.

7.5 Information Theories for Software Engineering

Software is a product of human intelligence that is used as a set of instructive information to implement computing behaviors on a generic computer. In a modern society, information plays more and more important roles because it is the only link between the physical (external) and the abstract (internal) worlds in human life. In cognitive informatics, software is perceived as a type of instructive and behavioral information that describes a solution for the design and implementation of a computing system [Wang, 2002g/03c/04b/05g/ 06a/07a].

A fundamental finding in computer science and software engineering is that software, as a unique entity, is not constrained by any law and principle known in the physical world [McDermid, 1991; Hartmanis, 1994]. This section attempts to demonstrate that software obeys the laws of informatics [Wang, 2006a], because software is a kind of instructive and behavioral information that is used to communicate with computer servers to provide specified functionality for users of the computing system.

7.5.1 THE INFORMATICS METAPHOR OF SOFTWARE

Software, in daily life, is simply meant as anything flexible and without a physical dimension. In the IT industry, software is perceived broadly as a concrete product [Baker, 1972; ISO 9001, 1989/94; ISO 9126, 1991; Taguchi, 1986; Jones, 1986; SQPL, 1990; Dromey, 1995]. With the *product metaphor*, a number of manufacturing technologies and quality assurance principles were introduced into software engineering. However, the phenomenon, in which we are facing almost the same problems in software engineering as we dealt with 40 years ago, indicates a failure of the manufacture-based and mass-production-oriented metaphor, and related technologies in software development. Therefore, the nature of software and how it may be effectively produced must be re-thought in software engineering.

According to the *informatics metaphor*, software can be perceived as follows.

Definition 7.12 *Software* is a kind of coded and instructive information for a computing system that describes the expected architectures and behaviors of the system in a programming language and related design documentations.

The above definition indicates a new way to explain the properties and laws that govern the behavior of software, which forms an important part of the informatics foundations of software engineering. Definition 7.12 also indicates that the current philosophy and methodology in organizing and managing software engineering and software development organizations as mass manufacturing and quality control processes are perhaps fundamentally mismatching.

Software as instructive and behavioral information describes the following:

- What are the abstract (or logical) models of *objects* in a given computing system?
- What do we do with these objects?
- How do we do with these objects?

The first question listed above implies that in software design we need to describe the abstract architecture of the system and its components by logical and algebraic modeling techniques. The last two questions indicate the requirements for describing the static and dynamic behaviors of the system as a set of interacting objects. It is perceived that the processes and techniques widely used in the publishing industry and the journalism industry are worth to be intensively studied and adopted in software engineering [Wang, 2004b], on the basis of the informatics metaphor.

7.5.2 INFORMATICS LAWS THAT CONSTRAIN SOFTWARE BEHAVIORS

Adopting the informatics metaphor of software, a set of 19 properties and laws of information have been developed in Section 7.4, which are used to explain the fundamental characteristics of information and software behaviors. The informatics laws indicate that, although software does not obey any physical laws of the natural world, it is indeed constrained by the informatics laws. Therefore, information science forms one of the foundations of software engineering and computing science.

The 21st Law of Software Engineering

Theorem 7.2 The *informatics laws of software* state that software architectures, behaviors, and processes are constrained by the 19 *informatics* laws of basic information properties.

The manufacture-based metaphor for software development has dominated the methodologies and technologies of software engineering in the last four decades. However, unsolved fundamental problems in software engineering indicate that the theoretical and empirical needs for investigating the informatics and other theoretical foundations of software engineering are yet to be explored.

7.5.3 THE INFORMATICS ATTRIBUTES OF SOFTWARE QUALITY

As will be discussed in Section 8.2.4, quality is one of the basic engineering objectives in any engineering disciplines. However, the term quality, particularly software quality, is a very complicated and collective concept that has never been well defined and formalized.

According to the informatics property 7.17 of software, the quality attributes of software can be classified as *external* and *internal* attributes. The internal quality attributes are those of a software system when it is treated as a white box. The seven information-oriented internal quality attributes of software as identified in Property 7.17 are extended in Table 7.1, encompassing *completeness, no misinterpretation, consistency, exactness* (nonambiguous), *no confliction, feasibility* (can be implemented in technical and economical terms), and *verifiability* (can be measured) [Wang, 2004b].

The external attributes of software quality are those of the product or system when it is treated as a black box, such as *functionality, usability, availability, reliability, efficiency, portability,* and *maintainability* [ISO 9126, 1991; ISO 9000-1, 1994; ISO 9000-2, 1994; ISO 9000-3, 1991; ISO 9000-4, 1993; ISO 9001, 1989; ISO 9001, 1994; ISO 9002, 1994; ISO 9003, 1994; ISO 9004-1, 1994; ISO 9004-2, 1991; ISO 9004-4, 1993; Jenner, 1995].

Table 7.1
Information-Oriented Quality Attributes of Software

No	Attribute	Description
1	Completeness	*Completeness* is an information-oriented software quality attribute that states a software system should *completely* describe and implement the system requirements.
2	No misinterpretation	*No misinterpretation* is an information-oriented software quality attribute that states a software system should *correctly* describe and implement the system requirements.
3	Consistency	*Consistency* is an information-oriented software quality attribute that states a software system should *accurately* describe and implement the system requirements.
4	Exactness	*Exactness* is an information-oriented software quality attribute that states a software system should *accurately and nonambiguously* describe and implement the system requirements.
5	No confliction	*No confliction* is an information-oriented software quality attribute that states a software system should recognize and solve *conflict* system requirements.
6	Feasibility	*Feasibility* is an information-oriented software quality attribute that states a software system should be *feasibly* designed and implemented against known technical or financial constraints.
7	Verifiability	*Verifiability* is an information-oriented software quality attribute that states a software system should recognize and qualify any nonverifiable system requirements.

A more formal treatment of quality theories and software quality based on the informatics metaphor will be presented in Chapter 8 on the engineering foundations of software engineering. Cognitive informatics as an emerging discipline and its applications in software engineering will be presented in Chapter 9 on cognitive informatics foundations of software engineering.

7.6 Summary

Information is the third essence of the natural world supplementing *matter* and *energy*. **Informatics**, the science of information, studies the nature of information, its processing, and ways of transformation between information, matter, and energy. In a modern society, information plays more and more important roles because it is the **only link** between the physical (external) and the abstract (internal) worlds.

Software is a type of instructive and behavioral information that describes a solution for the design and implementation of a computing system. According to the **Hierarchical Abstraction Model of System Descriptivity** (HAMSD), software can be categorized as information at abstraction Level 3 – special notation systems and/or Level 4 – mathematics.

The **manufacture-based metaphor** for software development has dominated the methodologies of software engineering in the last three decades. A new **informatics-based metaphor** has been proposed in this chapter for software engineering. In this chapter, the nature of software has been examined, which helped to clarify the basic informatics characteristics of software. The informatics laws of software have been perceived as an important part of the foundations of software engineering and computer science. The information-based metaphor on software will result in the development of new software notations, processes, quality principles, verification techniques, and organizational methodologies in software engineering and in the IT industry.

This chapter has explored the informatics nature of software and the information laws of software engineering. Classic information theories have been briefly reviewed. Contemporary informatics and current perception on information in the IT and software industries have been presented. A set of 19 informatics laws that constrain the behaviors of software has been identified. Applications of informatics in software engineering have been described, which leads to the emerging area known as cognitive informatics that will be further investigated in Chapter 9. As a result, the **information science foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Information Science Foundations of Software Engineering*, readers have achieved the following strategic goals with knowledge architecture as summarized below.

Chapter 7. Information Science Foundations of SE

- Classic Information Theory
  - Shannon’s definition of information
    - The concept of entropy
    - The laws of thermodynamics
    - Transformation between information entropy and thermal entropy

- Predicative measurement of information
- Information and entropy
- Domain of classical information theory
- Subjectivity of classical information theory

■ Contemporary Informatics

- Information: the third essence of nature
- Measurement of information
- From machine to cognitive informatics
  - Cognitive informatics
  - Perspective on information in cognitive informatics
  - The role of information in mankind evolution

■ Informatics Laws of Software

- Equivalence between information-matter-energy
  - The equivalence of matter and energy
  - Transformation between matter, energy, and information
- Informatics laws and properties of software
  - Abstraction
  - Generality
  - Cumulativeness
  - Dependency on cognition
  - Multi-dimensional behavioral space
  - Sharability
  - Physically dimensionless
  - Weightless
  - Transformability between I-M-E
  - Multiple representation forms
  - Multiple carrying media
  - Multiple transmission forms
  - Dependency on media
  - Dependency on energy
  - Wearless and time dependency
  - Conservation of information entropy and thermal entropy
  - Information-based quality attributes
  - Susceptible to distortion

■ Information Theories for Software Engineering

- The informatics metaphor of software
- Informatics laws that constrain software behaviors
- The informatics attributes of software quality

SIGNIFICANT FINDINGS OF THIS CHAPTER

- The **IME model** reveals that *matter*, *energy*, and *information* are the three essences of the natural and the abstract worlds. The relationships between IME and their **transformations** are one of the fundamental questions in cognitive informatics. It is believed that any breakthrough in this area will be profoundly significant towards the development of next generation technologies in informatics, computing, software, and cognitive sciences.

- It is recognized that software is not constrained by the **physical laws** and principles discovered in the concrete world. However, software does obey the **laws of informatics**. A set of 19 informatics laws and properties of software has been identified.

- The classic information theory is not about the measure of information rather than its variability or **entropy**. A dilemma in the conventional information theory is that the measurement of the variability or entropy of information is dependent on the receiver's subjective judgment.

- The **most fundamental form of information** that can be represented and processed is ***bit***. Any form of information in the physical (natural) and abstract (mental) worlds can be unified on the basis of bits. This is the informatics foundation of modern digital computers and natural intelligence.

- The development of classical and contemporary informatics, the cross fertilization between computer science, software engineering, cognitive science and neuropsychology, has led to a whole range of new research areas known as **cognitive informatics**.

- **Perceptions on Information:** a) In **classic informatics**, *information* is defined as a probabilistic measure of the variability of messages which can be obtained from a message source. b) In **contemporary informatics**, *information* is defined as any property or attribute of the natural world that can be generally abstracted, quantitatively represented, and mentally processed. c) In **cognitive informatics**, *information* is defined as abstract artifacts and their relations that can be elicited, modeled, represented, stored, and processed by human brains.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Classic Information Theory

• The **classic information theory** is founded by Shannon during 1948-1949, while the term *information* was first adopted by Hartley in 1928, and extensively discussed by Bell and Goldman in 1953.

• Conventional information theory was modeled based on probability theory and statistics. **Information** is defined as a probabilistic measure of the variability of message which can be obtained from a message source.

• **Information** is a weighted probabilistic measure of the variability of messages (signals) that is expected from a message source via a transmission channel.

• **Entropy** is the extent of the trend of a system towards complete disorder or randomization. Entropy is not a physical entity, but a measure of the extent of chaos of a given system, which is closely related to the classic concept of information. The **quantity of information entropy** H_i of a source is determined by the average weighted information variability I transmitted by the source.

• The **information variability**, I_i, of the ith sign in a message is determined by its unexpectedness, i.e., $I_i = log_2 \frac{1}{p_i}$ [bit], where p_i is the probability that the ith sign is transmitted.

• The **total information variability** transmitted by a source or sender, I, is the weighted sum of the probability of all its n possible signs, known as the alphabet, in the message, i.e., $I = \sum_{i=1}^{n} p_i \bullet I_i = - \sum_{i=1}^{n} p_i \bullet \log_2 p_i$ [bit].

• **Thermodynamics** is the branch of physics built upon the fundamental laws obeyed by energy in the forms of heat and work and in their transformation. The three laws of thermodynamics are the basic theory that governs exchanges of energy.

• The **1st law of thermodynamics**, **conservation of energy**, states that energy can be neither created nor destroyed, so that the total input of energy,

E_i, in any transformation must equal the total output of energy, E_o, i.e., $\sum E_i \equiv \sum E_o$.

- The **2nd law of thermodynamics**, the **heat flow statement**, states that: (a) Entropy of the universe ΔH_u does not change when a reversible process occurs, i.e., $\Delta H_u = 0$, and (b) Entropy of the universe ΔH_u increases when an irreversible process occurs, i.e., $\Delta H_u > 0$.

- The **3rd law of thermodynamics**, the **state of maximum order**, states that a perfect crystal at a temperature of absolute zero possesses zero entropy, i.e., $\lim_{T \to 0} H = 0$.

- The **extended 2nd law of thermodynamics** states that in any system, the sum of the information entropy H_i and the thermal entropy H_t is a constant, i.e., $k_t H_t + k_i H_i = \varepsilon$, where $k_t, k_i,$ *and* ε are positive constants for a given system. In other words, the information entropy is also perceived as the *negative entropy.*

- The classic information theory was used to study models of communication channels and coding/decoding systems. Alternative information theories have been developed in the last decades to extend the usage of classical informatics.

Contemporary Informatics

- **Information** in contemporary informatics is defined as any property or attribute of the natural world that can be generally abstracted, quantitatively represented, and mentally processed.

- **Contemporary informatics** perceives that the implication and extension of information has been shifted from the probability of messages to the entity of messages that represents the messages, knowledge and/or abstracted real-world entities. With this new orientation, information is regarded as an independent and essential entity in modeling the natural world, particularly its abstract part.

- The **content of information** in modern informatics is measured by the cost of code to abstractly represent a given size of message M in a digital system based on k [12], i.e., $I_k = f: M \rightarrow S_k = log_k M$, where I_k is the content of information in a k-based digital system, and S_k the measurement scale based on k. The unit of I_k is the number of k-based digits. A **bit** is defined as the measure of information when a binary digital representation system is adopted, i.e., $k = b = 2$.

• In modern informatics, **IT** is a technology that can be used for the processing of information, such as *information acquisition, elicitation, storage, manipulation* (*adding, deleting, updating*), *production, presentation, searching,* and *retrieving*.

• **Cognitive informatics** is the transdisciplinary study of cognitive and information sciences that investigates into the internal information processing mechanisms and processes of the natural intelligence - human brains and minds.

• **Information** in cognitive informatics is defined as abstract artifacts and their relations that can be elicited, modeled, represented, stored, and processed by human brains.

Informatics Laws of Software

• The **informatics laws of software** state that software architectures, behaviors, and processes are constrained by the laws of *informatics* and *mathematics*.

• According to the **theory of special relativity** of Albert Einstein (1879 - 1855), one of the most astonishing results is that mass and energy are equivalent. According to the IME model, the three essences of the world are predicated to be transformable between each other in cognitive informatics.

• A set of 19 **informatics laws** and properties of software has been identified as follows:

1) Abstraction
2) Generality
3) Cumulativeness
4) Dependency on cognition
5) Three-dimensional behavioral space
6) Sharability
7) Dimensionless
8) Weightless
9) Transformability between I-M-E
10) Multiple representation forms
11) Multiple carrying media
12) Multiple transmission forms
13) Dependability on media
14) Dependability on energy

15) Wearless and time dependency
16) Conservation of information entropy and thermal entropy
17) Informatics quality attributes
18) Susceptible to distortion
19) Scarcity

- The **principle of universal constraints** states that both the natural world and the perceived abstract world are constrained by certain known or yet to be known restrictions and laws, due to the limitations of natural resources and/or human cognitive capability.

Information Theories for Software Engineering

- According to the cognitive information model, **software** can be perceived as a kind of coded and **instructive information** that describes the algebraic process logic of software system architectures and behaviors in computing.

- The **internal quality attributes** of software systems are such as *completeness, no misinterpretation, consistency, exactness*, *no confliction, feasibility*, and *verifiability*. The **external quality attributes** of software system are such as *functionality, usability, availability, reliability, efficiency, portability,* and *maintainability*.

- The **nature of software** is its *instructive characteristics* and the *information-* and *mathematics-based metaphors*.

Questions and Research Opportunities

7.1 Referring to the *Information-Matter-Energy* model (IME, Theorem 1.1), discuss why information plays an important role in modeling software properties, behaviors, and software engineering theories and methodologies.

7.2 Why is software not constrained by any physical law and principle known in the concrete world? What is the impact of this discovery to software engineering theories and methodologies?

7.3 What is the relationship between information in classic informatics and entropy in physics?

7.4 Comparing Corollaries 7.1 and 7.2, discuss how the *extended 2nd law of thermodynamics* integrates information entropy and thermal entropy into a coherent framework.

7.5 What are the perceptions of information in contemporary informatics and in the information, computer, and software industry?

7.6 What is the perception of internal information in the brain as modeled in cognitive informatics?

7.7 What is the role of information in mankind evolution?

7.8 Why is it impossible to directly transfer acquired information and knowledge of a person on to the next generation or peers? What would be the indirect approaches to do so?

7.9 What is the set of perceived *transformability among I-M-E*? Which pairs of transformations have been formally proven? Why will natural or machine intelligence play an important role in searching the remainder potential transformability?

7.10 Try to summarize the 19 fundamental properties of information in a structured framework with categorizations.

7.11 How to prove informatics Property 8 – information is weightless?

7.12 The three dependencies of information, Property 4 – dependency on cognition, Property 13 – dependency on media, and Property 14 – dependency on energy, reveal the relationships between information and human brain, hardware, and energy. Discuss if these three dependencies of information are always necessary in software engineering or there are exceptional options.

7.13 The laws of informatics that are applicable to software and software engineering are identified as the 19 basic properties of informatics. Based on the informatics laws and properties, try to discuss the following:

(a) Doesn't software obey any of the above information laws?

(b) Does software obey any informatics laws (or basic properties) that were not identified in the given set?

(c) Does software obey any physical laws of the world?

7.14 What does the unit of information, *bit*, mean in classic, contemporary, and cognitive informatics?

7.15 Theorem 7.1 states that the *primitive form of information* is the bit. Explain why this view forms a foundation to integrate both computer science and information science into a common framework.

7.16 Comparing the three definitions of information in classic, contemporary, and cognitive informatics, try to analyze the evolution on intensions and extensions of the concept information and their applications in the IT industry.

7.17 Summarize how information is measured in classical, contemporary, and cognitive informatics.

7.18 Assuming the package of information is $X = 2^{40}$ bytes, answer the following questions:

(a) How many bits do you need to represent the information, X, in a computer in parallel?

(b) If a neural cell is equivalent to 2^{10} bits in parallel, how many neural cells do you need to represent X in the brain?

7.19 What are the differences between the external and internal attributes of software? What are the information-oriented internal quality attributes of software?

7.20 Read the following chapter in information systems:

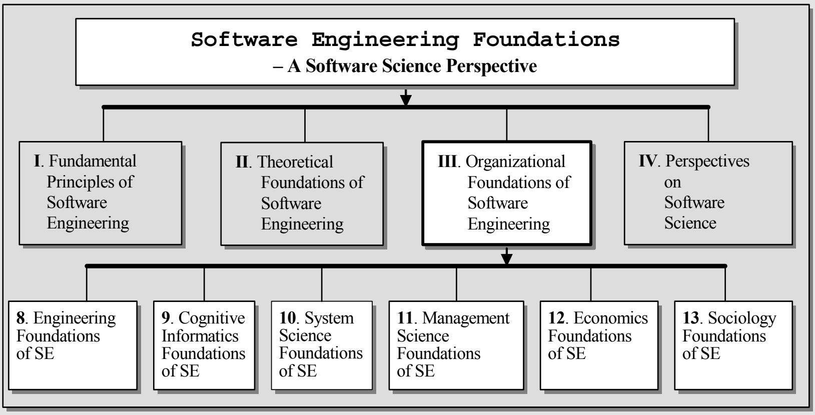
Robert G. Murdick (1986), Chapter 5, Data, Information, and Communication, MIS Concepts and Design, 2nd ed., pp. 140-177.

Discuss the following topics in a group or individually:

- About the author.
- What are the attributes and measure of information?
- What are the relationships between information, data, and software?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

PART III

ORGANIZATIONAL FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Fundamental Principles of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE

Organizational foundations of software engineering incorporate multi-facet principles, transdisciplinary theories, and empirical knowledge for software engineering. Part III attempts to explore the organization and system metaphors toward software engineering. Three main threads are adopted in this part known as *system science*, *cognitive informatics*, and *organizational theories* at different levels in the domains of engineering science, management science, economics, and sociology.

It is recognized in this part that the hidden reasons caused so many failures of large-scale software engineering projects are neither purely technical ones nor unsatisfied programming skills, but mainly because of both the organizational deficiency of nonoptimal labor allocation and the incorrect sequence of interlocked labor-duration-cost determination in coordinative work organization.

The knowledge structure of Part III on *Organizational Foundations of Software Engineering* is as follows:

- Chapter 8. Engineering Foundations of Software Engineering
- Chapter 9. Cognitive Informatics Foundations of Software Engineering
- Chapter 10. System Science Foundations of Software Engineering
- Chapter 11. Management Science Foundations of Software Engineering
- Chapter 12. Economics Foundations of Software Engineering
- Chapter 13. Sociology Foundations of Software Engineering

This part addresses the organizational and cognitive theories and methodologies of software engineering in a transdisciplinary approach. The structural organization of software engineering perceives software as abstract systems. The fundamental view towards software engineering perceives software as a set of cognitive and intelligent behaviors. With system science theories as an overarching framework, the organizational theories for software engineering form a hierarchical structure in this part covering multidisciplinary foundations of engineering science, management science, economics, and sociology from the bottom-up.

Chapter 8, *Engineering Foundations of Software Engineering*, presents the generic engineering principles and their applications in software engineering. Engineering is deemed as an important organizational methodology that emerged during the industrial revolutions. It helps to understand the nature, status, and problems of software engineering, as well as its future trends, based on comparative studies between the generic engineering principles and current software engineering practices. Key empirical knowledge and methodologies that may be learned from other matured engineering disciplines are discussed. Basic engineering principles commonly shared in most engineering disciplines are elicited on engineering objectives, organization, technology, professionalism, and domain characteristics. A comprehensive set of engineering principle for software engineering, such as engineering characteristics, division of labor, and professionalism for software engineering is derived. A formal coordinative work organization theory for engineering science in general and software engineering in particular is developed that reveals how software engineering projects may be optimally organized. Empirical methodologies for software engineering, such as case studies, experiments, trials, benchmarking, and standardization, are reviewed.

Chapter 9, *Cognitive Informatics Foundations of Software Engineering*, introduces a new transdisciplinary field that studies the internal information processing mechanisms of the brain and their engineering applications. Large-scale software systems are highly complicated systems that mankind has ever handled or experienced before. Software is a unique abstract artifact that does not obey any known physical laws. However, it is recognized that software should be constrained by the laws of cognitive informatics, mathematics, and systems as explored in this book. Theories of cognitive informatics and its potential impacts on, and applications in, information-based sciences and engineering disciplines, particularly in software engineering, are discussed. Cognitive informatics models of the brain, such as the Layered Reference Model of the Brain (LRMB), the cognitive models of memories, and the cognitive model of natural intelligence, are developed. The cognitive model of internal information presentation in the brain, particularly the Object-Attribute-Relation (OAR) model, is presented. This chapter describes the cognitive informatics and intelligent behavioral metaphor of software and software engineering. The cognitive informatics foundations that address the cognitive constraints of software engineering are described, which lead to the understanding and formal measurement of the cognitive complexity of software systems.

Chapter 10, *System Science Foundations of Software Engineering*, provides a powerful means for dealing with complicated objects and phenomena in software engineering. Treating software engineering and large-scale software projects via system engineering is also a promising trend in dealing with the problems, complexities, quality assurances, and human factors in software engineering. This chapter describes the system metaphor of software and software engineering, and explores theories of systems science, underlying principles, and modeling techniques of systems engineering. The classic system philosophies and system topology are described. A new mathematical structure of abstract systems known as system algebra is developed. Principles of system theories, such as generic architectures, equilibrium, synchronization, and dissimilation, are formally and rigorously treated. Applications of system theories and system engineering techniques in software engineering are described with formalized software system models and formal explanations of many important software engineering phenomena as system engineering issues.

Chapter 11, *Management Science Foundations of Software Engineering*, studies organizational behaviors, executive decision making, and resource optimization on given internal and external constraints in software engineering. Historically, software engineering has focused on programming methodologies, programming languages, and software development models. One of the critical areas to software engineering – organizational and management infrastructures – has been largely ignored. Management science foundations for software engineering encompass management principles, classical management thought, decision theories, and quality system theories beyond programming and technical aspects of software development. A set of organizational theorems and laws are formally derived. A theoretical framework of decision theories is developed with the mathematical models of decisions, the cognitive process of decision making, formal description of decision strategies, the extended game theories, and decision grid theory for a series of dynamic decision making. Quality systems are presented focusing on quality principles, quality assurance, and quality management systems. Then, the concept and methodology of process-based software engineering is developed in order to deal with complicated management issues in software engineering.

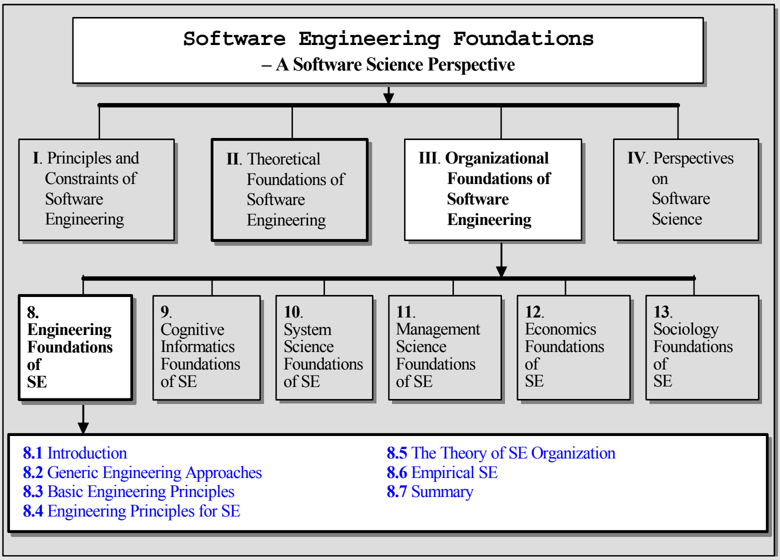
Chapter 12, *Economics Foundations of Software Engineering*, studies how resources are efficiently used to develop software and how services are provided in software engineering. This chapter introduces fundamental principles and methodologies utilized in engineering economics and their applications in software engineering. It also introduces formal methodology into economic analysis and modeling. A set of formal economic models such as the production, costs, and market models is developed based on fundamental principles of microeconomics. The dynamic values of money and assets, as well as their patterns in cash flows, are formally modeled. Economic analysis methodologies for engineering decisions such as project costs, benefit-cost ratio, payback period, and rate of return are rigorously described. On the basis of the formal treatment of economic theories and principles, software engineering economics is presented on elements of software costs, software engineering project costs estimation, economic analyses of software engineering projects, and the software legacy maintenance cost model, which leads to the finding of the important phenomenon known as software maintenance crisis in software engineering.

Chapter 13, *Sociology Foundations of Software Engineering*, investigates how a software engineering environment may be organized efficiently and effectively on certain group and social constraints. This chapter completes the final piece of the puzzle of the systematic theory on *coordinative work organization* at the highest level of scopes – the society level. It forms an important methodology for optimal allocation of labor, resources, and schedules for a given workload in a society in general, and in a software engineering context in particular. This chapter presents a formal treatment of the sociological theories, models, and their applications in software engineering. Fundamental principles of sociology are reviewed, which covers social structures, social behaviors, and social norms. Social psychology such as fundamental human traits, collective behaviors, and the perceptual influence on them are described, which form the underlying theory for explaining the human factor in engineering systems and societies. Theories of social organization that provide an essential understanding for coordinative work organization at various levels of societies are presented. A formal model of social organization is developed based on a new mathematical model known as the *complete organization tree*. Sociology for software engineering is explored on social environment of software engineering, ergonomics for software engineering, and human factors in terms of human strengths, weaknesses, and uncertainty in the context of software engineering. The theoretical foundation of quality assurance in programming and software engineering is developed.

Part III will establish the organizational foundations of software engineering with engineering science, cognitive informatics, and system science at various scopes such as management science, economics, and sociology. Supplemented to Part II, this part will reveal that important aspects of software engineering theories are the organizational and cognitive theories. It will demonstrate that the profound causes that result in all the failures in software engineering history are not only pure technical reasons, but also organizational reasons due to the limitations of human cognitive capability.

Chapter 8

ENGINEERING FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE
8.1 Introduction
8.2 Generic Engineering Approaches
8.3 Basic Engineering Principles
8.4 Engineering Principles for SE
8.5 The Theory of SE Organization
8.6 Empirical SE
8.7 Summary

8. Engineering Foundations of SE

Knowledge Structure

❍ Generic engineering approaches

- Engineering: a concept emerged from the industrial revolutions
- Science and generic scientific method
- Engineering vs. sciences
- Fundamental goals and constraints of engineering
- Generic engineering approaches
- The generic engineering maturity model (GEMM)

❍ Basic engineering principles

- Principles of engineering organization
- Principles of engineering management
- Principles of engineering technology
- Principles of engineering professionalism

❍ Engineering principles for software engineering

- The engineering characteristics of SE
- Characteristics of SE in the engineering age
- Professionalism of SE
- Division of labor
- Unique principles of SE

❍ The theory of software engineering organization

- The characteristics of coordinative work in engineering
- Laws of work organization in SE
- The mythical man-month explained
- Decision optimization in SE

❍ Empirical software engineering

- SE case studies
- SE trials
- SE standardization
- SE experiments
- SE benchmarking

Learning Objectives

- To gain generic engineering principles.
- To know generic engineering approaches.
- To understand the essences of interpersonal coordination in engineering and the interchangeability between labor and time.
- To understand the laws of abstract work organization for engineering projects.
- To be able to apply the fundamental theories of engineering organization in software engineering.
- To be able to apply the empirical software engineering methods such as case studies, experiments, trials, benchmarking, and using standards.

"Software engineers are not just good programmers."

David L. Parnas (1998)

"Large software systems are among the most complex systems engineered by man."

J.V. Guttag (2002)

"It is this application of expertise that causes the fields of both bridge building and software development to be areas where engineering principles are applicable. It is the cost of failure that makes these principles required."

R. Gisselquist (1998)

8.1 Introduction

Engineering is a set of applied scientific disciplines seeking solutions for complicated problems and systems that could not be done by individuals. The aim of engineering is to repetitively produce complicated artifacts in an efficient way on the basis of scientific theories and principles.

The etymology of the words *engineer* and *ingenuity* comes from the same Latin root, *ingenium*, which means talent, genius, cleverness, or native ability. Thus, engineering may be perceived as an approach to both *problem solving* and *industrial organization*.

Engineering may also be deemed as a *profession*. The Accreditation Board for Engineering and Technology (ABET) defines engineering as "the profession in which a knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to economically utilize the materials and forces of nature for the benefit of mankind [ABET, 1986]."

The basic task of scientists is to perform *research* that creates new knowledge, while the basic task of engineers is to perform *design* and *development* that result in new applications and products. Therefore, engineering is a *methodology* and *process* that converts theoretical concepts into useful applications to satisfy human needs.

Definition 8.1 *Engineering* is a technological and organizational methodology and approach by which human beings can repetitively plan, design, develop, produce, maintain, and/or use complicated artefacts, in rigorous, systematic, efficient, and refining processes, that cannot be done by individuals.

Software engineering is a discipline that adopts engineering approaches to develop large-scale software with high productivity, low cost, controllable quality, and measurable development schedules. It is recognized that: "Large software systems are among the most complex systems engineered by man [Guttag, 2002]." Therefore, engineering approaches and generic engineering principles form an important part of the basic theoretical and empirical foundations of software engineering. However, the *engineering metaphor* in the term software engineering was vague from the very beginning, and it has rarely been well defined and explained in the literature.

This chapter attempts to explore a set of generic engineering principles and their applications in software engineering, particularly the theory of coordinative work organization. It helps to understand the nature, status, and problems of software engineering, as well as its future trends, based on comparative studies between the generic engineering principles and current software engineering practices. Key empirical knowledge and methodologies that may be learned from the generic engineering principles will be discussed. Engineering principles for software engineering will be elicited on engineering objectives, organization, technology, professionalism, and domain characteristics.

In the remainder of this chapter, the engineering foundations of software engineering will be presented as follows. Section 8.2 explores generic engineering approaches that may be learnt by software engineering. Section 8.3 discusses basic engineering principles commonly shared in most engineering disciplines. Section 8.4 describes the engineering principle for software engineering, such as engineering characteristics, division of labor, and professionalism for software engineering. Section 8.5 creates the coordinative work organizational theory for software engineering that reveals how software engineering projects may be optimally organized and what are the major reasons of historical failures in software engineering. Section 8.6 reviews empirical methodologies for software engineering such as case studies, experiments, trials, benchmarking, and standardization.

8.2 Generic Engineering Approaches

The advances in sciences and the increases of economical demands led to the industrial revolutions during the 18th and 19th centuries. The wealth created by standardized and mass-produced products encouraged the exponential growth in engineering in the 20th century. Then, the computer revolution brought the world entirely new fields of engineering, such as computer engineering, software engineering, information engineering, knowledge engineering, and intelligent engineering.

In order to explain the fundamental engineering principles of software engineering, let us review the historical development of matured engineering disciplines and elicit their common approaches and methodologies. This can be done in the following subsections by answering the following questions: a) What are the differences between science and engineering? b) What are the generic approaches to engineering? c) How have various engineering disciplines matured in history? and d) What are the generic principles of engineering?

8.2.1 ENGINEERING: A CONCEPT EMERGED FROM THE INDUSTRIAL REVOLUTIONS

Engineering is a concept of industrial organization emerged from the industrial revolutions [Ure, 1835; Soanes and Stevenson, 2003]. The industrial revolutions were a time of drastic change and transformation from hand tools and hand made items to machine manufactured and mass produced goods.

The first industrial revolution begun in England in the 1730s emerged from inventions and technology innovations in cotton weaving. Before the first industrial revolution in the early 18th century, England's economy was based on its cottage industry. Workers would buy raw materials from merchants and take it back to their cottages in order to produce goods at homes. It was usually owned and managed by one person or a family. Since the productivity of the cottage workers was low, goods were high in price and exclusive only to the wealthy people. Under the increasing demand for cotton cloth, the flying shuttle was invented in 1733 that resulted in the reduction of weaving time in half. This invention triggered the first industrial revolution. Inventions such as the spinning jenny and the water-powered frame helped the manufacture of cotton goods by dramatically improved productivity with machinery and mass production. In this way, the cottage industry had been inevitably replaced by the factory system. Mass production made usually expensive items affordable by less wealthy people. Therefore, the quality of life had been improved. In the meantime, steam engines were invented that provide stronger power than horses and enable a faster mode of transportation for people and resources.

The second industrial revolution in the beginning of the 19th century proved more drastic, not only in inventions, but also in social and organizational reforms as life with machinery had already been assimilated into society. The second industrial revolution utilized the power of electricity based on Michael Faraday's invention. Electricity improved life by supplying people with light as well as electricity to power machines. Communications have been improved by the telephone and telegraph. Radio waves were discovered that enabled messages to be sent over long distances in virtually no time. During the 1800's, over 70,000 chemical compounds were analyzed, and petroleum begun to be widely used as an alternate energy source. As a result, steam engines were replaced by the internal combustion engines. This allowed a person to own a car instead of using public transportation. Then, the airplane industry was born following the first flight of man-made aircraft by Orville and Wilbur Wright.

As the first industrial revolution was centered by machinery, power, and mass production, the second industrial revolution was characterized by electricity, transportation, and the internal combustion engine. Both industrial revolutions brought on more technology, power, and wealth followed by the third industrial revolution in the 20th century of computer and information, which is still going on and has already dramatically and fundamentally extended human's capability, reachability, and cognitive power.

The characteristics of the industrial revolutions, such as machinery, mass production, energy, power systems, high-speed transportation, and telecommunications, were the cradle of the concept of engineering. As Andrew Ure (1835) observed in *The Philosophy of Manufactures*, the improvements in machinery in the industrial revolutions have a three-fold bearing:

- "They make it possible to fabricate some products which, but for them, could not be fabricated at all.
- "They enable an operative to turn out a greater quantity of work than he could before – time, labor, and quality of work remaining constant.
- "They effect a substitution of labor comparatively unskilled, for that which is more skilled.

Ure's observation revealed the great achievement of the engineering approach to industrialization that resulted in extended human capability, improved productivity, and reduced skill requirement.

The impacts of industrial revolutions and the industrialization of the economies can be described from the following five aspects [Macionis et al., 1997]:

- New forms of energy
- The centralization of work in factories
- Manufacturing and mass production

- Specialization and division of labor
- Wage labor and management as profession

More generically, the industrial revolutions extended human *physical capability* by machines and power engines, while the new information revolution is focused on the extension of human *intelligence*, *memory*, and the capacity for *information processing* by computers, communication networks, and robots.

Contrary to the traditional individual-based production process, engineering is a methodology for enabling a group of people to work together to produce a complex product, or achieve a common goal, which could not be reached by individuals physically, technically, and/or economically. Therefore, the essence of engineering is the organizational methodology for enabling team work. Further details of this important concept will be discussed in Chapters 11 and 13 on management science and sociology foundations of software engineering.

8.2.2 SCIENCE AND THE GENERIC SCIENTIFIC METHOD

Science is both an organized system for the systematic study of particular aspects of the dual world known as the natural and abstract world, and a process of inquiry for generating a body of knowledge towards them. Therefore, science is not only a set of systematic and formulated knowledge about the dual world, but also a generic method to explore it.

Definition 8.2 *Science* is a systematic cognitive methodology for exploring and explaining the nature, for cumulating and organizing the knowledge obtained about it, and for applying the knowledge in solving engineering and technological problems.

Science stresses an *objective approach* to the phenomena being studied, and scientific questions about nature tend to emphasizes *how* things occur rather than *why* they occur. It involves the application of the scientific method to problems formulated by trained minds in particular disciplines [Kuhn, 1970].

In a formal sense, the *scientific method*, according to Francis Bacon (1561 - 1626), can be generically described below.

Definition 8.3 The *generic scientific method* refers to the model for research that involves the following sequence:

a. Identifying the problem

b. Collecting data within the problem area (by observations, measurements, etc.)

c. Sifting the data for correlations, meaningful connections, and regularities

d. Formulating a hypothesis (a generalization), which is an educated guess that explains the existing data and suggests further avenues of investigation

e. Testing the hypothesis rigorously by gathering new data

f. Confirming, modifying, or rejecting the hypothesis in light of the new findings

Scientists may be interested in different aspects of nature, but they use a similar intellectual approach to guide their investigation. Scientists must first formulate a problem to which they can then seek an answer. The answer generally involves an explanation relating to order or process in nature. The scientist is primarily interested in the mechanisms by which nature works rather than in questions of ultimate purpose.

Once a question has been raised, the scientist seeks answers by collecting data relevant to the problem. The data, which may consist of direct observations and measurements, or derived results, are carefully sifted for regularity and relationships. An educated guess, called a hypothesis, is then drawn up in order to place the data into a conceptual framework.

The hypothesis makes up the lattice-work upon which scientific understanding is structured. The hypothesis constitutes a generalization that describes the state of affairs within an area of investigation. Inductive logic is used to formulate a hypothesis (a generalization) from the particulars (specific) of the data. Since the scientific method involves such an inductive process at its very core, it is often described as the inductive method.

A hypothesis must be both logical and testable. It is tested by constructing experiments and gathering new data, which in the end will either support or refute the hypothesis. An experiment must be reproducible, which means that other scientists must be able to repeat the experiment and get the same results. Once the experiments have been completed, the results must be weighted to see if the hypothesis should be accepted, modified, or rejected. Although scientists are very inquisitive and highly creative in the thought processes, their curiosity may be constrained by previous, long-accepted views.

Definition 8.4 The criteria that constitute a good *hypothesis*, H_g, can be defined as a 5-tuple with *causality* (C), *originality* (O), *generality* (G), *predictability* (P), and *falsifiability* (F), i.e.:

$$H_g \triangleq (C, O, G, P, F) \tag{8.1}$$

where

(a) C = *true* states that the hypothesis must be able to explain the causal relationship of existing data or observed phenomena.

(b) O = *true* states that the hypothesis must be able to create a new relationship between two or more entities or phenomena.

(c) G = *true* states that the hypothesis must be able to explain a set of similar phenomena rather than an individual instance.

(d) P = *true* states that the hypothesis must be able to predicate new phenomena and their consequences on the basis of the defined causality.

(e) F = *true* states that the predictability and causality of the hypothesis must be able to be proven true or false.

Formal proofs and/or repeated confirmations of a hypothesis against the five criteria as defined in Definition 8.4 elevate it to the status of a theory.

The 20th Principle of Software Engineering

Theorem 8.1 The *relationship between a hypothesis and a theory* states that the necessary and sufficient conditions for a hypothesis $H_g(C, O, G, P, F)$ to be proven as a theory $\mathcal{T}$ are *iff* it fulfills the following criteria, i.e.:

$$H_g \vdash \mathcal{T},\ iff\ C \wedge O \wedge G \wedge P \wedge F = \mathbf{T} \tag{8.2}$$

When a fundamental theory has been formally proven and/or repeatedly confirmed over a long period of time, it may be accepted as a scientific *law*. When a hypothesis is substantially contradicted by new findings, it is rejected to make way for new hypotheses.

It is noteworthy that a hypothesis which withstands the rigor of present tests may have to be altered in the light of future evidences. In other words, a theory, or a proven hypothesis, is a relatively true explanation of a given set of phenomena in a given space and time. Absolutely true theories may only exist in the results of philosophy and mathematics.

8.2.3 ENGINEERING VS. SCIENCE

In a speech, Richard Feynman (1963) perceived that "Science means, sometimes, a special method of finding things out. Sometimes it means the body of knowledge arising from the things found out. It may also mean the new things you can do when you have found something out, or the actual doing of new things. This last field is usually called technology [Feynman and Brown, 2000]."

In the view of system philosophy, there is neither number two in sciences nor number one in engineering. The former is true because sciences are aimed at advancing the knowledge, where no reinvention or rediscovery is recognized. The latter is true because both engineering design and implementation are characterized by the polymorphism in a large solution space. It is impossible to prove if a given engineering solution is the best or optimistic both technically and economically. This is particularly true in software engineering.

8.2.3.1 Science and Scientists

Science is a process of inquiry for generating a body of knowledge. All sciences are characterized by a common method as a logic of inquiry in their quest for knowledge, such as tenacity, intuition, reference, rationalism, and empiricism.

W.I. Beveridge observed that "The curiosity of the scientist is usually directed toward seeking an understanding of things or relationships which the notices have no satisfactory explanations. Explanations usually consist in connecting new observations or ideas to accepted facts or ideas. An explanation may be a generalization which ties together a bundle of data into an orderly whole that can be connected up with current knowledge and beliefs. That strong desire scientists usually have to seek underlying principles in masses of data not obviously related may be regarded as an adult form or sublimation of curiosity [Beveridge, 1957]." Further, Beveridge wrote: "Scientists are cautious and conservative individuals, recognizing that most phenomena are multidetermined and that new evidence may necessitate replacing an old explanation with a better one."

Christensen pointed out that the objectives of science are description, explanation, prediction, and control [Christensen, 1997]:

- *Description:* The portrayal of a phenomenon, fact, or mechanism by identifying variables, constants, and their relations and constraints.
- *Explanation:* The determination of the cause of a given phenomenon that answers why the phenomenon exists and its occurring conditions.
- *Prediction:* The ability to anticipate the occurrence of an event based on the described knowledge and explained conditions.

- *Control:* The manipulation of the conditions that determine a phenomenon and the elimination of the influence of extraneous conditions. Controlled inquiry is an absolutely essential process in science because without it the cause of an effect could not be isolated.

Scientists are professionals who investigate the universe around us and invent new ways of using its resources. All scientific work is carried out systematically and originally. Scientific approaches include experiment, observation, testing, exploration, classification, measurement, equipment, modeling, and development of theories [Beveridge, 1957; Sober, 1995].

8.2.3.2 Engineering and Engineers

Referring to Definition 8.1, engineering is an organizational approach by which human beings can repetitively plan, design, develop, produce, maintain, and/or use complicated artifacts in a rigorous, systematic, and refining process. Thus, an *engineer* is a professional who has a disciplined role with required skills in an engineering branch.

A. Eide and his colleagues thought [Eide et al., 1979] that "Both the engineer and scientist are thoroughly educated in the mathematical and natural sciences, but the scientist primarily uses this knowledge to acquire new knowledge, whereas the engineer applies the knowledge to design and develop usable devices, structures, and processes. In other words, the scientist seeks to know, the engineer aims to do." In summary, scientists explore what is; engineers find out how to do.

Definition 8.5 An *engineer* is a professional who is regulated and experienced to practise engineering by using science, mathematics, and technology for creative design and implementation of applications, products, systems, and processes.

In the Computing Curricula – Software Engineering (CCSE), IEEE/ACM identify the following characteristics of engineers in general [IEEE/ACM, 2001/03; Wang, 2005h]:

- "Engineers proceed a task by making a series of decisions, carefully evaluating options, and choosing an approach at each decision-point that is appropriate for the current task in the current context. Appropriateness can be judged by tradeoff analysis, which balances costs against benefits.
- "Engineers measure things, and when appropriate, work quantitatively; they calibrate and validate their measurements; and they use approximations based on experience and empirical data.

- "Engineers emphasize the use of a disciplined process when creating a design.
- "Engineers can have multiple roles: research, development, design, production, testing, construction, operations, management, and others such as sales, consulting, and teaching.
- "Engineers use tools to apply process systematically. Therefore, the choice and use of appropriate tools is key to engineering.
- "Engineering disciplines advance by the development and validation of principles, standards, and best practices.
- "Engineers reuse designs and design artifacts."

According to Definitions 8.1 and 8.5, it can be seen that the domain of engineering and engineers is as broad as the demand of mankind. Engineers are trained to think in analytical and objective terms and to approach problems methodically and systematically. As Michel Sintzoff stated "The clear and structured representation of design knowledge and reasoning, on an industrial scale and for various domains, helps to solve the problem of generality in software engineering, which is akin to that of generality in artificial intelligence [Sintzoff, 1989]."

8.2.3.3 Relationship between Science and Engineering

The relationship between science and engineering can be analyzed from the aspects of the disciplines and the professions as shown in Fig. 8.1. The disciplines of science and engineering can be contrasted by their domains, although there is no clear cut way. In general, science transfers information about nature into knowledge and theories; while engineering embodies knowledge into methodologies and products. The differences between science and engineering in terms of objectives, methodologies, criteria, and embodied results are summarized in Table 8.1.

Table 8.1
Characteristics of Engineering and Science

Discipline	Major objectives	Basic methodology	Criteria	Embodied results
Science	Knowledge	Inductive	Originality	Information, knowledge, and methodologies
Engineering	Products, skills, and applications	Deductive	Utility and efficiency	Technologies, products, and know-how

The roles of scientists and engineers as different professions are also illustrated in Fig. 8.1. Scientists work on scientific research, and engineers apply scientific theories into industry by engineering development and by finding the ways for suitable mass production. Although there are overlaps between the roles of scientists and engineers, it is obvious that an engineer is not responsible for developing fundamental theories, and a scientist is not directly involved in manufacturing a product.

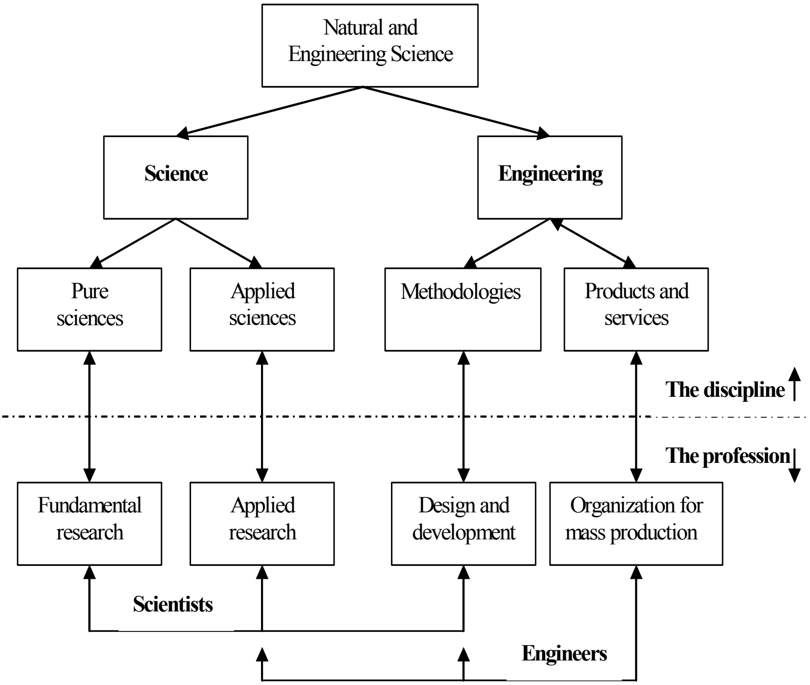


Figure 8.1 Engineering vs. science

8.2.4 FUNDAMENTAL GOALS AND CONSTRAINTS OF ENGINEERING

Science and engineering disciplines share a number of common goals in their pursuits as Steven Weinberg, the 1979 Nobel Prize laureate, expressed: “Our job in physics is to see things simply, to understand a great many complicated phenomena, in terms of a few simple principles.” Science pursues *originality, generality,* and *simplicity* in principles and theories. In addition to the goals of science, engineering seeks *efficiency*, *productivity*, and *quality* in implementation of scientific principles and theories into repetitive and mass production.

The three generic engineering goals can be described by a triangular Engineering Objective Model (EOM) as shown in Fig. 8.2 [Wang, 2006a]. In the EOM model, each of the three generic goals, efficiency, productivity, and quality, obeys a basic constraint for engineering organization and practice in terms of *costs*, *time*, and *utility*, respectively. It is found, unfortunately, the three basic goals in engineering are interlocked.

In the EOM model, productivity is the principal objective and major purpose of any engineering discipline. The improvement of productivity is the key to achieve other engineering goals by technical innovation, i.e., by increasing δ as described in Eq. 8.2. For example, the innovation of automatic exchangers and stored program-controlled switching systems in the telecommunication industry in the 1940s and the 1990s have dramatically improved telephone handling capability, which eventually solved the telephone traffic crisis if it would still be handled manually. Therefore, it is inevitable that software engineering should set its paramount goal on the improvement of productivity by using automatic tools for software code generation.

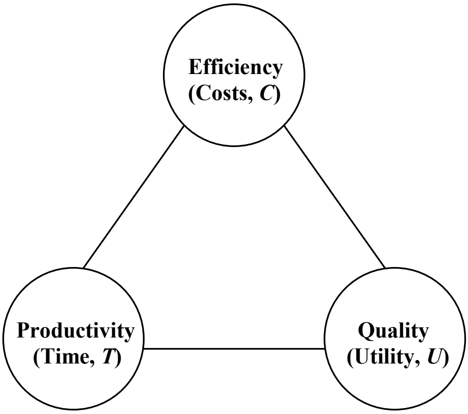


Figure 8.2 The engineering objective model (EOM)

The EMO model as shown in Fig. 8.2 demonstrates that the three basic engineering goals cannot be achieved at the same time in a given engineering context. That is, any goal among the three may be achieved in the costs of the remainder. A formal treatment of this observation is described in the following theorem.

The 22nd Law of Software Engineering

Theorem 8.2 The *conservation of basic engineering constraints* states that the three basic constraints of engineering goals known as time (*T*), costs (*C*), and utility (*U*) are conservative in a given engineering context, i.e.:

$$f_t(T^{-1}) + f_c(C^{-1}) + f_u(U) \\ = k\frac{U}{T \bullet C} = \delta \qquad (8.3)$$

where both δ and *k* are a constant.

According to Theorem 8.2, the following conclusions may be predicted for a given engineering project:

a) The shortening of time (*T*) and reduction of costs (*C*) will sacrifice the quality of expected result or its utility (*U*).

b) The reduction of time (*T*) and requiring of better result (*U*) will increase costs (*C*).

c) The reduction of costs (*C*) and requiring of better quality (*U*) will increase the production time (*T*).

In the EOM model, quality is a complicated term that will be formally modelled and analyzed in Section 11.4.1.

Because software engineering is a specific branch of engineering disciplines, it obeys the same generic engineering rules as stated in Theorem 8.2. The most fundamental categories of goals for software engineering are still productivity, efficiency, and quality, although there are various goals identified such as improve customer satisfaction, ensure quality, shorten time to market, decreasing costs and effort, improve process capability, enhance reliability/dependability/code stability, provide better services, minimum defects, estimate project accurately, and provide better maintainability.

8.2.5 GENERIC ENGINEERING APPROACHES

As we discussed in Section 8.2.1, the engineering approaches and disciplines emerged during the industrial revolutions. Before the industrial revolutions, people produced goods as craftsmen in small or limited scales and they learnt things by doing. For improving productivity as well as quality, and for lowering the requirements for skills in mass production, a generic industrial engineering approach [Eide et al, 1979; Wang and Patel, 2000] had been formed as outlined below:

- To identify repeatable work processes;
- To identify standard and reusable components of products;
- To adopt division of labor (people are specialized in a defined role in processes);
- To equip specialized tools for the roles and processes;
- To recognize management as a profession for organization of the processes and for coordination of the roles.

The above key steps of a generic engineering approach form the basis of almost all existing engineering disciplines. Historically, every engineering discipline in the modern industries has been developed and matured in the same approach: first a kind of art, then a discipline of engineering.

For instance, in the early 19th century, watches were produced manually. Therefore, there were no identical watches. At this stage, the watchmakers were characterized as craftsmen rather than engineers. This resulted in low productivity and high price, and one would perhaps need to find the original watchmaker in order to have a watch fixed. In the middle and late 19th century, the industrial revolutions addressed some of the problems and introduced the approach to engineering. As a result in the watch manufacturing industry, watches could be mass produced by machines, and all watches were identical so that parts were interchangeable among watches of the same brand. At this stage, the traditional watch-smiths had become engineers who were responsible and skilled for one or limited production processes in watch manufacturing.

The generic industrial approach to engineering is also applicable to software engineering, although it has often been ignored in research and practice.

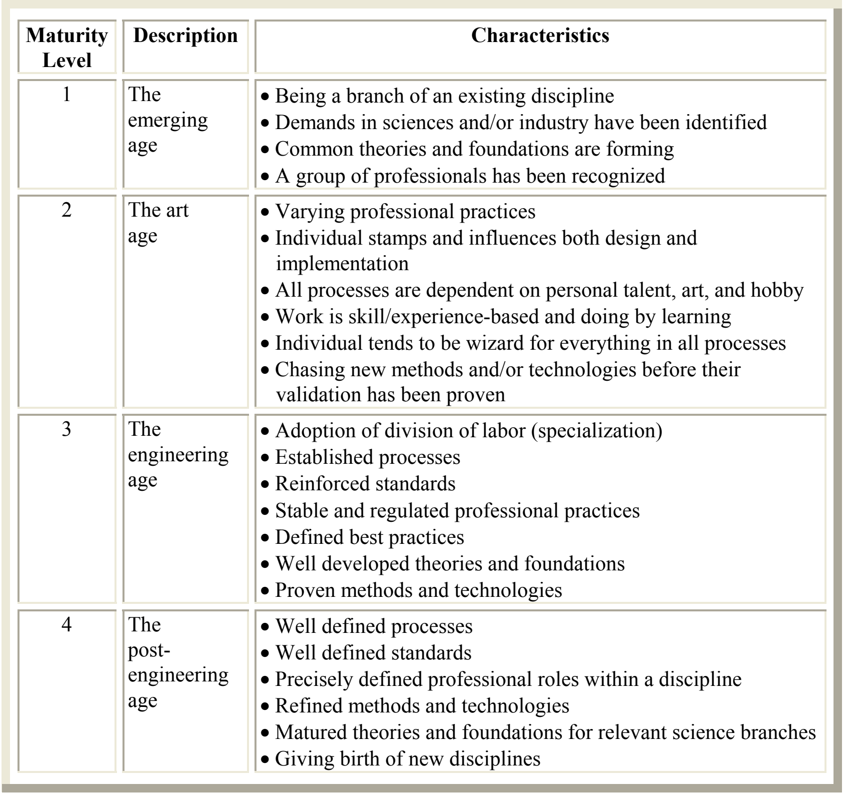
8.2.6 THE GENERIC ENGINEERING MATURITY MODEL (EMM)

Malcolm Gregory (1971) pointed out: “By examining the roots of engineering, we are able to ensure the broad flow of history and to view the present as a part of that flow. This helps us to put the present in its context and to take a better view of our goals, aspirations, and actions.”

In order to answer how the existing engineering discipline matured, a generic engineering maturity mode may be created from the history of industrialization by comparing the differences between an engineer and a craftsman over time. Looking at the time dimension, engineering is a discipline matured from arts of craftsmen in terms of scale and rigor. When asking how the industrial revolutions had changed the traditional individual or family watchmakers, history tells us that the manual watchmakers had disappeared except a few that existed as a special profession. Similar evolution traces may be found in other traditional engineering disciplines.

The history indicates a universal engineering maturity model of all engineering disciplines [Wang and Patel, 2000] as shown in Table 8.2, where the key characteristics of each maturity level have been identified. Based on the definitions in Table 8.2, the following theorem can be derived.

Table 8.2
The Engineering Maturity Model (EMM)



Maturity Level	Description	Characteristics
1	The emerging age	• Being a branch of an existing discipline • Demands in sciences and/or industry have been identified • Common theories and foundations are forming • A group of professionals has been recognized
2	The art age	• Varying professional practices • Individual stamps and influences both design and implementation • All processes are dependent on personal talent, art, and hobby • Work is skill/experience-based and doing by learning • Individual tends to be wizard for everything in all processes • Chasing new methods and/or technologies before their validation has been proven
3	The engineering age	• Adoption of division of labor (specialization) • Established processes • Reinforced standards • Stable and regulated professional practices • Defined best practices • Well developed theories and foundations • Proven methods and technologies
4	The post-engineering age	• Well defined processes • Well defined standards • Precisely defined professional roles within a discipline • Refined methods and technologies • Matured theories and foundations for relevant science branches • Giving birth of new disciplines

The 21st Principle of Software Engineering

Theorem 8.3 The *Engineering Maturity Model* (EMM) states that the applied engineering disciplines have four maturity levels known as the levels of *emergence* (L_1)*, art* (L_2)*, engineering* (L_3)*,* and *post-engineering* (L_4)*, i.e.:*

$$EMM : L_1 \subseteq L_2 \subseteq L_3 \subseteq L_4 \qquad (8.4)$$

The EMM model of engineering maturity can be illustrated as shown in Fig. 8.3. Applying the EMM model, it may be found that some examples of existing engineering disciplines have already been at Level 4 – the post-engineering age, such as civil engineering, mechanical engineering, and electrical engineering. Electronic engineering would be at Level 3 – the engineering age, since it is still under rapid development within the context of a wide range technical innovation. Biological engineering is an example of those at Level 1 – the emerging age of an engineering discipline.

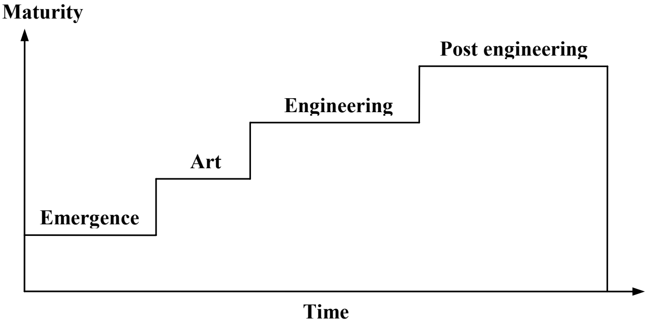


Figure 8.3 The engineering maturity model (EMM)

The EMM model can be taken as a reference for analyzing and organizing software engineering for a maturing engineering discipline. Based on the EMM model, we can predict that software engineering as a young engineering discipline is going to be matured in the same way: from art to engineering. Checking with the engineering maturity characteristics, software engineering is considered to be a good example of a Level-2 discipline in the art age, while it is under a transition toward Level 3 – the engineering age.

8.3 Basic Engineering Principles

The fundamental engineering objectives and the engineering approaches have been discussed in Section 8.2. This section elicits basic engineering principles, which focuses on those of engineering organization, technology, management, and professionalism. Applications of the basic engineering principles in software engineering will be explored in Section 8.4. A formal treatment of engineering organization in general and for software engineering in particular will be presented in Section 8.5 on the basis of the newly developed coordinative work organization theory [Wang, 2007d].

8.3.1 PRINCIPLES OF ENGINEERING ORGANIZATION

Engineering disciplines emerged and developed in the industrial revolutions share the following common principles in engineering organization:

- Apply systematic processes
- Adopt division of labor
- Support co-operative work
- Adopt quantitative measurement
- Establish standards
- Use tools and machinery
- Plan actual schedule
- Optimise resources allocation
- Derive predictable outputs
- Seek controllable quality

The essence of the above principles is the establishment of efficient engineering infrastructures and the rational forms of engineering organization. The key organizational principle of engineering invented in the industrial revolutions is *division of labor*, or limiting and specializing roles of labors in the whole production processes, which plays an important role in engineering organization. For instance, in electronic engineering an electronic engineer is not supposed to be specialized in all application areas of electric engineering: from low to high frequency circuits, from analogue to digital circuits, from real-time systems to home appliances. Similarly, an automobile engineer is not supposed to be skilled in all areas of car manufacturing and maintenance, such as mechanical structures, engines, transmissions, electronic systems, micro-controllers, petrol, lighting, safety facilities, and bodies of vehicles. Therefore, for large-scale software development, what we need is highly skilled software engineers who are competitive for one or limited roles, rather than a person with all-round skills in the software engineering processes. This is what we may learn from the general principles of industry engineering. More rigorous treatment of engineering organization theories will be developed in Section 8.5.

8.3.2 PRINCIPLES OF ENGINEERING TECHNOLOGY

The principles of engineering technology are elicited from a set of common processes and tactical approaches shared in all engineering disciplines. The technical principles of engineering are identified as follows [Wang, 2004c]:

- Identify rigorous foundations
- Apply established theories and methods
- Adopt specialized notation systems
- Improve visualization and tangibility
- Comparative study alternative methodologies
- Adopt matured technologies
- Identify repeatability in develop, design, and manufacturing
- Identify standard components
- Adopt mass production technologies
- Maximum reuse
- Adopt modeling and prototyping technologies
- Adopt measurement and metrics
- Reinforce rigorous testing and validation
- Improve compatibility and exchangeability
- Adopt quality assurance techniques
- Encourage technical innovation
- Pursue engineering elegance and efficiency
- Develop tools for self-sufficiency

More than a half of the technical principles of engineering principles have not been systematically adopted and applied in software engineering. This indicates that software engineering is a very young discipline as an engineering profession.

8.3.3 PRINCIPLES OF ENGINEERING MANAGEMENT

The principles of engineering management are at the center of all engineering principles, which establish guidelines for basic engineering organization, group collaboration, and support environments. The managerial principles of engineering are identified as follows [Wang, 2004c]:

- Recognize management roles
- Establish team work environment
- Understand the role of interpersonal coordination in groups
- Improve communications
- Ensure work product integration
- Maintain project data and documentation

8.3.4 PRINCIPLES OF ENGINEERING PROFESSIONALISM

Richard Gisselquist believed that "producing software is an engineering endeavor at the level of responsibility and ethics. … Engineers do not celebrate until they can walk across the completed bridge, holding their children's hands [Gisselquist, 1998]." This subsection discusses the principles of engineering professionalism in term of ethics.

According to Wright (2002), "Ethics is the study of the morality of human actions. It is the science of determining values in human conduct and of deciding what ought to be done in different circumstances and situations. Engineering ethics represents the attempts of professional engineers to define proper courses of action in their dealings with each other, with their clients and employees, and with the general public."

Although ethical principles are nontechnical ones for a professional engineer, they are designed to coordinate the professional practice in a sector. The nine professional characteristics of engineering principles, as shown below, represent an important aspect of the nontechnical features of a matured engineering profession.

- Establish laws and regulations
- Accumulate patents and know-hows
- Develop a code of ethics
- Define proudness and responsibility for professionals
- Identify a body of knowledge
- Promote continuous professional education
- Establish license or certificate schemes
- Evaluate environment effects
- Regulate safety roles

For establishing software engineering as a respected professional discipline, all above characteristics will attract much more attention in a modern society. Professional regulations in software engineering may be implemented by registration, certification, and/or licensing. There are a number of codes of ethics and professional practice for software engineering, which share similar principles and philosophies toward a cohesive way we act as software engineering professionals [IEEE/ACM, 1998].

8.4 Engineering Principles for Software Engineering

Engineering approaches to large-scale software development have been identified as established methodologies, processes, tools, standards, organization methods, management methods, and quality assurance systems. Interesting findings on what software engineering may learn from generic engineering principles are discussed in this section.

8.4.1 THE ENGINEERING CHARACTERISTICS OF SOFTWARE ENGINEERING

Analyzing the descriptions on generic engineering approaches and basic engineering principles in Sections 8.2 and 8.3, it can be seen that in order to identify a comprehensive set of characteristics of software engineering, a comparative approach of thinking between existing engineering disciplines and software engineering may be inspirational. This subsection takes a *comparative approach* to software engineering, in order to perceive the nature and characteristics of software engineering and to understand what software engineering may learn from the generic engineering principles.

It was a dilemma: "to be or not to be" on the combination of the two terms software and engineering. Software professionals have been arguing the term *software engineering* and its intensions and extensions for more than four decades since Friedrich Bauer proposed it in 1968 [Bauer, 1972/1976; Naur and Randell, 1969]. Yet still some fundamental questions remain, such as: a) Is software development an engineering discipline? and b) Are software developers engineering professionals? There were completely different assertions and opinions on the contradictory issues, and it is still confusing the academia, practitioners, and students in software engineering.

However, according to the EMM model presented in Theorem 8.3, the above myth is caused by a confusion of timely maturity in perceiving the software profession and software engineering. The rational answer to the question if software development is an engineering discipline is that although it is *not* at present and in the past, it will be and should be *yes* in the future.

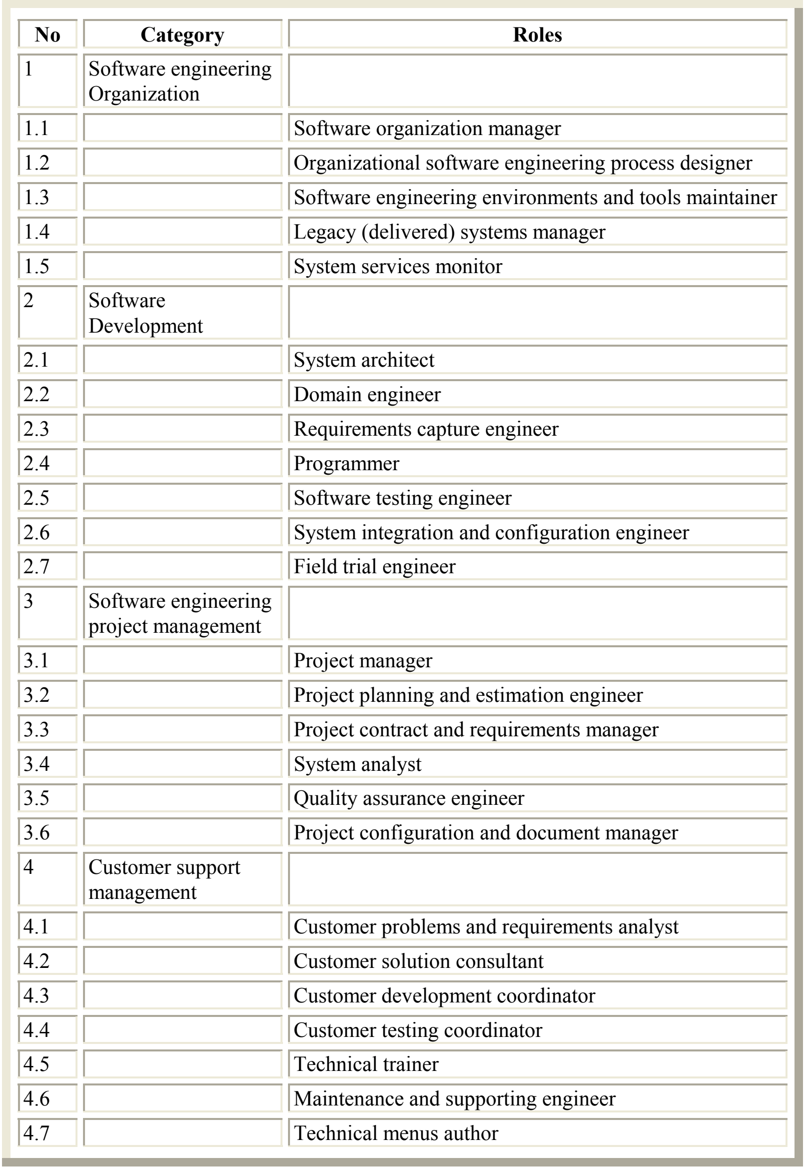
As discussed in Sections 8.2 and 8.3, engineering is a set of disciplines seeking solutions for complicated problems and systems that could not be done by individuals. The key aim of engineering is to repetitively produce complex artifacts in an efficient way. Thus, to many professionals, engineering means systematic planning, teamwork, rigorous process, repeatability, and efficiency.

Software engineering is a maturing engineering discipline that adopts the generic engineering principles in the development of large-scale software, which could not be produced by individuals. Currently, software development is evolving from the laboratory-oriented and all-round-programmer-based practice to an industry-oriented and process-based platform, and software developers are experiencing changes of roles from craftsmen to regulated professionals – the software engineers [Wang and King, 2000a; Wang and Patel, 2000]. The practices of the former are based on personal talents, tastes, and art, while those of the latter are based on disciplined processes and repeatable professional activities.

8.4.2 DIVISION OF LABOR

According to the generic engineering principles, one of the keys of software engineering organization and practice is division of labor, which is so obvious and so often to be ignored in current software engineering practice. For large-scale software development, what we need is highly skilled software engineers who are competitive for one or limited roles, rather than a person with all-round skills in the software engineering processes.

Table 8.3
Roles of Software Engineers in Software Engineering



No	Category	Roles
1	Software engineering Organization	
1.1		Software organization manager
1.2		Organizational software engineering process designer
1.3		Software engineering environments and tools maintainer
1.4		Legacy (delivered) systems manager
1.5		System services monitor
2	Software Development	
2.1		System architect
2.2		Domain engineer
2.3		Requirements capture engineer
2.4		Programmer
2.5		Software testing engineer
2.6		System integration and configuration engineer
2.7		Field trial engineer
3	Software engineering project management	
3.1		Project manager
3.2		Project planning and estimation engineer
3.3		Project contract and requirements manager
3.4		System analyst
3.5		Quality assurance engineer
3.6		Project configuration and document manager
4	Customer support management	
4.1		Customer problems and requirements analyst
4.2		Customer solution consultant
4.3		Customer development coordinator
4.4		Customer testing coordinator
4.5		Technical trainer
4.6		Maintenance and supporting engineer
4.7		Technical menus author

A software engineer is a professional whose role and skills are regulated by the software engineering discipline and processes. Examining the requirements for functions of software engineers in software engineering at the technical, managerial, and organizational categories, a variety of roles can be identified as shown in Table 8.3 [Wang and King, 2000a]. Observing Table 8.3, it can be found that a software engineer may be responsible for only one or limited role(s) rather than a master of all the skills in software engineering processes. This is what we may learn from the universal principles of industrial engineering, which is so obvious and so easy and so often to be ignored in practice. Therefore, the key is 'division of work', or limitation of the roles of a software engineer in the whole software development processes. Formal models of division of labor will be presented in Section 11.2 on management foundations of software engineering.

8.4.3 CHARACTERISTICS OF SOFTWARE ENGINEERING IN THE ENGINEERING AGE

Conventional industries produce products from raw materials via engineering approaches; while the software industry produces software solutions for problems via software engineering. Software engineering is going to be a discipline that fully adopts engineering approaches, such as established methodologies, processes, tools, standards, organization methods, management methods, quality assurance systems and the like, in the development of large-scale software. The aims of software engineering are to improve productivity and quality, keep timeliness, prolong software life span, and maximum benefit in software development.

Because software engineering is a young discipline, there is still some way to go for software engineering to be a matured engineering discipline. A fundamental issue we may learn from the generic principles of industrial engineering is there are still significant gaps in many important practices of software engineering in the engineering way, such as:

- Team-work oriented
- Human-oriented programming and documentation rather than machine-oriented
- Following common roles rather than personal hobbies
- Explicit description of roles, best practices, and regulated processes rather than leaving them loosely as personal experience or private knowledge
- System test and validation should carried out independently from original developers or vendors

- Individual software engineers should be prepared to fit in specific processes rather than tend to be a master of all-round activities in development
- Maximizing application and reuse of available components and tools in development rather than tends to be self-sufficient

A detailed analysis of characteristics of software engineering at different maturity ages is provided in Table 8.4. The information in Table 8.4 shows that current software development practices and software engineering education are still located in-between the ages of art and engineering, because the center of education and practices is mainly craftsman-and-laboratory-environment-oriented. For software engineering to evolve into a mature engineering age there is still much to do as described in the right-hand column of Table 8.4. However, from a historical point of view, it is encouraging to see that software engineering, as an engineering discipline, has been matured to be between Level 2 and Level 3 as in the EMM model in just four decades. Some existing engineering disciplines, such as civil engineering and manufacturing engineering, would have taken hundreds of years to reach their current levels of maturity.

By recognizing the current status of software engineering as a discipline locating between Levels 2 and 3 according to the EMM scale, responsibilities of software engineering researchers and practitioners are to push forward software engineering to a matured engineering discipline by applying the generic engineering principles gained from other matured engineering disciplines.

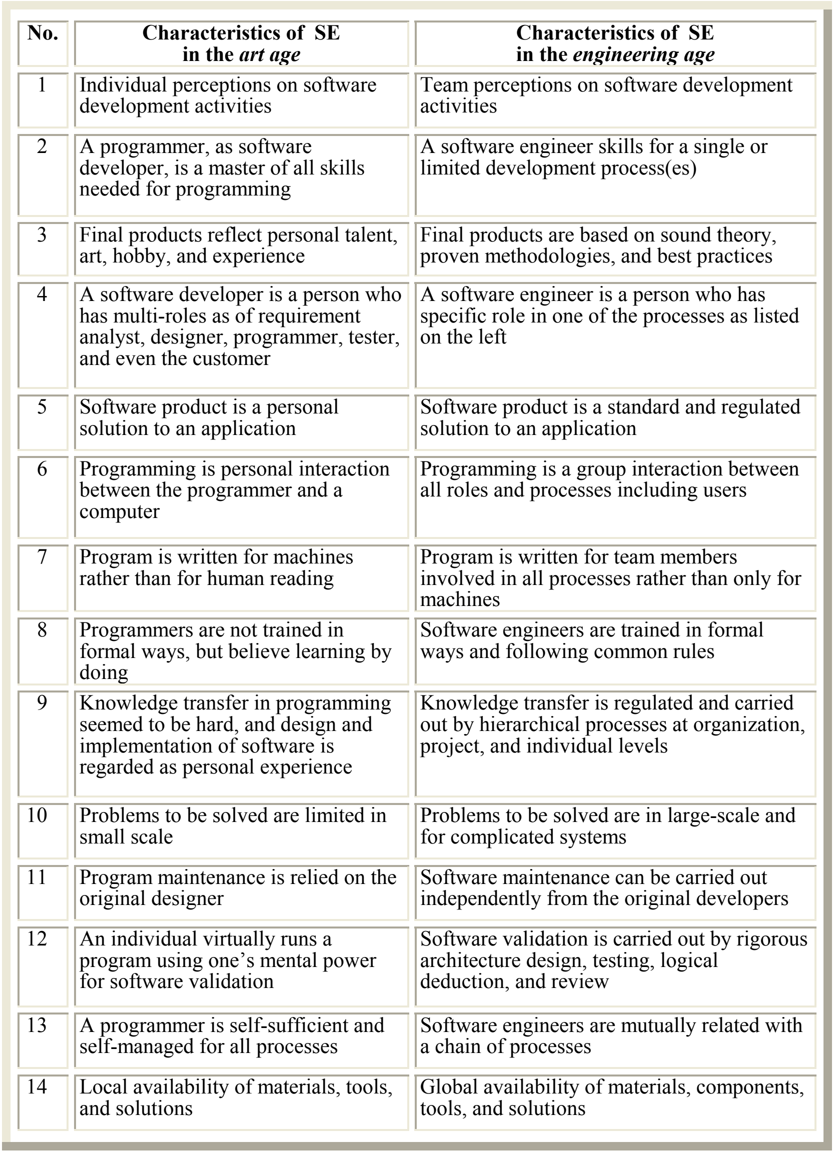
8.4.4 UNIQUE PRINCIPLES OF SOFTWARE ENGINEERING

In addition to the generic principles of software engineering as discussed in the previous sections, unique domain specific principles of software engineering may be identified. These unique principles can be classified into the categories of cognitive characteristics and special problem-domain characteristics of software engineering.

The fundamental cognitive informatics principles of software engineering are those of its informatics properties, intelligent behaviors, denotational mathematics needs, and cognitive complexity. In Section 3.5.1 eight fundamental cognitive characteristics of software engineering are identified. Detailed discussions of the basic cognitive characteristics of software engineering will be presented in Sections 9.5 and 9.6 [Wang, 2004b/06a/07a].

The domain-specific principles of software engineering encompass the following basic characteristics that determine the difficulty of software development and require a broad knowledge structure for software engineers [Wang and Patel, 2000]:

Table 8.4
Characteristics of Software Engineering Practices at Different Levels of EMM



No.	**Characteristics of SE in the *art age***	**Characteristics of SE in the *engineering age***
1	Individual perceptions on software development activities	Team perceptions on software development activities
2	A programmer, as software developer, is a master of all skills needed for programming	A software engineer skills for a single or limited development process(es)
3	Final products reflect personal talent, art, hobby, and experience	Final products are based on sound theory, proven methodologies, and best practices
4	A software developer is a person who has multi-roles as of requirement analyst, designer, programmer, tester, and even the customer	A software engineer is a person who has specific role in one of the processes as listed on the left
5	Software product is a personal solution to an application	Software product is a standard and regulated solution to an application
6	Programming is personal interaction between the programmer and a computer	Programming is a group interaction between all roles and processes including users
7	Program is written for machines rather than for human reading	Program is written for team members involved in all processes rather than only for machines
8	Programmers are not trained in formal ways, but believe learning by doing	Software engineers are trained in formal ways and following common rules
9	Knowledge transfer in programming seemed to be hard, and design and implementation of software is regarded as personal experience	Knowledge transfer is regulated and carried out by hierarchical processes at organization, project, and individual levels
10	Problems to be solved are limited in small scale	Problems to be solved are in large-scale and for complicated systems
11	Program maintenance is relied on the original designer	Software maintenance can be carried out independently from the original developers
12	An individual virtually runs a program using one's mental power for software validation	Software validation is carried out by rigorous architecture design, testing, logical deduction, and review
13	A programmer is self-sufficient and self-managed for all processes	Software engineers are mutually related with a chain of processes
14	Local availability of materials, tools, and solutions	Global availability of materials, components, tools, and solutions

- Intangible objects and work products, and intricate relations and interactions between them
- Problem domain is infinite, including all application areas of all existing engineering disciplines
- Software engineering is design-intensive opposed to repetitive mass production
- Application development is one-off activity
- Development processes are stable and repetitive
- A specific design and implementation of a software system is only one of all possible solutions for a real-world problem on the basis of tradeoffs and constraints
- Software engineering needs new forms of denotational mathematics as identified in Section 4.5 that are different from current analytic ones

The most significant and unique characteristic of software engineering lays on the fact that its problem domain is infinitive, because it encompasses almost all other domains in the real world, from scientific problems and real-time control to word processing and games. It is infinitely larger than the specific and limited problem domains of the other engineering disciplines. This stems from the notion of a computer as a universal intelligent machine, and is a feature fundamentally dominating the complexity in engineering design and implementation of varying software systems.

8.4.5 PROFESSIONALISM OF SOFTWARE ENGINEERING

Via contrasting professional engineers and amateurs in software engineering, this subsection highlights professionalism as one of the important requirements for software engineers. Then, the software engineering ethics and professional practice as recommended by IEEE/ACM are summarized.

8.4.5.1 Professionalism of Software Engineers

There is a special phenomenon in software engineering that anybody who is able to use a programming language may claim that one can programming or even be a software engineer. This is just like that one who acquires reading and writing ability in a natural language may claim oneself as a writer; or one who is able to build a simple shelter or doghouse may claim oneself as a civil engineer.

If knowing or even understanding a programming language is not enough to be a qualified software engineer, then what else is needed? The answers may be obtained by analyzing the differences between professionals and amateurs.

Professional software engineers are persons with professional cognitive models and knowledge on software engineering. They are trained and experienced in:

- Fundamental knowledge governing software and software engineering practices
- Proven algorithms
- Problem domain knowledge
- Problem solving experience
- Programming languages
- Program developing tools/environments
- Solid programming knowledge
- A global view on software development, including required functionalities, exceptions handling, and fault-tolerance

However, amateurish programmers are persons who know only one or a couple of programming languages but lack fundamental knowledge, skills, and experience as those of professionals identified above. Amateurs may be characterized as follows in software engineering:

- Possession of an ad hoc structure of programming knowledge
- Eagerly trying what are directly required for a program
- Tending to focus on details without a global and systematic view on software as a system

In discussing "what makes a good software engineer" in a panel, Marcia Finfer (1989) believed: "the answer, in my opinion, is simply the combination of both innate skill and significant experience in building real systems against a set of functional and performance requirements and a given budget and schedule." This shows that professional experience is a primary factor of professional software engineers. Also, possession of fundamental principles of software engineering is essential towards excellence of software engineers.

Software engineering encompasses theory, technology, practice, and application of software in computer-based systems. A central theme of software engineering education is to engender an engineering discipline in software engineers and students, enabling them to define and use processes, models, and metrics in software and system development.

A software engineer as a professional must demonstrate the ability to analyze, design, verify, validate, implement, and maintain software systems, using appropriate quality assurance techniques/methods in all practices. They must possess the necessary team and communication skills to function in a typical software development environment. Engineering responsibility and practice have to be stressed, which includes conveying ethical, social, legal, economic, and safety issues. These concerns must be reinforced in advanced work with appropriate use of software engineering standards. Software engineers should also learn methods for technical and economic decision making, such as project planning and resource management.

The social responsibility of software engineers above their personal and professional responsibilities is stressed by David Parnas. He wrote: "My view is that those of us who have received an extensive education from society have a debt to repay; we have to share our knowledge with that society when it can be of benefit to that society [Parnas, 1994]."

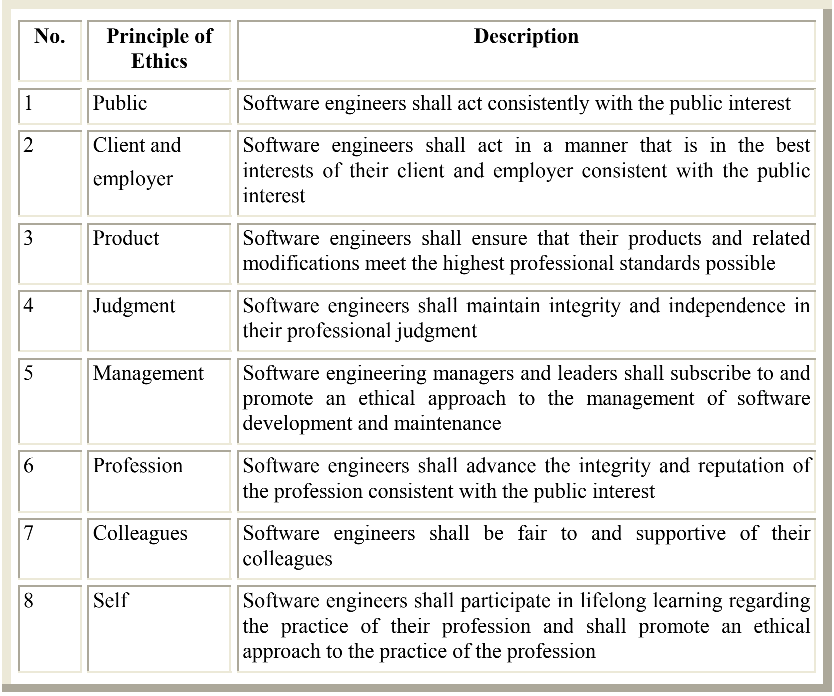
8.4.5.2 Ethical Practice in Software Engineering

The *Software Engineering Code of Ethics and Professional Practice* was recommended by the IEEE/ACM Joint Task Force in 1998 [IEEE/ACM, 1998]. The code requires that: "Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession."

In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following eight aspects of ethics as shown in Table 8.5.

Professionalism of software engineers is mainly reflected in professional practice, professional judgment, public responsibility, and product responsibility. They are specified in the IEEE/ACM *Software Engineering Code of Ethics and Professional Practice* [IEEE/ACM, 1998] as follows.

Table 8.5
A Summary of IEEE/ACM Software Engineering Code of Ethics and Professional Practice [IEEE/ACM, 1998]



No.	Principle of Ethics	Description
1	Public	Software engineers shall act consistently with the public interest
2	Client and employer	Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest
3	Product	Software engineers shall ensure that their products and related modifications meet the highest professional standards possible
4	Judgment	Software engineers shall maintain integrity and independence in their professional judgment
5	Management	Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance
6	Profession	Software engineers shall advance the integrity and reputation of the profession consistent with the public interest
7	Colleagues	Software engineers shall be fair to and supportive of their colleagues
8	Self	Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession

Professional practice of software engineers: Acting software engineering ethically; promoting public awareness; extending software engineering knowledge by life-long learning; supporting other engineers practicing the ethics; putting the professional interest above private ones; obeying all laws governing their work; avoiding false, speculative, vacuous, deceptive, misleading, or doubtful descriptions of work products; taking responsibility for detecting, correcting, and reporting errors in software and associated documents; avoiding associations with businesses and organizations conflicting with this code; and reporting violations of the ethics.

Professional judgment of software engineers: Tempering technical judgments to human values; endorsing work products cautiously; maintaining objective judgment when evaluating work products; and avoiding conflict interest.

Public responsibility of software engineers: Accepting full responsibility for one's own work; moderating all parties' interests with the public good; maintaining software's safety, quality, and protecting the environment; disclosing any actual or potential danger to the public caused by software, its installation, maintenance, support or documentation; being encouraged to volunteer professional skills to good causes; and contributing to public education concerning the discipline.

Product responsibility of software engineers: Striving for high quality, acceptable cost and a reasonable schedule; ensuring proper and achievable objectives for a project; ensuring the use of an appropriate methodology; working to follow professional standards; ensuring the understanding of project specifications; reviewing and approving specifications; proving realistic estimates of cost, scheduling, personnel, quality and outcomes on a project; ensuring adequate testing, debugging, and review of software and related documents; ensuring adequate documentation of a software system; respecting the privacy and information security; being careful to use only accurate data; and maintaining the integrity of data and information.

8.5 The Theory of Software Engineering Organization

In his classic book on *The Mythical Man-Month*, Frederick Brooks presents a well-known empirical study on the myths of the relationship between labor (number of persons) and time (duration in months) in software engineering [Brooks, 1975/95]. However, in the last chapter of the book, Brooks has still left the conclusions open: "Propositions of the mythical man-month: true or false?"

From a more generic management science point of view, the convention to measure the project workload by a product of labor and time known as person-month really caused more problems than it explained, because workload is a common and complicated phenomenon existing not only in software engineering, but also in project management and economical decision making in all engineering disciplines.

8.5.1 BASIC PROPERTIES OF COORDINATIVE WORK IN ENGINEERING

This subsection formally analyzes the properties of coordinative work and the age-old myth on project workload or effort in term of person-month. Mathematical models that explain the equivalence and transformability between labor and time in work organization are systematically developed, which rigorously describes the conditions of the interchangeability of basic elements in a coordinative workload [Wang, 2006g/07d].

8.5.1.1 The Mechanisms of Coordinative Workload and Effort

Definition 8.6 The *workload W* of a coordinative project is determined by a product of the number of labor *L* and the duration *T* needed or spent in a project, i.e.:

$$W = L \bullet T \qquad [\mathrm{PM}] \qquad (8.5)$$

where the unit of labor is *person* (P), the unit of duration is *month* (M), and as a result the unit of workload is *person-month* (PM).

There are numerous myths on the simple relationship between labor and duration defined in Eq. 8.5 in empirical studies, because a number of fundamental questions on the nature of the hybrid product of workload in PM remain [Brooks, 1975/95; Wang, 2006g/07d]. For example, how many persons and how many months are needed for a given *W*? Are 1.0P • 10.0M = 10.0P • 1.0M = 10.0PM?

All the empirical questions in applications on the nature of coordinative workload can be reduced to the following fundamental problems.

Question 8.1 Whether labor *L* or duration *T* is arbitrarily determinable for a given workload *W* in a coordinative work?

Question 8.2 Are labor *L* and duration *T* interchangeable for a given workload *W* in collaborative work organization?

The following lemma answers Question 8.1. Theorem 8.4 will provide formal explanations for Question 8.2.

Lemma 8.1 The generic form of *workload W* carried out by more than one person is always supplemented by an inevitable overhead *h*, i.e.:

$$W = L \bullet T_1 (1+h) \qquad [\mathrm{PM}] \qquad (8.6)$$

where *h* is called the interpersonal coordination overhead in a multiple personal project, and T_1 is the time needed to complete the work by only one person.

According to Lemma 8.1, the *workload W* defined in Eq. 8.5 is a special case where the project only involves a single person ideally and therefore there is no interpersonal coordination overhead *h*.

8.5.1.2 The Rate of Interpersonal Coordination

It is observed that many factors may affect the workload of a coordinative project [Mooney, 1947; Gray, 1989; Hardy and Phillips, 1998; Huseman and Miles, 1988; Huxham, 1996; Pasquero, 1991; Roberts and Bradley, 1991; Wang, 2007d; Wood and Gray, 1991], such as documentation, swap between roles in a project, and interactions to other groups in the organization. However, a macro indicator known as the *interpersonal coordination rate r* is a unique factor that distinguishes a single person project and a multiple-person coordinative project. Therefore, the role of *r* is the key to solve the myths on coordinative work organization.

Definition 8.7 *Interpersonal coordination activities* are tasks that can not be done by an individual, such as communication, meeting, synchronization, peer review of work products, standardization, supervision, and quality assurance.

The effort on interpersonal coordination activities as a necessary overhead of a coordinative project can be collectively analyzed by the extra time spent by individuals in the project.

Definition 8.8 The *interpersonal coordination rate r* is an average ratio of the time spent on interpersonal coordination activities t_r and the total working time of a person *T* in a given project, i.e.:

$$r = \frac{t_r}{T} \tag{8.7}$$

According to Definition 8.8, an empirical method for collecting and calculating *r* on the basis of work time distributions is provided in Ex. 8.19.

The average rate of interpersonal coordination *r* has a scope of 0 (0%) through 1.0 (100%), where $r = 0$ means there is no interpersonal coordination and $r = 100\%$ means all time has been spent on interpersonal coordination. These are the two extremes that constrain a coordinative work.

For instance, in software engineering, a wide variety of factors may affect the interpersonal coordination rate. Ten major factors, such as the scope of project, importance, difficulty, complexity, domain knowledge requirement, experience requirement, special process needed, schedule constraints, budget constraints, and other process constraints, have been identified in [Wang and King, 2000a] as summarized in Table 8.6, where a set of sample weights for the coordination factors for a specific project is also given as an example.

Table 8.6
Key Factors Affecting the Rate of Interpersonal Coordination in Software Engineering

No.	**Factors of project**	**Scope of Weight (w_i)**		
		High	**Medium**	**Low**
1	Scope	√		
2	Importance		√	
3	Difficulty	√		
4	Complexity	√		
5	Domain knowledge requirement			√
6	Experience requirement		√	
7	Special process needed		√	
8	Schedule constraints	√		
9	Budget constraints			√
10	Other process constraints		√	

Note: High = 10, Medium = 5, and Low = 1

When the weight for each project factor is determined on a measurement scale of 1 through 10, the interpersonal coordination needed for the given project can be empirically determined as follows:

$$r = \frac{t_r}{T} \propto \frac{\sum\limits_{i=1}^{10} w_i}{100} \tag{8.8}$$

Eq. 8.8 indicates that the empirical range of *r* in software engineering is 0.001 ≤ *r* < 0.999, or *r* is between 0.1% to 99.9%. For example, with the particular layout of a project as given in Table 8.6, the average interpersonal coordination rate *r* is proportional to:

$$\begin{aligned} r &= \frac{\sum\limits_{i=1}^{10} w_i}{100} = (10 + 5 + 10 + 10 + 1 + 5 + 5 + 10 + 1 + 5)/100 \\ &= 0.62 \end{aligned}$$

Real-world project data collected in recent surveys in the software industry show *r* is between 12.5% to 47.8% [Wang, 2007d]. Higher rate of *r* is also reported up to 70.0% at IBM [McCue, 1978]. The data also indicate that *r* may vary in different processes of software engineering, ranging averagely from 49.3% in the design process, 30.8% in the coding process, 60.0% in the integration/testing process, and 47.8% in the maintenance process, respectively. It is noteworthy that different development methodologies may affect the calibration of *r*. For instance, *r* = 30.2% for projects organized according to conventional waterfall models, and it may be up to 50.2% when projects are organized by extreme programming or agile processes.

8.5.1.3 The Overhead of Interpersonal Coordination

Definition 8.9 The *number of interpersonal coordination, n*, needed in a group of size L, $L \geq 2$, is determined by the number of pairwise coordination in the group, i.e.:

$$\begin{aligned} n &= C_L^2 = \frac{L!}{2!\,(L-2)!} \\ &= \frac{L \bullet (L - 1)}{2} \end{aligned} \tag{8.9}$$

where L is the number of labor in the group. In addition, a possible *k*-nary coordination, $k > 2$, within the group can be treated as multiple pairwise ones.

Lemma 8.2 The *overhead of interpersonal coordination h* in a multiple personal project ($L > 1$) is proportional to both the number of pairwise relations *n* and the average rate of time spent in each pair of coordination *r*, i.e.:

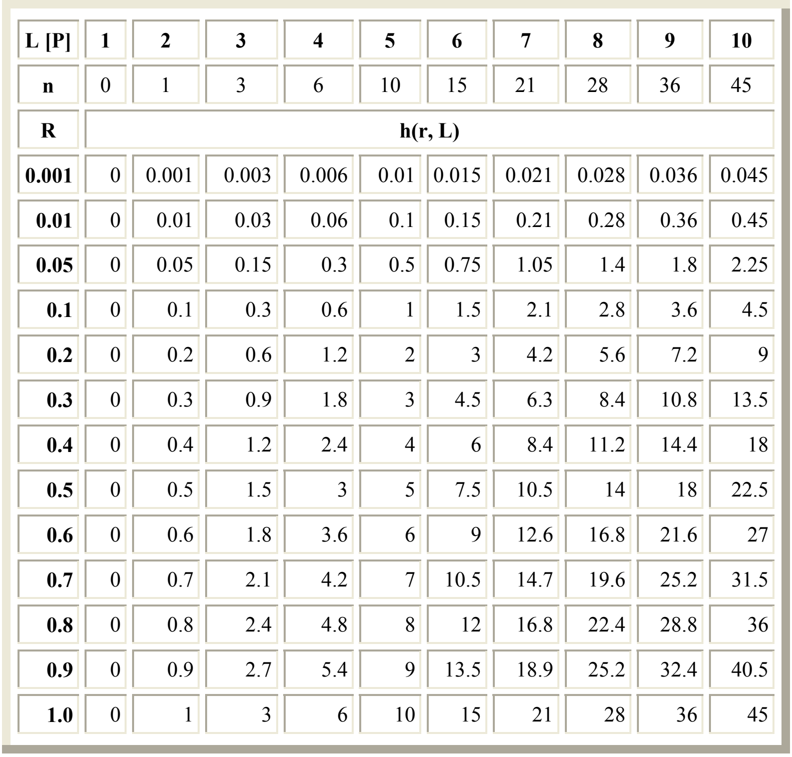
$$\begin{aligned} h &= r \bullet n \\ &= r \bullet \frac{L(L-1)}{2} \end{aligned} \tag{8.10}$$

where multiple personal relations in the project can be treated as the combinations of multiple pairwise relations.

Lemma 8.2 indicates that the interpersonal coordination overhead *h* is a function of *r* and *L*, i.e., *h*(*r*, *L*), which represents the efficiency of the transformation between labor and time in a coordinative project. For a given *r* for a coordinative project, the more the persons involved in the project, the faster the overhead for coordination increases. For instance, according to Eq. 8.10, if there are three persons, i.e., $L = 3$, in a coordinative project where $r = 0.4$, the interpersonal overhead $h = 1.2$; for $L = 10$, $h = 18$; and for $L = 1{,}000$, $h = 199{,}800$. Obviously, $h = 0$ if a project is with only one person.

Typical overheads, *h*(*r*, *L*), for interpersonal coordination are provided in Table 8.7 determined by Eq. 8.10. With the data derived in Table 8.7, Lemma 8.2 can be illustrated by the 13 curves as shown in Fig. 8.4, where the first curve *h*(0.001, *L*) is very close to zero. It is noteworthy that Lemma 8.2 is a generic model that is valid for the domains $0 < r \leq 100\%$ and $1 \leq L \leq \infty$ for any coordination project.

Table 8.7
Overhead of Interpersonal Coordination h(r, L)



L [P]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
n	0	1	3	6	10	15	21	28	36	45
R	**h(r, L)**									
0.001	0	0.001	0.003	0.006	0.01	0.015	0.021	0.028	0.036	0.045
0.01	0	0.01	0.03	0.06	0.1	0.15	0.21	0.28	0.36	0.45
0.05	0	0.05	0.15	0.3	0.5	0.75	1.05	1.4	1.8	2.25
0.1	0	0.1	0.3	0.6	1	1.5	2.1	2.8	3.6	4.5
0.2	0	0.2	0.6	1.2	2	3	4.2	5.6	7.2	9
0.3	0	0.3	0.9	1.8	3	4.5	6.3	8.4	10.8	13.5
0.4	0	0.4	1.2	2.4	4	6	8.4	11.2	14.4	18
0.5	0	0.5	1.5	3	5	7.5	10.5	14	18	22.5
0.6	0	0.6	1.8	3.6	6	9	12.6	16.8	21.6	27
0.7	0	0.7	2.1	4.2	7	10.5	14.7	19.6	25.2	31.5
0.8	0	0.8	2.4	4.8	8	12	16.8	22.4	28.8	36
0.9	0	0.9	2.7	5.4	9	13.5	18.9	25.2	32.4	40.5
1.0	0	1	3	6	10	15	21	28	36	45

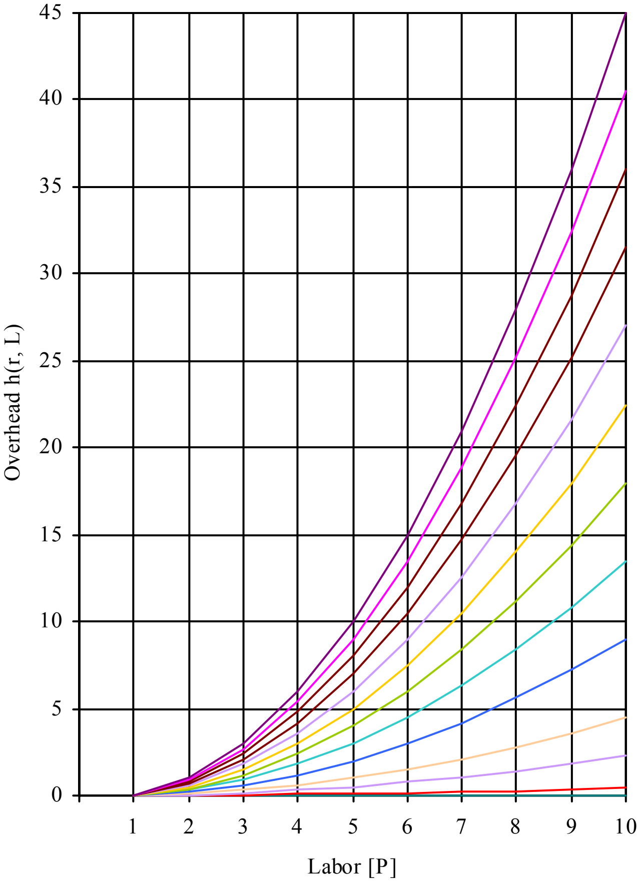


Figure 8.4 Overhead of interpersonal coordination when $r \in \{0.001 \ldots 1\}$

8.5.1.4 The Nature of Coordinative Work in Engineering

Definition 8.10 The *actual time T* spent in a multi-person project is determined by the ideal average time spent on work by a single person T_1 and the total overhead of interpersonal coordination h, i.e.:

$$\begin{aligned} T &= T_1(1 + h) \\ &= T_1(1 + r \bullet \frac{L(L-1)}{2}) \end{aligned} \tag{8.11}$$

where the more the persons involved, the longer the duration of the project.

Replacing *T* in Eq. 8.5 with the above expression, the inherent nature of coordinative work and the generic form of the actual workload in a coordinative project can be revealed by the following law.

The 23rd Law of Software Engineering

Theorem 8.4 The *coordinative workload in engineering* states that the *actual workload W* of a coordinative project is a function of the average interpersonal coordination rate *r* and the number of labor *L* in the project, i.e.:

$$\begin{aligned} W &= L \bullet T \\ &= L \bullet T_1(1+h) \\ &= W_1(1+h) \\ &= W_1(1 + r \bullet \frac{L(L-1)}{2}) \quad [\mathrm{PM}] \end{aligned} \tag{8.12}$$

where T_1 is the *indicational duration* needed to complete the work by only one person, and W_1 is the *ideal workload* without the interpersonal overhead *h* or that of a single person project.

This is the first fundamental finding on laws of coordinative work organization [Wang, 2007d]. The 23rd Law of software engineering (Theorem 8.4) reveals that the ideal workload W_1 defined in Eq. 8.12 is a special case where the project only involves one person ideally and therefore there is no interpersonal coordination overhead ($h = 0$). Although, some other types of overhead may exist in various projects, $h(r, L)$ is the unique property found only in coordinative projects.

It is noteworthy that the generic coordinative workload model developed in Theorem 8.4 is development-cycle/structure independent, because it is only a function of $W(r, L)$. The theory fits all three forms of system organizations in serial, parallel, and hybrid structures at both the unit/process level and the whole project level, because a hybrid structure of work organization can be analyzed segmentally where each segment is a simple serial or parallel structure.

In the context of software engineering, since a software project organized by any process model falls into one of the three basic system structures, the project as a whole or as a set of segmented processes obeys the same laws. Theorem 8.4 is also applicable to any real-world instance or specific case that uses the waterfall model, incremental model, or process models, because various model adoptions may only change the instance values of the interpersonal coordination rate *r* rather than the law itself. In addition, the following empirical observations and heuristic principles in software and system engineering such as: a) The Brooks' principle that states "Adding people in a late project make it later [Brooks, 1975/95]," and b) The Schonberger's observation on "Why Projects are Always Late [Schonberger, 1981]," are specific evidences supporting the generic work organization theory.

8.5.2 LAWS OF WORK ORGANIZATION IN SOFTWARE ENGINEERING

Based on the understanding of the nature of coordinative workload in engineering projects and the key role of interpersonal coordination rate in team work, a set of laws for engineering organization in general and for software engineering organization in particular will be established in this subsection.

8.5.2.1 The Laws of Incompressibility of Software Engineering Workload

Observing Theorem 8.4 it can be seen that the ideal workload W_1 of a project is the minimum workload in coordinative tasks, and it cannot be reduced no matter how many persons are involved via any kind of labor allocation.

The 22nd Principle of Software Engineering

Theorem 8.5 The *incompressible workload* states that a given ideal workload W_1 in software engineering can not be compressed by any kind of labor allocation, i.e.:

$$W \geq W_1 = W_{min} \tag{8.13}$$

and in the best case when there is only one person involved, the minimum workload $W = W_1 = W_{min}$ may be reached.

Proof: According to Theorem 8.4, $W = W_1(1+h)$. (a) In a multi-person project, because $h > 0$ by any kind of labor allocation, therefore $W > W_1$. (b) In a single-person project, since $h = 0$, therefore, $W \geq W_1 = W_{min}$.

Theorem 8.5 indicates that via coordination in a multi-personal project the duration of the project may be reduced, but the total workload cannot be reduced because the minimum workload for a given project is reached at $W_{min} = W_1$. In other words, although labor and time may be interchangeable, the minimum workload for a given project is a constant. Therefore, the total workload in any type of coordinative labor allocation will be larger than the minimum.

8.5.2.2 The Law of Interchangeability between Labor and Time in Software Engineering

On the basis of Theorems 8.4 and 8.5, the mathematical model of the interchangeability between labor and time can be formally derived as follows.

The 24th Law of Software Engineering

Theorem 8.6 The *interchangeability of labor and time* (ILT) states that, for a given workload *W*, labor *L* and duration *T* are transformable under the following condition:

$$\begin{aligned} T &= \frac{W}{L} \\ &= \frac{W_1}{L}(1 + r \bullet \frac{L(L-1)}{2}) \\ &= \frac{W_1}{L}(\frac{1}{2}rL^2 - \frac{1}{2}rL + 1) \\ &= \frac{1}{2}W_1(rL - r + \frac{2}{L}) \end{aligned} \tag{8.14}$$

Proof: Solving Eq. 8.12 for *T* obtains the above conclusion.

The 24th Law of software engineering indicates that the duration of a coordinative project is a function of labor *L* and the interpersonal coordination rate *r* for the given project. In case where *r* is a variable in dynamic project organization, such as in different processes of a software engineering project, each individual process can be treated as a subproject with a constant *r*, or simply, a mathematical mean of *r* for all processes may be adopted.

8.5.2.3 The Laws of the Shortest Duration of Coordinative Work in Software Engineering

The 25th Law of Software Engineering

Theorem 8.7 The *shortest duration of coordinative work* states that there exists the *shortest duration T_{min}* under the *optimum labor allocation L_0* for a given ideal workload W_1 with a certain interpersonal coordination rate r, i.e.:

$$\begin{cases} T_{\min} = \{T \mid L = L_0\} \\ \quad\; = \frac{1}{2}W_1(rL_0 - r + \frac{2}{L_0}) \quad [M] & (8.15) \\ \\ L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil, \; r \neq 0 \quad [P] & (8.16) \end{cases}$$

Proof: Because $T(r, L)$ as given in Eq. 8.14 is a deferential function on L when r is known for a certain project, it reaches the minimum T_{min} when its derivative equals to zero, i.e.:

$$\begin{aligned} \frac{\partial T}{\partial L} &= \frac{\partial}{\partial L}(\frac{1}{2}W_1(rL - r + \frac{2}{L})) \\ &= \frac{1}{2}W_1(r - \frac{2}{L^2}) \\ &= 0 \end{aligned} \tag{8.17}$$

Eq. 8.17 yields $r - \frac{2}{L^2} = 0$, i.e., the optimum labor allocation is:

$$\begin{aligned} L_0 &= \left\lceil \sqrt{\frac{2}{r}} \right\rceil \\ &= \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil, r \neq 0 \end{aligned} \tag{8.18}$$

where L_0 will be rounded to the ceiling of an integer, i.e., the minimum number of persons needed for a given project.

This is the second fundamental finding on laws of coordinative work organization [Wang, 2007d]. The 25th Law of software engineering (Theorem 8.7) reveals, for the first time, that the optimal labor allocation in engineering project organization *is not* related to the *size* or the ideal workload of a given project as conventional empirical studies suggested. Surprisingly, it is merely determined by the interpersonal coordination rate for the project. The 25th Law had hardly been realized in empirical studies in management science and system engineering [Brooks, 1975/79; Schonberger, 1981], because of the vital need for a long chain of insightful reasoning that seamlessly transforms Eq. 8.5 through Eq. 8.18.

Although other factors as identified earlier may influence the shortest duration T_{min} in a certain project, Theorem 8.7 provides insight on coordinative work allocation out of all the trivial factors that have hidden the key truth of rational work organization for decades since the establishment of management science [Tayler, 1911] and system science [Klir, 1992].

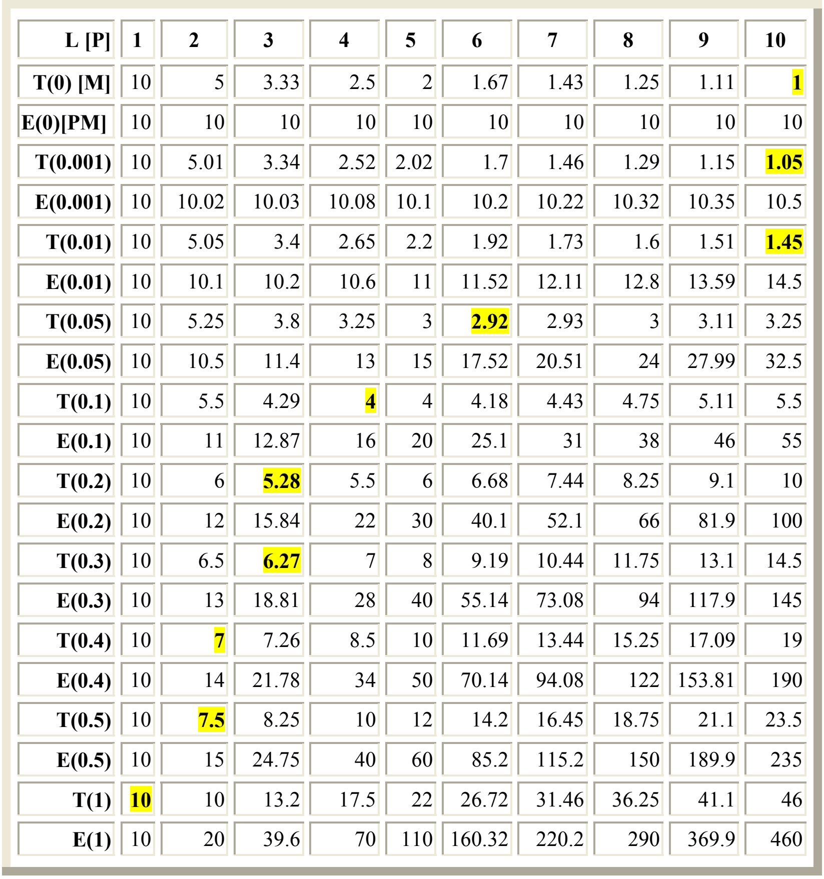
As a result of a complicated long-chain reasoning, Theorems 8.4 through 8.7 reveal and prove mathematically the existence and predictability of the minimum of project duration determined by the optimum labor allocation under certain group coordination rate *r*. Although there were empirical observations on the minimum, such as Brooks' work (1975), rigorous mathematical explanation has not been created for this profound phenomenon in management science [Tayler, 1911], system engineering [Klir, 1992], and operations theories [Schmenner and Swink, 1998].

It is noteworthy that the same work coordination laws and mathematical formulae may be used at subproject or individual process levels as well as at the whole project level. In the former case, the labor *L* does not necessarily be treated as a constant in the entire lifecycle of the given project. There are two ways to deal with *L*'s flexibility: a) Whenever *L* needs to be different in a certain process of a project, the workload of this process may be recalculated by the same law in the same mathematical form. b) In the planning phase, *L* may be deemed as an average of labor allocations in the entire lifecycle of the project.

A set of typical data between actual duration and actual workload against different labor allocations, subjected to the ideal workload W_1 = 10.0PM, is shown in Table 8.8. Any other specific cases can be determined by applying Eqs. 8.16 and 8.15.

In Table 8.8, the optimum labor allocation L_0 for each $T(r, L)$ curve is shaded where *T* reaches its minimum. The curves and trends of actual project durations against different labor allocations are illustrated in Fig. 8.5 known as the *Pigeon Diagram*. The curves indicate that a given optimum labor allocation L_0 for each curve will determine a certain minimum on the curve corresponding to the shortest project duration T_{min}.

Table 8.8
Actual Time T(r, L) and Actual Workload W(r, L) Distribution



L [P]	1	2	3	4	5	6	7	8	9	10
T(0) [M]	10	5	3.33	2.5	2	1.67	1.43	1.25	1.11	**1**
E(0)[PM]	10	10	10	10	10	10	10	10	10	10
T(0.001)	10	5.01	3.34	2.52	2.02	1.7	1.46	1.29	1.15	**1.05**
E(0.001)	10	10.02	10.03	10.08	10.1	10.2	10.22	10.32	10.35	10.5
T(0.01)	10	5.05	3.4	2.65	2.2	1.92	1.73	1.6	1.51	**1.45**
E(0.01)	10	10.1	10.2	10.6	11	11.52	12.11	12.8	13.59	14.5
T(0.05)	10	5.25	3.8	3.25	3	**2.92**	2.93	3	3.11	3.25
E(0.05)	10	10.5	11.4	13	15	17.52	20.51	24	27.99	32.5
T(0.1)	10	5.5	4.29	**4**	4	4.18	4.43	4.75	5.11	5.5
E(0.1)	10	11	12.87	16	20	25.1	31	38	46	55
T(0.2)	10	6	**5.28**	5.5	6	6.68	7.44	8.25	9.1	10
E(0.2)	10	12	15.84	22	30	40.1	52.1	66	81.9	100
T(0.3)	10	6.5	**6.27**	7	8	9.19	10.44	11.75	13.1	14.5
E(0.3)	10	13	18.81	28	40	55.14	73.08	94	117.9	145
T(0.4)	10	**7**	7.26	8.5	10	11.69	13.44	15.25	17.09	19
E(0.4)	10	14	21.78	34	50	70.14	94.08	122	153.81	190
T(0.5)	10	**7.5**	8.25	10	12	14.2	16.45	18.75	21.1	23.5
E(0.5)	10	15	24.75	40	60	85.2	115.2	150	189.9	235
T(1)	**10**	10	13.2	17.5	22	26.72	31.46	36.25	41.1	46
E(1)	10	20	39.6	70	110	160.32	220.2	290	369.9	460

Theorem 8.7 and the pigeon diagram reveal that the key hidden reasons that cause so many failures of large-scale software engineering projects are neither technical issues nor inadequate programming skills, but mainly because of the nonoptimal organization of coordinative work in complicated projects. In other words, nonoptimal labor allocation and/or incorrect order of project labor-duration determination are the black hole that results in the unexpected wastage of huge extra workload or resources in software engineering.

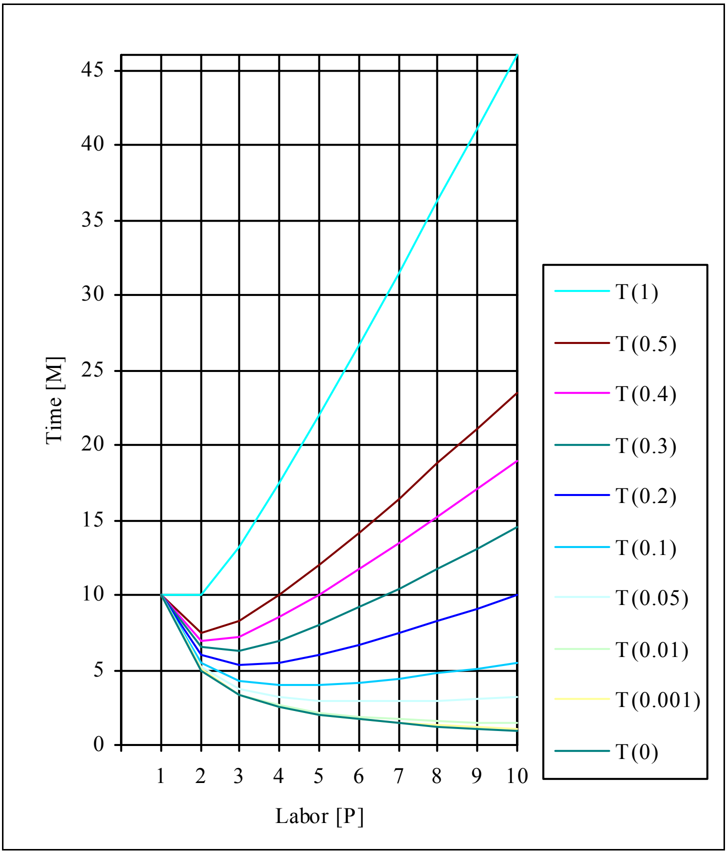


Figure 8.5 The *pigeon diagram*: actual time against number of labors (W_1 = 10PM)

Example 8.1 Assuming the ideal workload of a software engineering project is expected to be W_1 = 10.0PM and the organization has 1 to 10 persons available, determine the optimum allocation of labor L_0 and the shortest expected duration T_{min} for this project according to the 25th Law, given the average interpersonal coordination rate r = 10%.

Applying the 25th Law (Theorem 8.7), the optimum labor allocation is obtained as follows:

$$\begin{aligned} L_0(r) &= \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \\ &= \left\lceil \frac{1.414}{\sqrt{0.1}} \right\rceil \\ &= \lceil 1.414 / 0.316 \rceil \\ &= 5.0 \ [\text{P}] \end{aligned}$$

Replacing L_0 in Eq. 8.15 with the instantiation value $L_0(r)$, the shortest duration of the project can be determined as follows:

$$\begin{aligned} T_{\min} &= \frac{1}{2} W_1 (rL_0 \text{ - } r + \frac{2}{L_0}) \\ &= 0.5 \bullet 10.0 \bullet (0.1 \bullet 5.0 \text{ - } 0.1 + 2/5.0) \\ &= 5.0 \bullet 0.8 \\ &= 4.0 \text{ [M]} \end{aligned}$$

The above solutions indicate that, for a project with an expected 10.0PM workload by one person, the optimum labor allocation and shortest possible duration implemented by a coordinative project are 5.0 persons for 4.0 months, respectively, under r = 10%. This results in a real workload of W = 5.0P • 4.0M = 20.0PM by the coordinative team, where the gain is the reduction of project duration from 10.0M to 4.0M.

Example 8.2 Comparatively reanalyze Example 8.1 for a given average interpersonal coordination rate r = 50%.

The optimum solution yielded for the same project with ideally 10.0PM workload under r = 50% is to spend 7.5 months by 2.0 persons, which needs an expected workload W = 2.0P • 7.5M = 15.0PM.

On the basis of Theorem 8.7, an important corollary may be derived below, which clarifies the myth that whether labor L or duration T in a coordinative project is arbitrarily determinable as mentioned in the beginning of this section.

Corollary 8.1 An optimal work organization must be carried out in the following order for a given coordinative project:

a) To determine the optimum labor allocation L_0 (Eq. 8.16);
b) To obtain the shortest duration of the coordinative work T_{min} under L_0 (Eq. 8.15).

This is the third fundamental finding on laws of coordinative work organization [Wang, 2007d]. Corollary 8.1 indicates that the conventional common sense which believed that labor L or duration T in a coordinative project is arbitrarily determinable [Brooks, 1975/79; Boehm, 1981; Gantt, 1919; Tayler, 1911] would be a risky organizational practice that could easily result in a lot of waste of both resources and time without awareness in large coordinative engineering projects. Further analysis will be shown in Example 8.5 in Section 8.5.3.

8.5.3 THE MYTHICAL MAN-MONTH EXPLAINED

According to the organization laws of software engineering, it is proven that the trade-off between labor and time is possible under certain conditions as given in Theorem 8.7 and Corollary 8.1. This subsection analyzes the equivalence or the exchange rate between labor and time in coordinative work organization.

Definition 8.11 The *maximum gain of time ΔT* of a multi-person project is the difference between the time needed when only one person is allocated for the project and the shortest time T_{min} when labor is optimally allocated at L_0, i.e.:

$$\Delta T = T_1 - T_{min} \tag{8.19}$$

Definition 8.12 The *maximum increment of labor ΔL* of a multi-person project is the difference between the optimum allocated number of persons L_0 and the smallest group where $L_1 = 1$, i.e.:

$$\Delta L = L_0 - L_1, \quad L_1 \equiv 1 \tag{8.20}$$

The 23rd Principle of Software Engineering

Theorem 8.8 The *exchangeability from labor to time* states that the *exchange rate from labor to time* $\gamma_{L\sim T}$ in a coordinative work organization is determined by the ratio between the increment of time ΔT and the increment of labor ΔL, i.e.:

$$\begin{aligned} \gamma_{L\sim T} &= \frac{\Delta T}{\Delta L} \\ &= \frac{T_1 - T_{min}}{L_0 - L_1} \quad [\text{M/P}] \end{aligned} \tag{8.21}$$

The physical meaning of Eq. 8.21 is how many expected months may be gained or shortened in the schedule of a coordinative project by adding per labor into the project. It also explains how many months may be delayed if a person is withdrawn from the project.

Example 8.3 The exchange rate from labor to time $\gamma_{L\sim T}$ as given in Example 8.1 can be determined as follows:

$$\begin{aligned}\gamma_{L\sim T} &= \frac{T_1 \text{ - } T_{min}}{L_0\text{-}L_1} \\ &= \frac{10.0\text{-}4.0}{5.0\text{-}1.0} \\ &= 1.5 \ \ [\text{M/P}]\end{aligned}$$

The result shows that around the optimum labor allocation point, the increment of persons is most effective to progress a project. For this given example, the increment of each person can reduce the project duration for 1.5 months.

The 24th Principle of Software Engineering

Theorem 8.9 The *exchangeability from time to labor* states that the *exchange rate from time to labor* $\gamma_{T\sim L}$ in a coordinative work organization is determined by the ratio between the increment of labor ΔL and the increment of time ΔT, i.e.:

$$\begin{aligned}\gamma_{T\sim L} &= \frac{\Delta L}{\Delta T} \\ &= \frac{L_0 \text{ - } L_1}{T_1 \text{ - } T_{\min}} \ \ [\text{P/M}]\end{aligned} \qquad (8.22)$$

The physical meaning of Eq. 8.22 is how many persons' work is equivalent to a monthly increase/decrease in the project duration.

Example 8.4 The exchange rate from time to labor $\gamma_{T\sim L}$ as given in Example 8.3 can be determined as follows:

$$\begin{aligned}\gamma_{T\sim L} &= \frac{L_0\text{-}L_1}{T_1 \text{ - } T_{min}} \\ &= \frac{5.0\text{-}1.0}{10.0\text{-}4.0} \\ &= 0.67 \ \ [\text{P/M}]\end{aligned}$$

The result shows that, in the most effective case, an action to allow an extra month in the schedule is equivalent to the reducing of 0.67 person in the whole project lifecycle.

Comparing Eqs. 8.21 and 8.22, it can be observed that the two exchange rates are reciprocal. This leads to the following corollary.

Corollary 8.2 Labor and time are bidirectionally interchangeable or transformable in coordinative work organization under the constraints of Theorems 8.4 through 8.9, i.e.:

$$\gamma_{L\sim T} = \gamma^{-1}{}_{T\sim L} \tag{8.23}$$

Theorems 8.4 through 8.9 and related corollaries have answered the fundamental questions in coordinative work organization raised in the beginning of this section by rigorous reasoning and inferences. As a result, the insightful nature and inherent mechanisms of the problems in coordination work organization are systematically revealed and explained.

8.5.4 DECISION OPTIMIZATION IN SOFTWARE ENGINEERING

On the basis of the work coordination theory and laws presented in preceding sections, a number of decision optimization strategies may be derived towards the following objectives:

(a) The optimal labor allocations and the shortest project duration;

(b) The lowest workload and costs;

(c) The lowest overhead of interpersonal coordination.

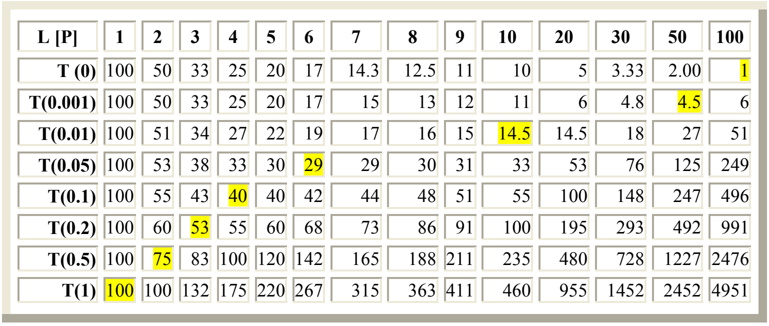
8.5.4.1 Optimization of Project Organization for the Shortest Duration

Corollary 8.3 The strategy for optimizing a project for the *shortest duration* is to set the project at the expected workload $W_{exp}(L_0, T_{min})$.

In the software industry, time to market is always a priority. Therefore, the shortest duration optimization strategy as provided in Corollary 8.3 is as practically important as that of the cost optimization strategy that will be described in the next subsection.

Table 8.8 and Fig. 8.5 described a small scale software engineering project with W_1 = 10.0PM. The case study on a large-scale project with W_1 = 100.0PM is summarized in Table 8.9. The optimal labor allocation L_0 for each $T(r, L)$ curve is shaded where T reaches its minimum.

Table 8.9
Actual Time and Actual Workload Distribution (W=100PM)



L [P]	1	2	3	4	5	6	7	8	9	10	20	30	50	100
T (0)	100	50	33	25	20	17	14.3	12.5	11	10	5	3.33	2.00	1
T(0.001)	100	50	33	25	20	17	15	13	12	11	6	4.8	4.5	6
T(0.01)	100	51	34	27	22	19	17	16	15	14.5	14.5	18	27	51
T(0.05)	100	53	38	33	30	29	29	30	31	33	53	76	125	249
T(0.1)	100	55	43	40	40	42	44	48	51	55	100	148	247	496
T(0.2)	100	60	53	55	60	68	73	86	91	100	195	293	492	991
T(0.5)	100	75	83	100	120	142	165	188	211	235	480	728	1227	2476
T(1)	100	100	132	175	220	267	315	363	411	460	955	1452	2452	4951

When W_1 = 100.0PM, the curves of actual project durations against different labor allocations are illustrated in Fig. 8.6. The optimum labor allocation for each curve can be found where $T(r, L)$ reaches its minimum.

Observing Table 8.9 and Fig. 8.6, it is noteworthy that no matter how large a software engineering project is, the optimum labor allocations are mainly ranged within 1 through 10 persons. Any other solutions are not an optimal labor allocation, because they do not result in the shortest project duration, rather than create a dramatically large actual workload.

This is the forth fundamental finding of the laws of coordinative work organization [Wang, 2007d] that leads to the following theorem.

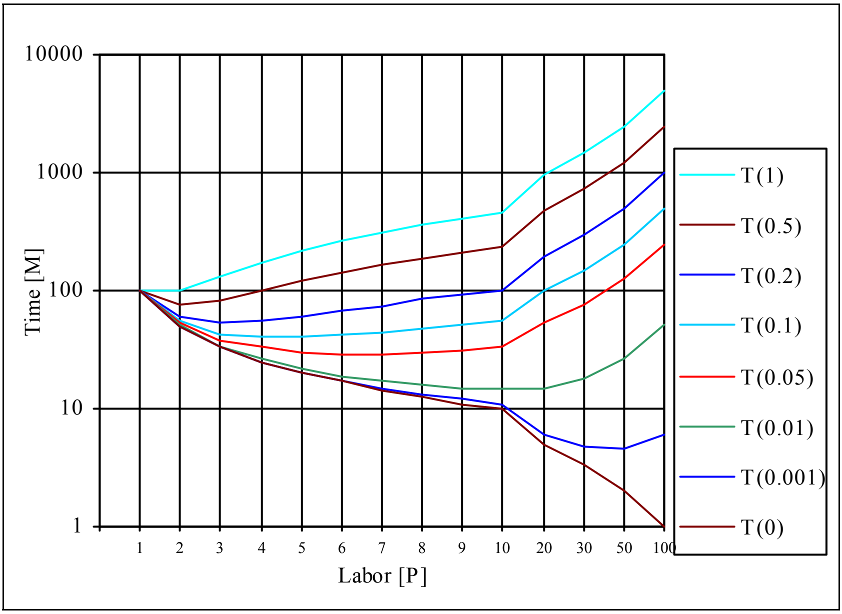


Figure 8.6 Actual time against number of labors when W_1 = 100PM

The 25th Principle of Software Engineering

Theorem 8.10 The *constraint on group size in coordinative work* states that there exists an upper limit of group size S_{max} in coordinative work organization in software engineering, i.e.:

$$\begin{aligned} S_{max} &= \max (L_0(r)) \\ &= 20 \text{ [P]} \end{aligned} \tag{8.24}$$

Therefore, large projects must be partitioned into multiple parallel groups that each of the groups obeys the same natural constraint.

Proof: Reformatting Eq. 8.16 the following expression is obtained:

$$r = \frac{2}{L_0^{\,2}} \tag{8.25}$$

For a required L_0 > 20.0P implies r < 0.005. Because an interpersonal coordination rate r less than 0.5% is impossible for carrying out any software engineering project, no project can be organized economically, efficiently, and technically sound in any form with more than 20.0 persons.

Ignoring the above natural constraints as described in Theorem 8.10, i.e., adding more labor into a maximum labor allocated project, will result in an exponentially increased actual workload as shown in the trends of the curves in Figs. 8.5 and 8.6, or in other words, a project failure in reality.

This is the fifth fundamental finding on laws of coordinative work organization. The truth has been evidenced by numerous failed projects in software engineering that involve hundreds of programmers in a single project [Brooks, 1975/95; Schonberger, 1981].

Corollary 8.4 A large software engineering project, $W_1 \geq 100.0$PM, with a higher coordination rate r, cannot be economically and feasibly completed with less than $T_{min} = 5.0$ M.

This corollary is a direct extension of Theorem 8.10. Usually, setting of $T_{min} \geq 10.0$M is safe for $W_1 \geq 100.0$PM.

It is noteworthy that Theorem 8.10 has ruled out the technical and economic feasibility for organizing large-scale software engineering project with a single large group. Therefore, the rational solution for real-world large-scale projects organization in software engineering is to adopt multi-groups in a hierarchical structure, rather than simply increasing the size of a single group. The only possible clue to do so is to divide the whole project into clearly partitioned and isolated parallel subprojects, provided that each of those subprojects should still obey the constraint on group sizes as given in Theorem 8.10.

Further discussions on large-scale project organization may be referred to Chapters 10 through 13, particularly Section 10.3.5 on System Organization Trees, Section 12.6.2 on the Formal Economic Model of Software Engineering Costs (FEMSEC), and Section 13.5.2 on Theory for Large-Scale Software Engineering Project Organization. These chapters explain how large-scale projects may be organized by hierarchical structures according to theories of system science, management science, economics, and sociology. Special attention may be paid to Law 45 on *FEMSEC* (Theorem 12.3), Law 47 on *organizational coordination efficiency* (Theorem 13.3), and Law 48 on *time-oriented optimization for large-scale project organization* (Theorem 13.4).

8.5.4.2 Optimization of Project Organization for the Lowest Effort /Cost

Using the data provided in Table 8.8, the curves of actual workloads with varying overhead rates against different number of labors are illustrated in Fig. 8.7.

Observing Fig. 8.7, it can be seen that all curves obey Theorem 8.5 that states $W_{min} = W_1$, i.e., the minimum effort/cost can only be reached when the project is carried out by one person.

> **Corollary 8.5** The strategy for optimizing a project for the *lowest effort/costs* is to set the project at $W_{min}(L_1, T_1)$.

According to Theorem 8.5 and Corollary 8.5, it is noteworthy that the group work in engineering organization is actually a generic mechanism and structure that allows the trading of time by labor as stated in the following corollary.

> **Corollary 8.6** *Coordinative work organization by groups* in engineering may gain time or shorten the project duration by using more man power, but cannot reduce the minimum project costs due to the natural constraint stated in Theorem 8.5, i.e., $W_{min} = W_1(L_1, T_1)$.

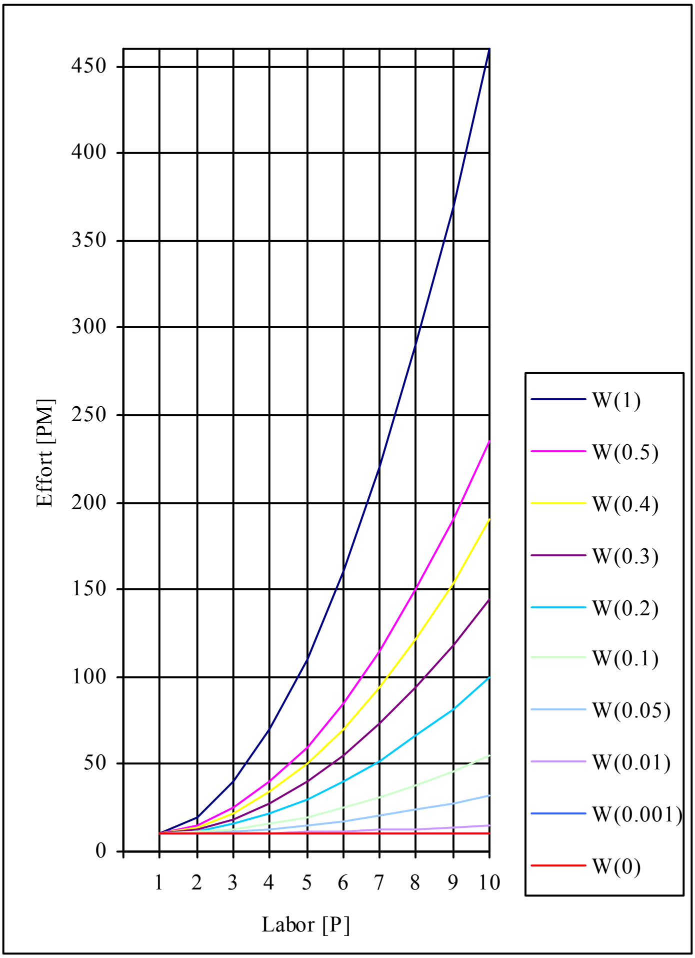


Figure 8.7 Actual effort against number of labors when W_1 = 10PM

Actually, most coordinative engineering projects enable organizers to pursue an optimization strategy for both the shortest project duration and the minimum costs at the same time. In this generic case, the optimal strategy is still W_{exp} as stated below.

Corollary 8.7 The strategy for optimization of a *coordinative project* for both the *shortest project duration* and the *minimum costs* is to set the project at $W_{exp}(L_0, T_{min})$. Otherwise, the waste of effort ΔW can be determined as:

$$\begin{aligned}\Delta W &= W - W_{\exp} \\ &= (L \bullet T) - (L_0 \bullet T_{\min}) \quad \text{[PM]}\end{aligned} \tag{8.26}$$

where W is the realized workload due to a nonoptimal work allocation.

Example 8.5 In Example 8.1, the optimal work allocation has been determined as $W_{\exp} = L_0 \bullet T_{\min} = 5.0 \bullet 4.0 = 20.0 \text{ [PM]}$. When the number of persons for this project is *subjectively* allocated by $L = 9.0$P, what will be the amount of the real workload W? How much effort would have been wasted due to this *nonoptimal* labor and time allocation?

According to Eqs. 8.14 and 8.12, the duration T and the real workload W of this project for given $L = 9.0$P and $r = 0.1$ can be determined, respectively, as follows:

$$\begin{aligned}T &= \frac{1}{2} W_1 (rL - r + \frac{2}{L}) \\ &= 0.5 \bullet 10 \bullet (0.1 \bullet 9.0 - 0.1 + 2/9.0) \\ &= 5.0 \bullet 1.02 \\ &= 5.1 \text{ [M]}\end{aligned}$$

$$\begin{aligned}W &= L \bullet T \\ &= 9.0 \bullet 5.1 \\ &= 45.9 \text{ [PM]}\end{aligned}$$

According to Corollary 8.7, due to the *nonoptimal* labor and time allocation in this coordinative project, the effort wasted, ΔW, can be expected as follows:

$$\begin{aligned}\Delta W &= W - W_{\exp} \\ &= 45.9 - 20.0 \\ &= 25.9 \text{ [PM]}\end{aligned}$$

The above example demonstrates that, due to the exponential curves of $W(r, L)$ as shown in Fig. 8.7, the average interpersonal coordination rate r is really the black hole that may consume a huge extra workload when the group size is not optimally determined in software engineering.

8.5.4.3 Optimization of Project Organization by Controlling the Interpersonal Coordination Rate

It is noteworthy, in Examples 8.1 and 8.2, that the interpersonal coordination rate r may significantly affect the optimization results of a project plan. A complicated engineering project, particularly in software engineering, may be easily turned to a failure due to bad organizational decisions with a nonoptimal labor allocation as stated below.

The 26th Principle of Software Engineering

Theorem 8.11 The *risk of nonoptimal work organization* states that the risks $\mathcal{R}$ due to irrational decisions of work organization are proportional to the coordination rate r in a project. That is, the higher the r, the higher the risk under nonoptimal labor allocation:

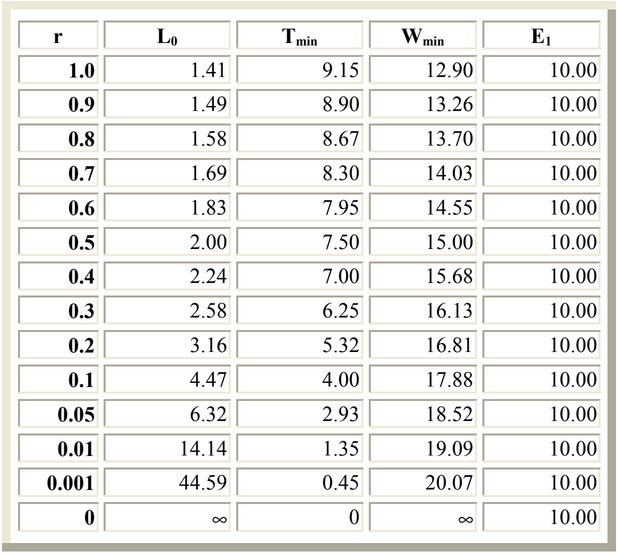
$$\mathcal{R} \propto r \tag{8.27}$$

A rule of thumb in optimal decision making by controlling r may be derived from the data shown in Table 8.10.

Corollary 8.8 The higher the interpersonal coordination rate r, the longer the possible shortest development duration, and the higher the total actual effort; and vice versa.

Observing Table 8.10 and Fig. 8.6, it is noteworthy that no matter how large a software engineering project is, the optimum labor allocations are mainly ranged within 1.0 through 20.0 persons (Theorem 8.10). Any other solution is not an optimum labor allocation, because they do not result in the shortest project duration rather than a creation of a dramatically large actual workload.

Table 8.10
The Optimum Labor Allocation and the Shortest Duration Minimum



r	L_0	T_{min}	W_{min}	E_1
1.0	1.41	9.15	12.90	10.00
0.9	1.49	8.90	13.26	10.00
0.8	1.58	8.67	13.70	10.00
0.7	1.69	8.30	14.03	10.00
0.6	1.83	7.95	14.55	10.00
0.5	2.00	7.50	15.00	10.00
0.4	2.24	7.00	15.68	10.00
0.3	2.58	6.25	16.13	10.00
0.2	3.16	5.32	16.81	10.00
0.1	4.47	4.00	17.88	10.00
0.05	6.32	2.93	18.52	10.00
0.01	14.14	1.35	19.09	10.00
0.001	44.59	0.45	20.07	10.00
0	∞	0	∞	10.00

Corollary 8.9 The optimal labor allocation in an individual software engineering project L_0 is ranged between 10 to 3 persons corresponding to the constrained coordination rate $1\% \leq r \leq 20\%$, i.e.:

$$\begin{cases} L_{0-min} = 3\ [\mathrm{P}], & r_{max} = 20\% \\ L_{0-\max} = 10\ [\mathrm{P}], & r_{min} = 1\% \end{cases} \tag{8.28}$$

Corollary 8.9 proves there is the natural constraint on the upper limit of group size, L_0, in software engineering project organization as stated in Theorem 8.10. No matter how much resources one would dispatch, the maximum team size of a software engineering project is constrained by the natural laws revealed in Theorems 8.4 through 8.11.

This section has addressed an age-old problem on coordinative project and group organization and optimization across many disciplines such as management science, operations theories, system science, software engineering, economics, and sociology. Conventional work has been focused on empirical studies of project planning and scheduling, and the inherent nature of the problem was hidden by too many trivial factors. This is the first time that it has been revealed that the interpersonal coordination rate in group is the black hole that has resulted in the failures of so many large-scale projects due to the exponential growing of unexpected actual workload under nonoptimal labor and work allocation. Based on the Wang's laws and theorems of rational work organization theories, a wide range of applications in optimal software engineering organization may be conducted, such as the decision optimization strategies in engineering coordination, and the determination of the best labor allocation, the shortest duration, and the lowest effort (cost) in project organization.

8.6 Empirical Software Engineering

It is recognized that software engineering requires both *theoretical* and *empirical* research. The former focuses on foundations and basic theories of software engineering; whilst the latter concentrates on fundamental principles, tools/environments, and best practices. The primary methodologies for empirical studies in software engineering encompass *case study*, *experiment*, *trial*, *benchmarking*, and *standardization*. These empirical methodologies may usually involve surveys and statistical analysis technologies as well.

8.6.1 SOFTWARE ENGINEERING CASE STUDIES

Case study is the first primary method for empirical studies in software engineering.

Definition 8.13 A *case study* is an intensive investigation and analysis of a particular technology, project, organization, or environment based on information obtained from a variety of sources such as interviews, surveys, documents, test or trial results, and archival records.

Case studies link a theory to practice, which allow conclusions to be drawn about the suitability of a given method on real-world problems in industrial scales. They also enable inductive inferences on a general theory based on a set of empirical data and applications.

Case studies may be used to validate a theory or method by empirical tests. They are also useful for providing a counter instance for a generally accepted principle. However, the drawback of case studies as an empirical method in software engineering is the difficulties of data collection and the generalization of findings via limited cases, particularly when they are positive but nonexhaustive.

8.6.2 SOFTWARE ENGINEERING EXPERIMENTS

Science as we know it today may be dated from the introduction of the experimental method during the Renaissance. It is a common means, in any scientific and engineering discipline, to gain empirical knowledge by conducting laboratorial and industrial experiments. The validation of empirical theories and best practices is by repetition and testing [Christensen, 1997].

Definition 8.14 *Experiment* is a fundamental research approach to identify causal relationships among variables under a controllable environment.

Rene Dubos perceived [Beveridge, 1957]: "The experiment serves two purposes, often independent one from the other: it allows the observation of new facts, hitherto either unsuspected, or not yet well defined; and it determines whether a working hypothesis fits the world of observable facts."

A major advantage of the experimental approach is that a causal relationship can be inferred with a high degree of control over irrelevant variables by either eliminating their influence or holding their influence constant. Another advantage is the ability to manipulate precisely one or more variables at one time in order to identify a possible causality.

The disadvantage of experiment is that laboratory findings are obtained in an artificial environment which precludes the generalization to a real-world situation. Hence, more *in vivo* experiments and in-field trials need to be adopted as described in the following subsection.

Experiment is seen vital for validating and assessing software engineering methodologies and techniques. Victor Basili and his colleagues promoted the experimental approach to software engineering [Basili et al., 1986/91]. Lawrence Votta and Adam Porter [Votta and Porter, 1995] proposed three types of experiments in software engineering as follows:

- Individual vs. groups
- Students vs. professional software developers
- In *vitro* vs. in *vivo* studies

where *in vitro* means in a controlled environment and *in vivo* stands for the way it really happens.

A generic software engineering experiment may be carried out in the following process: experimental design, conduct the experiment and collect data, result analysis, and interpretation results. Similar to this experimental process, Pierre Bourque and Alain Abran developed an experimental framework for software engineering research [Bourque and Abran, 1996]. The framework provides a model of software engineering experiment, which consists of the phases of *experiment definition, planning, operation, interpretation,* and *field testing*.

The experimental framework begins with definition of the problem and hypothesis of the experiment. The design of the experiment and measurement criteria is conducted in the planning phase. When the goal and hypothesis are defined and methods and measures are selected, the experiment can then be carried out in the operation phase. The experimental results are described and analyzed in the interpretation phase, no matter whether the results are either positive or negative in validating the hypothesis. A well designed software engineering experiment may result in new findings, theories, models, lessons learned or validation of a method or tool. The early phases of experiment may be carried out in laboratories or development centers. Then, field testing is necessary to examine the theory, method, or model in the industrial setting of software development organizations.

Systematic elimination, or vary one thing at a time, is a widely accepted principle in carrying out experiments. W.I.B. Beveridge observed that "It is when experiments go wrong that we find things out [Beveridge, 1957]," because a search for the unknown factor in an experiment may lead to an interesting discovery. Therefore he suggested that "A good maxim for the research man is: look out for the unexpected." This truth of the matter also lies in Pasteur's famous saying: "In the field of observation, chance favors only the prepared mind."

8.6.3 SOFTWARE ENGINEERING TRIALS

When a system is newly developed and tested, trial should be carried out in parallel with existing system before a decision may be made for putting the new system into operation as the main and active system. Statistics show that problems and failures may intensively occur during the turn over between the existing and new system, because of both technical and operational reasons.

Definition 8.15 *System trial* is a technology in empirical software engineering for safely putting a new system into operation.

The organization of software engineering trial is demonstrated in Fig. 8.8. Parallelism and output comparison between the new and old systems are key techniques for trials. Therefore, it is unwise to throw away the existing system and cease traditional practices before it is for sure that the new system is working as specified and expected in the field. Otherwise, there will be no back-up system and users could not resume the conventional practice whenever the new system should fail.

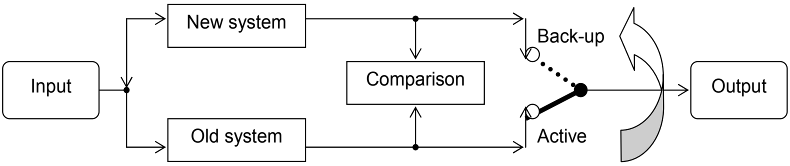


Figure 8.8 The architecture of software engineering trials

A typical procedure for system trial in empirical software engineering is provided in Fig. 8.9 [Wang, 2004c]. The six steps of trial should be followed carefully, particularly for the two evaluations when switching the new system from the back-up state to the active state, and when the old system is going to be shutdown and the traditional practices are going to be ceased.

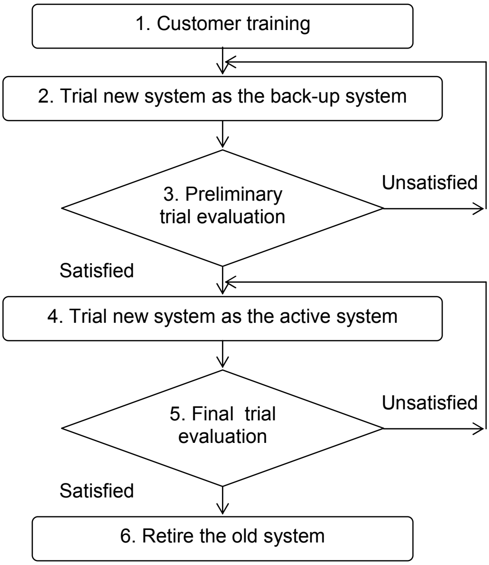

1. Customer training
2. Trial new system as the back-up system
3. Preliminary trial evaluation
Unsatisfied
Satisfied
4. Trial new system as the active system
5. Final trial evaluation
Unsatisfied
Satisfied
6. Retire the old system

Figure 8.9 The procedure of software engineering trials

It is noteworthy that the criteria for retiring the old or existing system are threefold: a) The trial is completed and the new system can produce the same output under the same operational conditions in site; b) The performance of the new system is satisfied; c) The users have been trained and have already been used to the new system. The trial and old system retirement procedure is particularly vital for safety-critical and real-time systems in software engineering.

Pilot or preliminary experiment is a typical way of engineering trial, which uses a small scale experiment at the laboratory to seek an indication as to whether a full scale field experiment is warranted. A pilot project is often applied in software engineering to test a new technology, architecture, tool, or system platform.

8.6.4 SOFTWARE ENGINEERING BENCHMARKING

Software engineering benchmarking is one of the important methodologies in software process engineering [Dutta et al., 1998; Wang, 2001e; Wang et al., 98b/99b/01; Wang and King, 2000a/00b; Chiew and Wang, 2002]. Benchmarking and benchmark-based process improvement is a cutting-edge technology in empirical software engineering for adaptive and relative process improvement.

Definition 8.16 A *benchmark* of a software engineering process system is a set of statistical reference data that represents the average performance and industrial norms of a set of processes in software engineering practices.

The key value of the software engineering benchmarking technologies is the establishment of the industrial norms and the quantitative measurement of common and best practices in different regions. On the basis of the benchmarks, software organizations are able to determine their current positions in a region, and to compare their practices against peers in the same sector. One of the major application areas of software engineering benchmarking is *benchmark-based process improvement.* Another application area of benchmarking is to enable software development organizations to compare and better manage their process improvement activities through benchmarking analysis.

A European process benchmark is developed by IBM (Europe) in the later 1990s, which encompasses 7 software engineering processes. Then, a series of worldwide surveys have been conducted by the author in order to establish a set of comprehensive benchmarks according to the Software Engineering Process Reference Model (SERPM) [Wang et al., 1998b/99a; Wang and King, 2000a] that consists of 51 processes characterized by 444

Base Process Activities (BPAs). This section describes the design and establishment of the two software engineering process benchmarks.

8.6.4.1 The IBM European Benchmarks on Software Engineering Practices

The IBM European benchmarks on software engineering provide seven high-level processes known as those of the organization, process, quality, methods, technology, planning, and measurements [IBM, 1996/97]. All processes defined in the benchmark are performed slightly above level 5 in the scale of 1 to 10, except the measurement process, as shown in Fig. 8.10.

In collaboration with IBM (Europe), the Center for Software Engineering at the Swedish Product Engineering Research Institute (IVF) conducted a national benchmarking survey to derive a national benchmark of software engineering practices for the Swedish software industry [IBM and IVF, 1997; Wang et al., 1998b/99b/99c/01]. The national benchmarks of software engineering practices are established against the seven IBM processes as illustrated in Fig. 8.10.

Generally, almost all processes of the Sweden software industry have been performed above level 5. Particularly the performances of the organization, quality, and measurement processes have exceeded the European benchmarks. The average performance level of all benchmarked processes is 5.2. The overall average value is used as a general national benchmark, where if a software development organization has a total mean process performance higher than 5.2, it has reached and/or exceeded the national norm in software development practices.

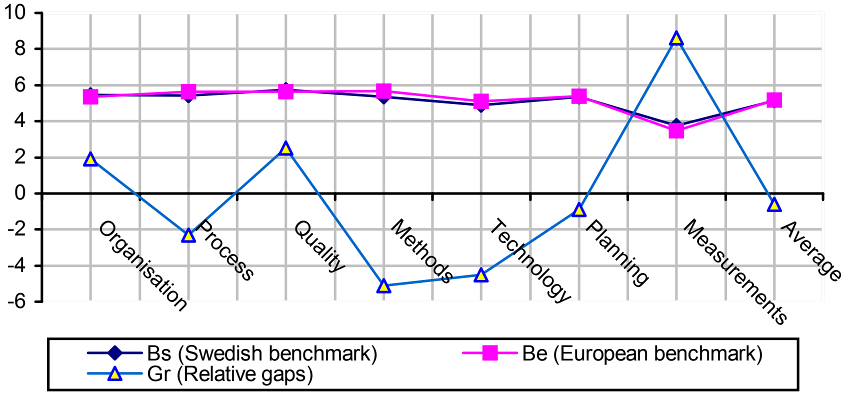


Figure 8.10 IBM (Europe) benchmark of software engineering practices

It is interesting to find that the Swedish national benchmarks are quite close to the European ones. The average difference of all processes is only -0.6%. This means the software engineering practices in the Swedish software industry have generally reached the best European practice level. Magnified gaps between the two benchmarks are shown by the G_r curve, which indicates the strengths and weaknesses of the Swedish software industry against the IBM European benchmarks.

8.6.4.2 The SEPRM Benchmarks on Software Engineering Processes

The IBM European benchmarks described in preceding subsection covered only 7 processes. There is thus a need to develop a comprehensive set of software engineering process benchmarks based on the SEPRM model [Wang et al., 1998b/99a; Wang and King, 2000a], which covers 51 processes characterized by quantitative process attributes [Wang, 2001e; Wang et al., 98b/99b/00b/01]. A high-level hierarchical structure of the SEPRM framework is shown in Fig. 11.22 and detailed descriptions of the process model will be given in Table 11.22.

SEPRM is a comprehensive software process model which possesses a superset of software engineering processes identified in current process models and standards such as CMM [Humphrey, 1988/89/95; Paulk et al., 1991/93/95], ISO 9001 [ISO 9001, 1989/94; ISO 9000-3, 1991], ISO/ICT 15504 [ISO/ICT, 2000; Dorling, Wang, et al., 1999]. SEPRM supports both goal-oriented and benchmark-based process establishment, assessment, and improvement.

The SEPRM software engineering process benchmarks derived based on a series of worldwide surveys [Wang et al., 1998b/99b] are shown in Fig. 8.11, where the number of a process corresponds to the serial number of processes in SEPRM as given in Table 11.22.

Based on the SEPRM benchmarks, the target capability levels for benchmark-based process improvement can be divided into three categories such as the basic, competitive, and advanced levels.

Definition 8.17 The *basic* level is the minimum level of process capability that a software organization should achieve in order to develop quality software according to the SEPRM benchmarks.

The basic level is suitable as a target for *initial* software organizations that are in the early stages of software process establishment and improvement.

Definition 8.18 The *competitive* level is an average level of process capability that ordinary software organizations have reached in software development according to the SEPRM benchmarks.

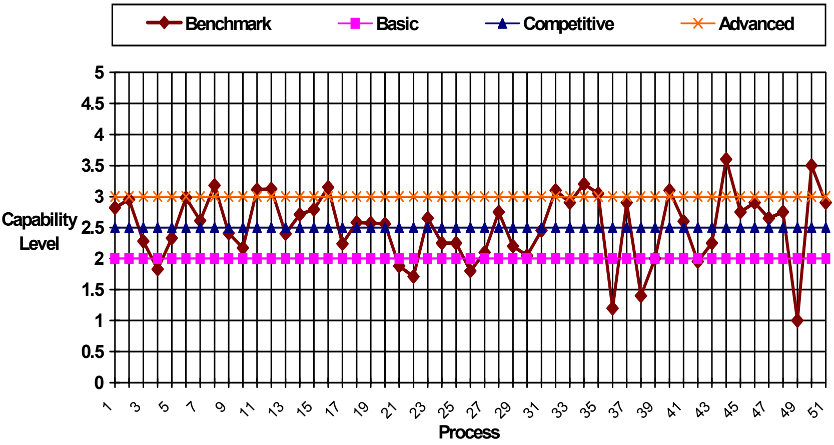


Figure 8.11 The SEPRM software engineering process capability benchmarks

The competitive level is suitable as a target for the *established* software organizations that pursue a stable software engineering process system and systematic process improvement.

Definition 8.19 The *advanced* level is the highest level of process capability that has been achieved by the top 10% of software development organizations according to the SEPRM benchmarks.

The advanced level is suitable as a target for the *experienced* software organizations that aim at optimizing existing process systems and producing high quality software for complicated and/or mission-critical systems.

One of the major application areas of software engineering benchmarking is *the benchmark-based process improvement.* Although the conventional *goal-based process improvement* technologies have been widely accepted, its philosophy of "the higher the better" has been questioned in practice. Particularly it is found that the determination of target capability levels for a specific organization tends to be virtual, infeasible, and sometimes overshot in the goal-based improvement approach. Benchmark-based process assessment and improvement [Wang, 2001e; Wang et al., 98b/99b/01; Wang and King, 2000a/00b; Chiew and Wang, 2002] provides a new approach to adaptive and relative process improvement based on a philosophy of "with a competitive margin just above the benchmarks." According to the benchmark-based process improvement method, the target capability levels of given software processes may be set relative to the benchmarks of the software industry, rather than to the virtually highest capability level as in a goal-based process approach.

Another application area of software engineering benchmarking is to enable software development organizations to compare and better manage their process improvement activities through benchmarking analysis. The key value of the software engineering benchmarking technologies is the establishment of the industrial norms and the quantitative measurement of common and best practices in different regions. On the basis of the benchmarks, software organizations are able to determine their current positions in a region, and to compare their practices against peers in the same sector.

8.6.5 SOFTWARE ENGINEERING STANDARDIZATION

Standardization is an attempt to regulate, integrate, and optimize existing methodologies and best practices in engineering research and in the industry. This subsection presents three categories of software engineering standards: the software development standards, software quality standards, and software engineering process standards.

Considering that a variety of software process models have been developed by international, national, professional, and industrial institutions in recent decades, standardization is a timely strategic action in this discipline. Standards are often arrived, however, at as the result of trade-offs between cutting-edge development and existing ones that are widely accepted as good practices. The active international standardization bodies in areas of software engineering, software process, and software quality are The International Organization for Standardization (ISO), and The Institute of Electrical and Electronics Engineers (IEEE).

Software engineering standards are not only records of best practices, but also means for reconciling successful practices with the underlying principles of the profession [Wang, 2001b]. Therefore, software engineering standards should be best practices validated by successful patterns of applications and rooted rationalized fundamental software engineering principles.

8.6.5.1 Software Development Standards

Historically, the standards for software engineering were focused on software development standards in the 1980s. Such standards include IEEE STD 1016 – Recommended Practice for Software Design Description, IEEE STD 830 – Guide to Software Requirements, IEEE STD 1012 – Software Verification and Validation Plans, IEEE STD 829 – Software Test Documentation, and IEEE STD 1008 – Software Unit Testing [IEEE, 1983/89; James, 1998].

In the 1990s, the effort of international software engineering standardization has been shifted to software quality and software engineering processes standards, which represents the new focus and industrial needs.

8.6.5.2 Software Quality Standards

Software quality system standardization is covered by the research inherent in the ISO Technical Committee (TC) 176 on quality management, quality assurance, and generic quality systems. A major serial standard developed by ISO TC176 is ISO 9000 (1987/91/93/94). ISO 9000 was published in 1987 and revised in 1994. ISO 9000 has been recognized worldwide for establishing quality systems. It is designed for quality management and assurance, which specifies the basic requirements for development, production, installation, and service at system and product levels. ISO 9000 provides a management organization approach, a product management system, and a development management system based on quality system principles.

Within the ISO 9000 suite, ISO 9001 and ISO 9000-3 are applicable to software quality systems for certifying the processes, products, and services within a software development organization according to the ISO 9000 model. ISO 9001 aims to set minimum requirements for a general quality management system. According to recent surveys [Wang and King, 2000b] the ISO 9001 model is the most popular quality system model in the global software industry. However, because ISO 9001 treats software development processes in the same way as any mass manufacturing system, its suitability to the creative and design-intensive software development processes is still uncertain.

ISO 9001 models a software quality system in 3 subsystems, 20 Main Topic Areas (MTAs), and 177 Management Issues (MIs) [ISO 9001, 1994; Jenner, 1995]. ISO 9001 provides a one-dimensional checklist-based software quality system assessment method. On the basis of feedback from both industries and researchers, a significant trend in ISO 9000/9001 revision (2000) is to shift from a check-list based quality system standard to a process-oriented one. The new version of ISO 9001 will include a process model based on Deming's Plan-Do-Check-Act cycle for product, service, and management processes. Also, ISO 9001:2000 will merge the 1994 versions of ISO 9001, ISO 9002, and ISO 9003 into a single and integrated standard. The 20 MTAs of ISO 9001 will be re-organized into 21 processes that are categorized into four primary processes: *management responsibility, resource management, product realization,* and *measurement, analysis and improvement*. New requirements in ISO 9001:2000, such as customer focus, establishment of measurable objectives, continual improvement, and evaluation of training effectiveness, are added for enabling quality process assessment and improvement.

Another important software quality standard is ISO 9126 – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use [ISO 9126, 1991]. ISO 9126 extends principles of quality control to software and summarizes the major characteristics and attributes of software quality. For an overview of ISO 9126 software quality model refer to Table 11.19.

ISO 9126 adopted a black-box philosophy that represents the customer's view of software products and systems. Further investigations [Dromey, 1995] argued that the ISO 9126 model has been focused only on the external attributes of software quality. Substantial internal attributes of software quality, such as of architecture, reuse description, coding styles, test completeness, run-time efficiency, resource usage efficiency, and exception handling, have not been modeled. In other words, the internal quality attributes of software may be characterized by the software engineering process-oriented standards and models. This observation explores an interesting connection between the process standardization and quality standardization in software engineering.

8.6.5.3 Software Engineering Process Standards

A number of software engineering standards and models have been developed in the last decade, such as TickIT [DTI, 1987; TickIT, 1987], ISO 9001 [ISO 9001, 1989/94], CMM [Humphrey, 1988/89/95; Paulk et al., 1991/93/95], ISO/IEC 12207 [ISO/IEC 12207, 1995], ISO/IEC 15504 [ISO/IEC 15504, 2000]. In addition, a number of regional and internal models have been adopted. According to a recent worldwide survey [Wang and King, 2000], ISO 9001 is the most popular standard in software engineering followed by CMM and ISO/IEC 15504. Some regional, internal, and industry sectors' process models, such as Trillium, also share a significant part of application in the software industry.

The process-related standards are developed within the international and professional standardization organizations such as the ISO/IEC JTC1/SC7 software/system engineering subcommittee and the IEEE. Significant standards coming forward are *inter alia*, ISO/IEC 12207 (1995) on Software Life Cycle Processes, ISO/IEC 15288 (1999) on System Life Cycle Processes, and ISO/IEC 15504 (2000) on Software Process Assessment and Capability Determination.

Major trends in software engineering process standardization have been considered to integrate the existing process-related standards and models, but standardization may also cover new process areas in software engineering. For instance, ISO/IEC 15504 is to align its process dimension to ISO/IEC 12207. In addition, extension for ISO/IEC 15504 has been proposed to cover some system life cycle processes, such as acquisition processes and broader system environment processes [Dorling, Wang et al., 1999; Wang and King, 2000a].

It is noteworthy that the list of software engineering standards is continuously expanding. As the evolution of software engineering theories, methodologies, and practices gets faster, more and more areas are expected to be covered by efforts in software engineering standardization. Candidate examples might be new standards for system requirement definition, domain knowledge infrastructures, software architectures/frameworks, and software engineering notations.

8.7 Summary

Engineering is a technological and organizational methodology and approach by which human beings can repetitively plan, design, develop, produce, maintain, and/or use complicated artefacts, in rigorous, systematic, efficient, and refining processes, that cannot be done by individuals.

Engineering is a **process** that converts theoretical concepts into useful applications to satisfy human needs. **Engineering approaches** and **generic engineering principles** form a part of the basic theoretical and empirical foundations of software engineering.

Software engineering is a discipline that adopts engineering approaches to develop large-scale software with high productivity, low cost, controllable quality, and measurable development schedules. Engineering principles for software engineering can be elicited on engineering objectives, organization, technology, professionalism, and domain characteristics.

This chapter has explored the generic engineering principles and engineering professionalism. The **organizational theory** for software engineering has developed, which reveals how software engineering projects may be optimally organized. A set of **empirical methodologies** for software engineering, such as case studies, experiments, trials, benchmarking, and standardization, has been described. As a result, the **engineering foundations** of software engineering have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, on *Engineering Foundations of Software Engineering*, readers have achieved the following strategic aims with the knowledge architecture as summarized below.

Chapter 8. Engineering Foundations of SE

■ Generic Engineering Approaches

- Engineering emerged from the industrial revolutions
- The generic scientific method
- Engineering vs. sciences
- Fundamental goals and constraints of engineering
- Generic engineering approaches
- The generic engineering maturity model (EMM)

■ Basic Engineering Principles

- Principles of engineering organization
- Principles of engineering technology
- Principles of engineering management
- Principles of engineering professionalism

■ Engineering Principles for Software Engineering

- The engineering characteristics of software engineering
- Division of labor
- Characteristics of software engineering in the engineering age
- Unique principles of software engineering
- Professionalism of software engineering

■ The Theory of Software Engineering Organization

- The characteristics of coordinative work in engineering
  - The mechanisms of coordinative workload and effort
  - The rate of interpersonal coordination
  - The overhead of interpersonal coordination
  - The nature of coordinative work in engineering
- Laws of work organization in software engineering
  - The laws of incompressibility of software engineering workload
  - The laws of interchangeability between labor and time in software engineering
  - The laws of the shortest duration of coordinative word in software engineering
- The mythical man-month explained
- Decision optimization in software engineering
  - Optimization of project organization for the shortest duration
  - Optimization of project organization for the lowest effort/cost
  - Optimization of project organization by controlling the interpersonal coordination rate

■ Empirical Software Engineering
- Software engineering case studies
- Software engineering experiments
- Software engineering trials
- Software engineering benchmarking
    - The IBM European benchmarks on software engineering practices
    - The SEPRM benchmarks on software engineering processes
- Software engineering standardization
    - Software development standards
    - Software quality standards
    - Software engineering process standards

SIGNIFICANT FINDINGS OF THIS CHAPTER

• **Engineering** is a concept of industrial organization emerged from the industrial revolution. Large software systems are among the most complex systems engineered by man.

• **Engineering approaches** to large-scale software development are those of *established methodologies, processes, tools, standards, organization methods, management methods,* and *quality assurance systems*.

• Every engineering discipline in the modern industries has been developed and matured in the same approach. The **generic engineering approach** is characterized by the following activities: a) To identify repeatable work processes; b) To identify standard and reusable components of products; c) To adopt division of labor; d) To equip specialized tools for the roles and processes; and e) To recognize management as a profession for organization of the processes and for co-ordination of the roles.

• Engineering disciplines emerged and developed in the industrial revolutions share the following common **principles for engineering organization**: a) Apply systematic processes; b) Adopt division of labor; c) Support co-operative work; d) Adopt quantitative measurement; e) Establish standards; f) Use tools and machinery; g) Plan actual schedule; h) Optimise resources allocation; i) Derive predictable outputs; and j) Seek controllable quality.

- The **coordinative work organization theory** (Theorem 8.4 through 8.11) for engineering project organization in general, and for software engineering project organization in particular, reveals that a set of **key factors** in coordinative engineering organization, such as *the optimal labor allocation, the shortest project duration, the minimum expected workload/effort/costs,* and *the interchangeability between labor and time*, is constrained by natural laws and a certain sequence for their determinations.

- The generic form of **workload** in coordinative work is always supplemented by an inevitable overhead, which is determined by the **interpersonal coordination rate *r*** in a multi-person project that formally and systematically explains **the mythic man-month** in software engineering (Theorem 8.7 and the **Pigeon diagram**).

- It is recognized that any theory, method, or technology has its own limitations and constraints. Therefore, to a certain extent, science and engineering are the searching of the maximum extent of general relations between entities, phenomena, and behaviors under a set of constraints.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Generic Engineering Approaches

- The great achievement of the **engineering approach to industrialization** results in extended human capability, improved productivity, and reduced skill requirement. The **industrial revolution** extended human *physical capability* by machines and engines. The **information revolution** is focused on the extension of human *intelligence*, *memory*, and the capacity for *information processing* by computers, communication networks, and robots.

- The **essence of engineering** is the organizational methodology for enabling coordinative team work in order to produce a complex product, or achieve a common goal, which could not be reached by individuals physically, technically, and/or economically.

- **Science** is a process of inquiry for generating a body of knowledge. The objectives of science are description, explanation, prediction, and control of category of objects under study. **Engineering** is an approach by which human being can repetitively plan, design, develop, produce, maintain, and/or use complicated artifacts in a rigorous, systematic, and refining process.

- **Scientists** explore what is; while **engineers** find out how to do. Science transfers information about nature into knowledge and theories; while engineering embodies knowledge into methodologies and products.

- The criteria that constitute a **good hypothesis** in scientific study are causality, originality, generality, predictability, and falsifiability.

- Sciences pursue *originality, simplicity,* and *generality* in principles and theories. In addition to the goals of sciences, engineering seeks *efficiency*, *productivity*, and *quality* in implementation of scientific principles and theories into repetitive and mass production.

- The law of **conservation of basic engineering constraints**, states that the three basic constraints of engineering objectives known as time (*T*), costs (*C*), and utility (*U*) are conservative in a given engineering context (Eq. 8.12).

- The **engineering maturity model** (EMM) of applied engineering disciplines states that there are four levels of engineering maturity, known as the phases of *emergence, art, engineering,* and *post-engineering*, in the evolution of any engineering discipline.

Basic Engineering Principles

- The **fundamental engineering principles** can be classified into the principles of *engineering objectives, organization, technology, management,* and *professionalism*.

- A significant part of the common engineering principles has not been systematically adopted and implemented in software engineering yet.

Engineering Principles for Software Engineering

- **Special characteristics of software engineering** are identified as:

  a) Intangible objects and work products, and intricate relations and interactions between them;

  b) Problem domain is infinite including all application areas of all existing engineering disciplines;

c) Software engineering is design intensive opposed to repetitive production;

d) Application development is one-off activity;

e) Development processes are stable and repetitive;

f) A software design and implementation is only one of all possible solutions for a real-world problem on the basis of tradeoffs and constraints;

g) Software engineering needs new forms of descriptive mathematics that are different from current analytic ones.

The Theory of Software Engineering Organization

- The **coordinative workload in engineering** states that the *actual workload* W of a coordinative project is a function of the average interpersonal coordination rate r and the number of labor L in the project (Theorem 8.4).

- The **incompressible workload** states that a given ideal workload W_1 in software engineering can not be compressed by any kind of labor allocation, i.e.: $W \geq W_1 = W_{min}$ (Theorem 8.5).

- The **interchangeability of labor and time** (ILT) states that, for a given workload W, labor L and duration T are transformable under the condition of Eq. 8.14 (Theorem 8.6).

- The **shortest duration of coordinative work** states that there exists the *shortest duration* T_{min} under the *optimum labor allocation* L_0 for a given ideal workload W_1 with a certain interpersonal coordination rate r (Theorem 8.7)

- An **optimal work organization** must be carried out in the following order for a given coordinative project: a) To determine the optimal labor allocation L_0 (Eq. 8.16); and b) To obtain the shortest duration of the coordinative work T_{min} under L_0 (Eq. 8.15).

- **Labor and time** are **bidirectionally interchangeable** or transformable in coordinative work organization under the constraints of Theorems 8.4 through 8.9.

- The **constraint on group size** in coordinative work states that there exists an upper limit of group size S_{max} in coordinative work organization in software engineering, i.e., S_{max} = max ($L_0(r)$) = 20 [P]. Therefore, large projects must be partitioned into multiple parallel groups that each of the groups obeys the same natural constraint. (Theorem 8.10)

- The **strategy for optimization of a coordinative project** for both the shortest duration and the lowest cost is to set the project at $W_{exp}(L_0, T_{min})$. Otherwise, the waste of effort ΔW can be determined as $\Delta W = W - W_{\exp} = (L \bullet T) - (L_0 \bullet T_{\min}) \text{ [PM]}$, where W is the realized workload due to a nonoptimal work allocation (Corollary 8.7).

- The **risk of nonoptimal work organization** states that the risks $\mathcal{R}$ due to irrational decisions of work organization are proportional to the coordination rate *r* in a project. That is, the higher the *r*, the higher the risk under nonoptimal labor allocation, i.e., $\mathcal{R} \propto r$ (Theorem 8.11).

Empirical Software Engineering

- **Theoretical software engineering** focuses on foundations and basic theories of software engineering; whilst **empirical software engineering** concentrates on heuristic principles, tools/environments, and best practices. The primary methodologies for empirical software engineering are *case study*, *experiment*, *trial*, *benchmarking*, and *standardization*.

- A **case study** is an intensive investigation and analysis of a particular technology, project, organization, or environment based on information obtained from a variety of sources such as interviews, surveys, documents, test or trial results, and archival records.

- **Experiment** is a fundamental research approach to identify causal relationships among variables under a controllable environment.

- **System trial** is a technology in empirical software engineering for safely putting a new system into operation.

- A **benchmark** of a software engineering process system is a set of statistical reference data that represents the average performance and industrial norms of a set of processes in software engineering practices.

- **Standardization** is an attempt to regulate, integrate, and optimize existing methodologies and best practices in engineering research and in the industry. This subsection presents three categories of software engineering standards: the software development standards, software quality standards, and software engineering process standards.

Questions and Research Opportunities

8.1 What is engineering and why did it emerge from the industrial revolutions?

8.2 What are the differences between science and engineering in terms of their objectives, methodologies, criteria, and embodied results?

8.3 Compare and contrast the generic engineering approaches and the generic scientific method.

8.4 What are the generic engineering principles? How can software engineering learn from them?

8.5 What is the nature of software engineering? Is software engineering unique or special in relation to the other engineering disciplines?

8.6 Is software development an engineering discipline? Are software developers engineering professionals? May both answers to the first two questions not be the same?

8.7 According to the EMM model (Theorem 8.3), discuss the following: a) Why should software engineering be considered as an engineering discipline even if it is immature at the given time? b) What will software engineering lead to (give birth) when it is matured?

8.8 Can software engineering methodologies and approaches be applied to other engineering disciplines? Try to provide an example.

8.9 Following Ex. 8.4, discuss how to implement the generic engineering principles in software engineering.

8.10 What is the fundamental difference between the objects under study in traditional engineering disciplines and software engineering?

8.11 In Section 3.5.1 eight fundamental cognitive characteristics of software engineering are identified. Choose one of them and explain its impact on the engineering of software development.

8.12 The three constraints of the basic objectives of engineering, known as *Time (T), Costs (C),* and *Utility (U),* are conservative in a given engineering context according to Theorem 8.2. When the duration of a software project should be reduced, what would be the consequences on the other constraints?

8.13 According to Theorems 8.4 and 8.7, explain the organizational reasons of a significantly high failure rate of software engineering projects in history?

8.14 When 6 programmers work together in a project team, what is the number of pairwise relations n? If the number of programmers is increased to 30, what is the impact on n?

8.15 How can programmer group(s) be optimally organized in large-scale software engineering projects? Discuss the situations with less than 10 and more than 20 persons in a group, respectively.

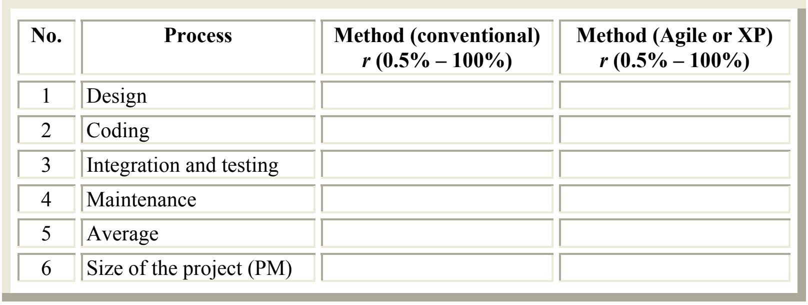
8.16 What are the profound factors that constitute coordinative team work in software engineering?

8.17 Whether labor L or duration T is arbitrarily determinable for a given workload W in a coordinative software engineering project? What is the law that constrains the determination of them?

8.18 Are time T and labor L interchangeable for a given workload W in software engineering? If so, what would be the constraints for the interchangeability between them?

8.19 The *average interpersonal coordination rate r* (Definition 8.8) can be determined as follows: $r = t' / (t + t')$, where t is the total time an individual used on pure software development tasks; t' is the total time one spent in work that is not directly used in software development such as meetings, discussions, communications, learning, training, travel, in site testing, services, etc.

Fill in the following form using your data during work, practice, or study (if you are a student), and analyze your findings.



No.	Process	Method (conventional) *r* (0.5% – 100%)	Method (Agile or XP) *r* (0.5% – 100%)
1	Design		
2	Coding		
3	Integration and testing		
4	Maintenance		
5	Average		
6	Size of the project (PM)		

8.20 Assume the ideal workload of a software engineering project is expected to be W_1 = 100.0PM, the organization has up to 30 persons available, and the average interpersonal cooperation rate r = 20%.

(a) According to the *ILT* Law (Theorem 8.6), determine the optimal allocation of labor L_0 and then the shortest expected project duration T_{min} for this project.

(b) What is the expected workload W_{exp} that this project may achieve under the optimal labor allocation and the shortest project duration as obtained in (a)?

(c) When the number of persons for this project is *subjectively* allocated as L = *25*P, what would be the resulted real workload (W)?

(d) How much effort would be wasted in solution (c) due to the *nonoptimal* labor and time allocation? (*Hint:* Consider $\Delta W = W - W_{exp}$).

8.21 According to Theorem 8.6, given W_1 = 6.0PM, W_{min} = 12.0PM, and r = 0.5, when 10 persons are *subjectively* allocated to a project, how much effort would be wasted in this project due to the *nonoptimal* labor and time allocation?

8.22 What is the black hole in coordinative work organization that would result in the unexpected wastage of huge extra workload and resources in software engineering?

8.23 According to Law 25 (Theorem 8.7), explain why the key reasons that cause so many failures of large-scale software engineering projects are not purely technical ones, but mainly organizational reasons that result in nonoptimal coordinative work organization in complicated software engineering projects.

8.24 Why should an optimal work organization be determined in the rational order from the optimal labor allocation L_0 to the shortest project duration T_{min}? What would be the consequences if this law is not obeyed in project planning and organization?

8.25 Five empirical methodologies for software engineering, known as *cases studies, experiments, trials, benchmarking,* and *standardization*, have been discussed in Section 8.6. Try to develop a table to compare the advantages and disadvantages of these empirical methodologies in software engineering research and practice.

8.26 How to implement division of labor in software engineering?

8.27 According to the procedure of software engineering trials, explain why a software system trial should be divided into four phases, and what the conditions of each transition to the next phase are.

8.28 Given a heuristic principle of software engineering, such as *review/inspection* or *system engineering*, which empirical method(s) you would like to use in order to validate it in software engineering. Why?

8.29 Why do *ethics* and *professionalism* play important roles in software engineering?

8.30 Discuss the following software engineering situations that require good ethical judgment of a software engineer [Vliet, 2000]:

Suppose you are testing a part of a big software system. You find quite a few errors and you are certainly not ready to deliver. However, your manager is pressing you. The schedule has already slipped by quite a few weeks. Your manager in turn is pressed by his boss. The customer is eagerly awaiting delivery of the system. Your manager suggests that you should deliver the system as is, continue testing, and replace the system by a better version within the next month.

- How would you react to this scheme?
- Would you simply give up?
- Argue with your manager?
- Go to his boss?
- Go to the customer?

8.31 An argument on software engineering is that because both professionals and amateurs can write programs, programming has no scientific foundations. Do you agree with this observation? Why?

8.32 Read the following classic article in software engineering:

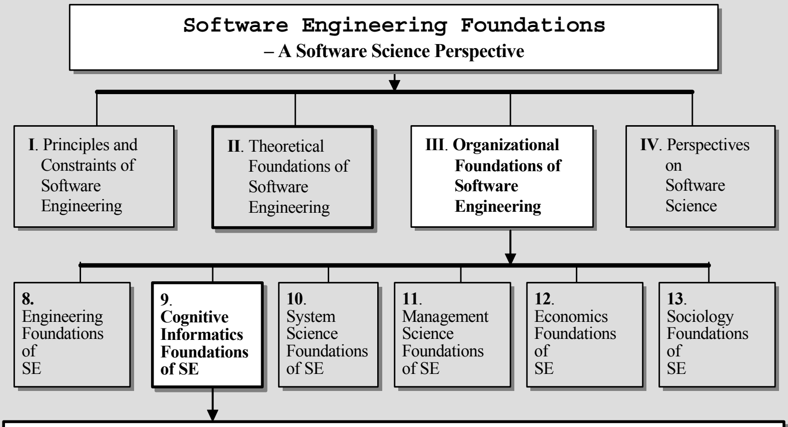
> David L. Parnas (1995), On ICSE's 'Most Influential' Papers, ACM Software Engineering Notes, 20(3), pp. 29-32.

Discuss the following topics in a group:

- About the author.
- What are the major problems in software engineering research according to the author?
- What is the relationship between proven theories and empirical methodologies of software engineering?
- What conclusions of the article interested you? Why?
- Your argument(s) or counter-points on any of the conclusions derived in this article.

Chapter 9

COGNITIVE INFORMATICS FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE

9. Cognitive Informatics Foundations of SE

Knowledge Structure

❍ Cognitive informatics

- Cognitive philosophy
- The emergence of cognitive informatics
- The theoretical framework of cognitive informatics
- Neural informatics foundations of the brain

❍ Cognitive informatics models of the brain

- The Layered Reference Model of the Brain (LRMB)
- Cognitive properties of internal information
- Natural intelligence vs. artificial intelligence
- The cognitive model of the brain

❍ Cognitive informatics of knowledge representation

- The Hierarchical Neural Cluster (HNC) model of memory
- The Object-Attribute-Relation (OAR) model of internal information representation
- The extended OAR model of the brain
- The cognitive mechanisms of long-term memories
- The memory capacity of human brain

❍ Cognitive informatics for software engineering

- Cognitive informatics properties of SE
- The cognitive foundation of software comprehension
- SE psychology
- SE skills and experiences

❍ Cognitive complexity of software

- The relative cognitive weights of generic software structures
- Psychological experiments on the cognitive weights
- Calibration of the relative cognitive weights of BCS's

Learning Objectives

- To understand the need for extending informatics to the study of the brain – a profound problem shared by almost all science and engineering disciplines.
- To be aware of the neural informatics foundation of cognitive informatics.
- To recognize the theoretical framework of cognitive informatics.
- To understand the cognitive informatics models of the brain (i.e., LRMB, intelligence model of the brain, logical model of the brain).
- To understand the cognitive models of internal knowledge representation (i.e., HNC, OAR, EOAR).
- To appreciate the cognitive informatics foundations of software engineering for addressing the cognitive constraints of software engineering.
- To understand the cognitive complexity of software and the relative cognitive weights of BCS's.

"Knowingness is a cognition, so is unknowingness."

Confucian (551 – 479BC)

"I think, therefore I am"

Rene Descartes (1596-1650)

"Elaborate apparatus plays an important part in the science of today, but I sometimes wonder if we are not inclined to forget that the most important instrument in research must always be the mind of man."

W.I.B. Beveridge (1957)

9.1 Introduction

In Chapter 7, it is reviewed that information science or informatics has developed from the classical information theory, contemporary informatics, to cognitive informatics in the past half century. *Cognitive informatics* is coined by Yingxu Wang in 2002 in the IEEE First International Conference on Cognitive Informatics (ICCI'02) [Wang, 2002d; Wang et al. 2002a]. Since then, it has been widely received as an emerging cutting-edge discipline that forges links between computing, cognitive psychology, information science, and software engineering.

It is identified that information is the product of either natural intelligence or machine intelligence, and the third essence of the natural and the perceived world. In computing, software engineering, informatics, intelligence science, and psychology, almost all hard problems yet to be solved share a common root in the understanding of the mechanisms of natural intelligence and the cognitive processes of the brain. This leads to the emerging discipline of research known as cognitive informatics [Wang, 2002d/02e/03a/03b/06b/06j/07a/07b; Wang and Wang, 2006; Wang and Kinsner, 2006; Wang et al. 2002a/06].

Cognitive informatics is the transdisciplinary study of cognitive and information science that investigates into the internal information processing mechanisms and processes of the natural intelligence – human brains and minds.

The study on cognitive informatics is triggered by the fundamental wonder of mankind to understand the brain – a quest is certainly as long as the human history itself. Studies of the brain were originally conducted in the domain of philosophy and psychology. Though, it is noteworthy that psychology was a part of philosophy in the early phase development of natural sciences [Leahey, 1980; Wilson and Keil, 2001]. Psyche means *spirit* or *soul* in both Greek and Latin. In 2500BC, the ancient Egyptians believed that the heart was the true seat of intelligence. It was not until 450BC, Greek physician Alcmaeon found that the brain is the central organ of sensations based on anatomic dissections of animals.

Psychological thoughts can be traced back to Plato and Aristotle in 400 to 320BC [Plato, 1961/75; Aristotle, 1925]. Plato (428-347BC), Greek philosopher, observed that philosophy begins in human wonder, a powerful desire to understand the world, not merely to act in it as animals do. Aristotle (394-322 BC), a quester for the nature, perceived psychology as the study of the soul, 'the form of a natural body having life potentially within it,' which differentiates the animate world from the inanimate one. However, Aristotle had also believed till 335BC that the organ of thought and sensation is the heart, and the brain is a radiator to cool it. After nearly half a century, Herophilus and Erasistratus first dissected a human body and found the nervous system of the brain in 300BC.

Psychology, as we know it, began with Rene Descartes (1596-1650), who in 1649 proposed that the brain functions like a machine [Descartes, 1979]. Descartes had also created a framework for thinking about mind and body for philosophers and psychologists. Then, after about 200 years, Wilhelm Wundt (1832-1920) found psychology as a science discipline by initiating a link between physiology and philosophy via an experimental approach in 1873 [Wundt, 1873].

The study on cognitive informatics is also rooted in the quests in life science, natural intelligence, and their interactions with machines and artificial intelligence.

Definition 9.1 A *living organism* is a physicochemical structure and process that possesses a high degree of complexity and is capable of self-regulation, metabolism, and perpetuates itself through time.

The primarily attributes of living organisms are featured as the following general characteristics [Fried and Hademenos, 1999]:

- *Movement:* The motions within the organisms or locomotion of the organisms through its environment.
- *Irritability:* The capacity to respond in a characteristic manner to stimuli in the internal or external environment.
- *Growth:* The ability to increase their mass of living materials by assimilating new materials from the environment.

- *Adaptability:* The tendency to undergo or institute changes in their structure, function, or behavior that improve their capacity to survive in a particular environment.
- *Reproduction:* The ability to reproduce new individuals like themselves.
- *Lifecycle:* The existence of a clear lifecycle and life span for a given generation.

According to the IME model (Theorem 1.2), information theories discussed in Chapter 7 can be classified as the *external* informatics. Complementary to it, there is a whole range of new research areas known as cognitive informatics that studies the mechanisms of internal informatics inside the brain.

Leveson stated that "If our problems in building and interacting with complex systems are really rooted in intellectual manageability and human limits in managing complexity, then we will need to stretch these limits to build ever more complex systems [Leveson, 1995]." Large-scale software systems are highly complicated systems that humans have ever been handled or experienced before. Software is a unique abstract artifact that does not obey any known physical laws. However, it is recognized that software should be constrained by the laws of cognitive informatics, mathematics, and systems as explored in this book. This section explores theories of cognitive informatics and its potential impacts on, and applications in, information-based sciences and engineering disciplines, particularly software engineering. The mathematical and system foundations of software engineering are presented in Chapters 4 and 10, respectively.

This chapter describes the cognitive informatics and intelligent behavioral metaphor of software and software engineering. In the remainder of this chapter, the cognitive informatics foundations of software engineering will be presented in five sections. Section 9.2 introduces the new transdisciplinary field of study known as cognitive informatics. Section 9.3 develops cognitive informatics models of the brain, such as the layered reference model of the brain, the cognitive models of memories, and the cognitive model of natural intelligence. Section 9.4 explores the cognitive model of internal information presentation in the brain, particularly the object-attribute-relation model. Section 9.5 presents the cognitive informatics foundations of software engineering, which leads to the understanding and formal measurement of the cognitive complexity of software systems in Section 9.6.

9.2 Cognitive Informatics

Cognitive informatics is the science for brain and natural information processing. A profound problem in natural and engineering sciences is cognitive informatics, which studies the mechanisms and processes of the brain in information processing, and the understanding of the natural intelligence. The cognitive informatics foundations are shared by multidisciplinary studies, such as philosophy, psychology, computing, software science (engineering), informatics, neuroscience, neurobiology, mathematics, and linguistics.

9.2.1 COGNITIVE PHILOSOPHY

It is interesting to note that philosophers all over the world shared similar perceptions towards cognition, information, and the natural intelligence. A historical story is told of two Chinese philosophers walking around a lake and seeing fishes swimming and jumping lively and freely in the water. The following dialogue took place (Zhuang Tsui, 369BC - 286BC, Chuang Tsui • Outer Chapters, Chapter 17, Autumn Water):

Philosopher A: "The fishes must be very happy because they are lively playing in the lake.

Philosopher B: "Well, you are not a fish. How do you know that they are happy?

Philosopher A: "Then, you are not me. How do you know that I don't know the feeling of the fishes.

Philosopher B: "Just as I am not you, I don't know you; You are not a fish, therefore you don't know the feeling of the fishes.

Philosopher A: "As you claim that I don't know something, you implied you know what I know. Therefore, I am able to perceive what the meaning of the fishes' behavior is.

This is a perfect situation to demonstrate the objectives of the study in cognitive informatics – how human beings acquire, process, interpret, and express information by using the brain, and how the minds of different individuals are understood.

Rene Descartes (1596-1650), French philosopher and mathematician, believed that the commonly accepted knowledge would be doubtful because of the subjective nature of human senses. He attempted to rebuild human knowledge structure using the fundamental concept known as 'cogito ergo sum' (*I think, therefore I am*). He once wrote:

> "If you would be a real seeker after truth, it is necessary that at least once in your life you doubt, as far as possible, all things [Descartes, 1979]."

The subjectivity of the perceived world as Descartes expressed has been formally described in Theorem 1.2 known as the *generic worldview* in the IME model.

The brain is perhaps the last thing in the natural world yet to be explored and understood. One of the most interesting findings in cognitive informatics is that so many science and engineering disciplines, such as informatics, computing, software engineering, and cognitive sciences, share a common root problem – how the natural intelligence processes information.

Along with the development of natural sciences, particularly psychology, cognitive science, and cognitive informatics, there are a number of significant scientific discoveries as shown in Table 9.1, which shed light on the nature of human beings and, in the same time, blow on human self-esteem [Leahey, 1980; Wang, 2003b].

Table 9.1
Scientific Discoveries Impacting on Human Esteem

No.	Time	Theory	Description
The 1st blow	1473-1543	*Universe view* (Nicholas Copernicus)	Human beings did not live at the center of the universe.
The 2nd blow	1809-1882	*Evolution* (Charles Darwin)	Human beings were part of nature – being animals like any other species.
The 3rd blow	1856-1939	*Nonconsciousness* (Sigmund Freud)	The human ego is not master in its own house, since many human behaviors are determined by nonconscious life functions.
The 4th blow	2003	*Partial autonomous* (The author of this book)	Human beings do not fully behave autonomously. The natural intelligence can be classified into those of *reflective, perceptive, cognitive,* and *instructive*, where the reflective and instructive intelligence are external event-driven and environment-dependent.

In Table 9.1, Darwin's finding from the viewpoint of cognitive informatics is that the human brain at the basic level has no difference from other animal species. However, it possesses the following advantages as presented in Theorems 9.1 and 9.2 [Wang, 2003b; Wang and Wang, 2006].

The 26th Law of Software Engineering

Theorem 9.1 The *quantitative advantage of human brain* states that the magnitude of the memory capacity of the brain is tremendously larger than that of the closest species.

The 27th Law of Software Engineering

Theorem 9.2 The *qualitative advantage of human brain* states that the possession of the abstract layer of memory and the abstract reasoning capacity makes human brain profoundly powerful on the basis of the quantitative advantage.

Although Freud's theory indicates that many fundamental human behaviors are nonconscious and there was no direct access and control on them by the conscious minds [Freud, 1895], psychology and cognitive science still maintain a common belief that an individual human being is at least autonomous and behaves purely on own desires and motivations. However, this assertion may be doubt in cognitive informatics.

The *basic life function* of the human brain is information processing. Although the brain may be stimulated by both external and internal information, the internal information is previously acquired from external sources. The willingness-driven mechanism of human behaviors was thought to be purely determined by internal information and conditions such as goals, desires, and emotions. Based on this perception, an individual may be considered as an autonomous human being. However, all willingness-driven behaviors as the nonconscious life functions as identified by Freud are synthetically dependent on the historically cumulated external events, information, status, and current internal physiological and subconscious conditions. That is, the universal causality on the study of human brains and cognitive behaviors can still be preserved, even some of the willingness-driven cause-effects are not so obvious due to long-term and indirect feedback in the human memory [Wang, 2003b].

The willingness-, event- and time-driven life functions and their cognitive processes may be formally described by RTPA [Wang, 2002a], which provides an expressive mathematical means for rigorously describing the meta cognitive life-functions such as abstraction, search, quantification, categorization, and memorization, as well as higher cognitive life-functions such as recognition, imagination, comprehension, inference, learning, and problem solving.

This section shows the philosophical differences between cognitive informatics and AI research. The philosophy of AI is based on the principle of Turing tests for evaluating functional equivalence between machine and human intelligence. AI attempts to answer how a computer can do what a human being does [Wang, 2002d]. However, before the fundamental mechanisms of natural intelligence are well understood, AI would be a search in dark without theoretical references.

The philosophy of cognitive informatics is autonomic and *supplementary computing* that explores theories and techniques addressing:

- What are the fundamental mechanisms of natural intelligence of the brain?
- How does internal information be represented, processed, and utilized?
- What can a computer do while human beings cannot?
- What can a computer do better than human beings?
- What will be the next generation architectures of computers that may learn from the human brains and natural intelligence?

In recent genome research people expect that the decoding and probing of human genomes will solve almost all problems and answer almost all questions about the myths of the natural intelligence. Although the aim is important and encouraging, computer scientists would doubt this promising prediction. This is based on the basic reductionism of science and the following observations: Although the details of computer circuitry are fully observable at the bottom level, i.e., at the gate even the molecular level, only seeing computers as the low-level structures would not help explaining the mechanisms of computing rather than get lost in an extremely large number of interconnected similar elements, if the high-level functional architectures and logical mechanisms of computers were unknown.

This is one of the motivations of this chapter to investigate into the cognitive informatics models of the brain at the system logical and functional levels. Another motivation is, according to the functional model of the brain, genes may only explain things at the level of inherited life functions, rather than at the level of acquired life functions, because the letter cannot be directly represented in genes in order to be inherited. Therefore, high-level cognitive functional models of the brain are yet to be sought to explain the fundamental mechanisms of the natural intelligence.

9.2.2 NEURAL INFORMATICS FOUNDATIONS OF THE BRAIN

The brain is the organizing and processing center of the nervous system. It receives impulses from the spinal cord and 12 pairs of cranial nerves arising in the sensory organs and other organs. It develops appropriate responses and sends forth these responses by motor neurons. The brain consumes about 25% of all the oxygen used in the body and is extremely sensitive to oxygen or glucose deprivation.

Definition 9.2 *Neural Informatics* (NeI) is a new interdisciplinary enquiry of the biological and physiological representation of information and knowledge in the brain at the neural level and their abstract mathematical models.

Neural informatics is a branch of cognitive informatics, where memory is recognized as the foundation and platform of any natural or artificial intelligence [Wang and Wang, 2006; Wang, 2007a/07g]. This subsection briefly introduces the basic unit of the brain known as the neurons and their mechanisms. Then, it describes the physiological structures of the brain and its functional lobes.

9.2.2.1 Neurons and Synapses

Nerve cells are called *neurons* or *nerve fibers*. Neurons may be classified into three groups:

- *Sensory neurons*: They carry impulses from receptors to the *Central Nervous System* (CNS, including brain and spinal cord). *Receptors* detect external or internal changes and send the information to the CNS in the form of impulses.
- *Motor neurons*: They carry impulses from the CNS to *effectors* (muscles or glands). This allows a person to respond to the messages that the brain or spinal cord has received.
- *Interneurons*: They are located entirely within the CNS and connect sensory and motor neurons.

Neurons that transmit impulses to other neurons (or effectors) do not actually touch one another. The junction between two neurons or between a neuron and its effecter muscle or gland is the *synapse* as shown in Fig. 9.1. The small gap or space between the axon of one neuron and the dendrites or cell body of the next neuron is called the *synaptic cleft*. On arriving at the *synaptic knobs* (terminal end) of the axon, the impulse stimulates the release of chemical substances called *neurotransmitters*. Neurotransmitters swiftly diffuse across the synaptic cleft and change the permeability of the next neuron. This causes depolarization and generates an electrical impulse which in turn is carried by the neuron's axon to the next synapse.

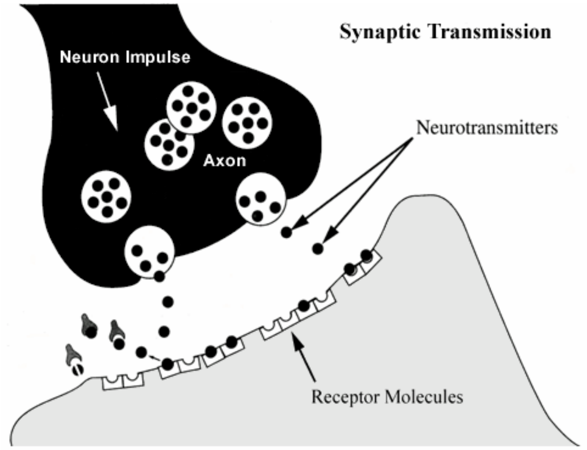


Figure 9.1 Synaptic transmission

An important function of the presence of synapses is that they ensure one-way transmission of impulses in the neural networks. A nerve impulse cannot go backward across a synapse because neurotransmitters can only be released by a neuron's axon.

There are two types of neuron effects known as the *excitatory* and *inhibitory*, respectively. As shown in Fig. 9.2, transmitter chemicals from neurons *A* and *B* are both excitatory; while that of *C* is inhibitory. Although neither *A* nor *B* is capable of causing sufficient depolarization to initiate an action potential in neuron *D*, when neurons *A* and *B* fire at the same time, a sufficient amount of transmitter chemical is released to cause depolarization of the postsynaptic membrane. The production of an action potential in neuron *D* requires the sum of two or more excitatory neurons, which is known as the *summation mechanism* of input signals.

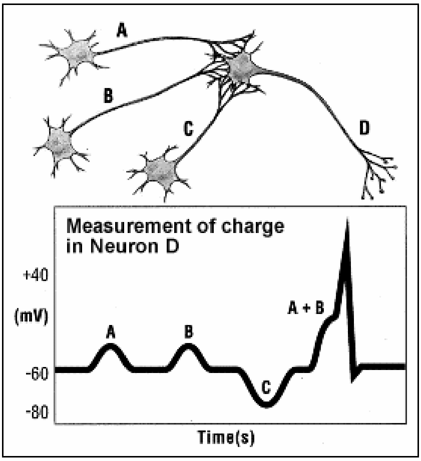
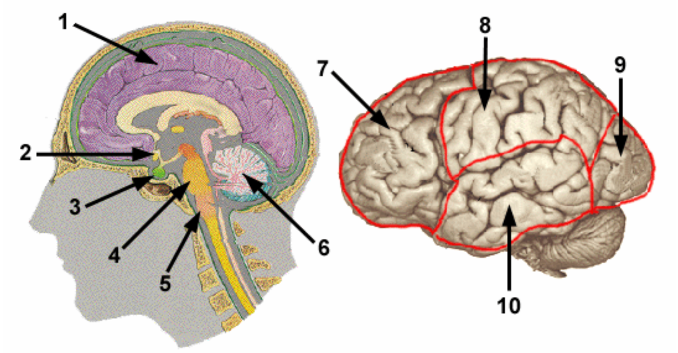


Figure 9.2 Summation of input signals in neural networks

It is noteworthy that almost all cells in the body have a lifecycle in which they reproduce themselves via divisions. This mechanism allows human trait information to be transferred to offspring through genes (DNA) replications during cell reproduction. However, it is observed that the most special mechanism of neurons is that they are the only type of cells in human body that does not go through reproduction but remains alive throughout the entire human life [Thomas, 1974; Fried and Hademenos, 1999; Kandel et al., 2000]. The advantage of this mechanism is that it enables the physiological representation and retention of acquired information and knowledge to be memorized permanently in long-term memory. But the vital disadvantage of this mechanism is that it does not allow acquired information to be physiologically passed on to the next generation, because there is no DNA replication among memory neurons. This special mechanism of neurons in the brain explains not only the foundation of memory and memorization, but also the wonder why acquired information and knowledge cannot be passed and inherited physiologically through generation to generation.

9.2.2.2 Physiological Structure of the Brain

The brain is divided into three major portions, as shown in Fig. 9.3, encompassing the *brain stem* (an extension of the spinal cord), the *cerebellum*, and the large folded *cerebrum* sitting atop the brain stem.



1. Cerebrum 2.Hypothalumus 3. Pituitary gland 4. Pons 5. Medulla oblongata
6. Cerebellum 7. Frontal lobe 8. Parietal lobe 9. Occipital lobe 10. Temporal lobe

Figure 9.3 Structure of the Brain

The major components of the brain and their functions can be described as follows:

- *Cerebrum*: The cerebrum is a storage of information accumulated from senses such as hearing, sight, etc.
- *Hypothalamus*: The hypothalamus regulates temperature and glandular secretions.
- *Cerebellum*: The cerebellum regulates motor impulses that stimulate or inhibit skeletal muscles.
- *Medulla oblongata*: The medulla oblongata regulates heartbeat and blood pressure.

The functions of the four lobes of the brain as shown in areas 7 through 10 in Fig. 9.3 can be described as follows:

- *Frontal lobe*: It controls higher layer cognitive processes such as abstract reasoning, motor processing, and aspects of personality.
- *Parietal lobe*: It controls somatosensory sensation processing such as those of skin and muscles, as well as senses of space and motion.
- *Occipital lobe*: It controls vision and visual processing.
- *Temporal lobe*: It controls auditory processing and languages.

The brain is divided into the left and right *hemispheres*. Studies show that each hemisphere of the brain controls different functions. The *left* *hemisphere* controls speech, logic, calculations, writing, and mathematics. The *right hemisphere* controls artistic conceptions and spatial perceptions discriminating shapes and forms. Information travels from one side to the other through the links between the two hemispheres known as the *corpus callosum*.

9.2.2.3 Cognitive Models of Memories

Memory is the foundation for maintaining a stable state of an animate system. It is the foundation for any form of natural and machine intelligence. Without memory intelligence cannot exist. This subsection explores types of human memory, their cognitive models, and their neurophysiological foundations.

9.2.2.3.1 The Magnitude of Human Brain

It is perceived that the elementary function and mechanism of the brain as a hierarchical neural network is quite simple, but its magnitude is extremely high [Turing, 1950; Kleene, 1956; Rabin and Scott, 1959; Widrow and Lehr, 1990; Kotulak, 1997; Leahey, 1997; Gabrieli, 1998; Matlin, 1998; Payne and Wenger, 1998; Harnish, 2002]. This is a common phenomenon of the naturally grown intelligence, which uses tremendous number of highly recurrent but simple elements to implement highly complex intelligent mechanisms and concurrent behaviors.

A comparison between the brain capacities of human and those of many other species, as shown in Fig. 9.4, can be served as an explanation of the above assumption [Smith, 1993; Kotulak, 1997; Pinel, 1997; Rosenzmeig et al., 1999].

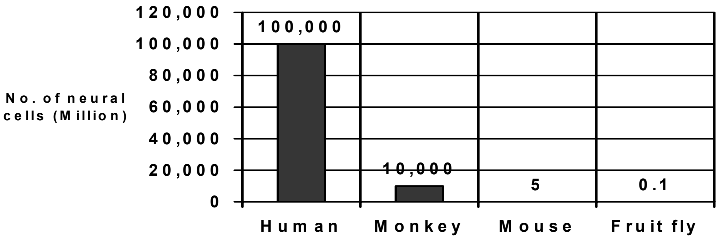


Figure 9.4 Brain capacities of human beings and other animals

Fig. 9.4 shows that the human brain contains about 100 billion (100 × 10^9) neurons. Further investigations reveal that each of them, in average, possesses thousands of synapses connecting to other neurons. Comparing the human brain and those of other animals, the magnitude of capacity of the human brain highlights a significant difference. This is one of the evidences that indicates the memory capacity is the key to distinguish humans from other species.

The trend of growth and the magnitude of human neurons in the brain are shown in Fig. 9.5 [Marieb, 1992; Smith, 1993; Pinel, 1997; Rosenzmeig et al., 1999]. Conventionally, long-term memory is perceived as static and fixed in adult brains [Marieb, 1992; Smith, 1993; Pinel, 1997; Sternberg, 1998]. This is based on the observation that the capacity of adult brains has already reached a stable state and would not grow continuously. However, latest discoveries in neuroscience and cognitive informatics indicate that long-term memory is dynamically reconfiguring, particularly at the lower levels of the neural clusters [Baddeley, 1990; Squire et al., 1993; Gabrieli, 1998; Solso, 1999; Wang and Wang, 2006]. Otherwise, the mechanisms of memory establishment, enhancement, and evolution, that are functioning everyday in the brain, cannot be explained.

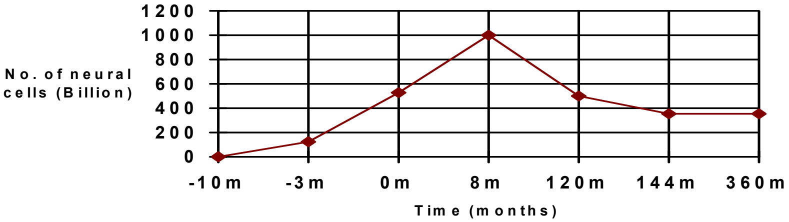


Figure 9.5 Trend of growth of the human brain

The two perceptions above are actually not contradictory. The former observes that the macro-number of neurons will not increase significantly in an adult brain. The latter perceives that information and knowledge should be physically and physiologically represented in long-term memory by newly grown synapses between the existing neurons.

9.2.2.3.2 Taxonomy of Human Memories

Types and structures of memories in the human brain have attracted a lot of interest since psychology was emerged as an independent discipline nearly 130 year ago. In 1890, William James identified that there are three components in human memory [James, 1890]:

- The after-image
- The primary memory
- The secondary memory

The after-image memory proposed above is considered a relatively narrow concept because there are other sensorial inputs to the memory, such as hearing and touch. Thus, the after-image was gradually replaced by the concept of sensory memory. Therefore, contemporary theories on memory classification [Baddeley, 1990; Smith, 1993; Squire et al., 1993; Gabrieli, 1998] can be commonly described as follows:

- The sensory memory
- The short-term memory
- The long-term memory

Examining the above types of memory it may be seen that there is a lack of an output-oriented memory, because the sensory memory is only an input-oriented buffer. A new type of memory called the *action buffer memory* is introduced recently [Wang, 2002d/2007g; Wang and Wang, 2006], which denotes the memory functions for the output-oriented actions, skills, and behaviors, such as a sequence of movement and a pre-prepared verbal sentence. Therefore, according to cognitive informatics, the logical architecture of memories in the brain can be classified into the following categories:

- The sensory buffer memory (SBM)
- The short-term memory (STM)
- The long-term memory (LTM)
- The action buffer memory (ABM)

The 27th Principle of Software Engineering

Theorem 9.3 The *Cognitive Model of Memory* (CMM) states that the architecture of human memory is parallel configured by the Sensory Buffer Memory (SBM), Short-Term Memory (STM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM), i.e.:

$$\begin{aligned} \text{CMM} \triangleq\ & \text{SBM} \\ & \parallel \text{STM} \\ & \parallel \text{LTM} \\ & \parallel \text{ABM} \end{aligned} \qquad (9.1)$$

9.2.2.3.3 Functional Models of Memories

A set of functional models of SBM, STM, LTM, and ABM in CMM is presented below for explaining the architectures, functions, and behaviors of memories in the brain.

(a) The Sensory Buffer Memory

The *sensory buffer memory* (SBM) is an input-oriented temporary memory.

Model 9.1 The functional model of SBM is a set of *queues* corresponding to each of the sensors of the brain.

The capacity of SBM is quite small. Some psychological experiments reported that the capacity of SBM is about 7±2 digits [Miller, 1956]. However, this type of memory was confusedly called the short-term memory according to Miller [Miller, 1956; Smith, 1993]. The basic mechanism of SBM is that the contents stored in it can only last for a short moment until new information arrives to the same sensory. When the new information arrives, the old one in the buffered queue should either be moved into STM or be replaced by the new one. This explains why the SBM seems to be rather small.

(b) The Short-Term Memory

The *short-term memory* (STM) is the working memory of the brain. Information lasting period in STM is about 24 hours [Wang and Wang, 2006], although some literature considered it is only a few minutes to a few hours [Baddeley, 1990; Smith, 1993; Sternberg, 1998). Out of this time span, the information will be either moved into the long-term memory or removed (forgot) or replaced from STM.

Model 9.2 The functional model of STM is a set of *stacks*.

This model explains why people can remember better the events and information gained in early morning and later evening [Smith, 1993; Sternberg, 1998]. The former is true because the stacks are relatively empty, so that sufficient working space is available. The latter can be proved based on the mechanism of stacks, in which the information buffered last is on the top of STM that gain the priority to be processed and memorized first.

(c) The Long-Term Memory

The *long-term memory* (LTM) is the permanent memory that human beings rely on for retaining acquired information in terms of facts, knowledge, and skills. LTM is apparently unlimited, because of its enormous neurons (at $100 \bullet 10^9$ level), and much more potential synapses connections (at $10^{8,432}$ level), which will be further analyzed in Section 9.4.5.

Model 9.3 The functional model of LTM is *hierarchical neural clusters* with partially connected *neurons* via *synapses*.

The structure of LTM is dynamic and partially interconnected neural networks as shown in Figs. 9.4 and 9.5, where a connection between two neurons by a synapse represents a relation.

Although the number of neurons in LTM was perceived as static and it made no change in an adult's brain [Marieb, 1992; Smith, 1993; Pinel, 1997; Sternberg, 1998] as shown in Fig. 9.5, the connections between the neurons in the form of synapses in LTM are dynamic and lively reconfiguring, particularly at the lower levels or on leaves of the neural clusters. This fits the observations in neural science and explains the mechanisms of memory establishment, evolution, and the effects of learning.

(d) The Action Buffer Memory

Supplementary to the input-oriented SBM, the *action buffer memory (ABM)* is an output-oriented temporary memory.

Model 9.4 The functional model of ABM is a set of *parallel queues*, each of them representing a sequence of actions or a process.

ABM has not been reported in the literature before and is discovered in [Wang, 2002d/2007g; Wang and Wang, 2006]. Detailed mechanism of ABM will be explained in Section 9.3.4, where the ABM will be put into the context of the dynamic memory model.

9.2.2.3.4 Neurophysiological Foundations of Memories

Because information and knowledge have to be represented physiologically in the brain at the bottom level, the functional models of memories, such as LTM, STM, SBM, and ABM, should be physically identified and mapped to physiologically organs in the brain. The major organ that accommodates memories in the brain is the cerebrum or the cerebral cortex [Baddeley, 1990; Squire et al., 1993; Smith, 1993; Gabrieli, 1998; Sternberg, 1998; Solso, 1999; Wang and Wang, 2006] as shown in

Table 9.2. Particularly, the association and premotor cortex in the frontal lobe, the temporal lobe, sensory cortex in the frontal lobe, visual cortex in the occipital lobe, primary motor cortex in the frontal lobe, supplementary motor area in the frontal lobe, and procedural memory in the cerebellum.

Table 9.2
Neural Physiological Foundations of Memories

Memory	**Corresponding part in the cerebrum**
LTM	• Association cortex in the frontal lobe • Premotor cortex in the frontal lobe
STM	• The temporal lobe
SBM	• Sensory cortex in the frontal lobe • Visual cortex in the occipital lobe
ABM	• Primary motor cortex in the frontal lobe • Supplementary motor area in the frontal lobe • Procedural memory in cerebellum

In Table 9.2, the relations between memories and their corresponding parts in the cerebral cortex and lobes are established. LTM as the largest and dynamic memory of the brain is mainly located at the association cortex in the frontal lobe of the cerebrum.

The CMM model and the mapping of the four types of human memory onto the physiological organs in the brain reveal a set of fundamental mechanisms of neural informatics. The theories of cognitive informatics and neural informatics explain a number of important questions in the study of natural intelligence. Enlightening results derived in cognitive informatics and neural informatics are listed below, which will be explained throughout this chapter:

a) LTM establishment is a subconscious process during sleeping (See Theorem 9.10);

b) The general acquisition cycle of LTM is equal to or longer than 24 hours (See Model 9.8);

c) The mechanism of LTM establishment is to update the entire memory of information represented as an OAR model in the brain (See Theorem 9.11);

d) Eye movement and dreams play an important role in LTM creation.

The latest development in cognitive informatics and neural informatics has led to the determination of the magnitude and expected capacity of human memory, which will be presented in Section 9.4.5.

9.2.3 THE EMERGENCE OF COGNITIVE INFORMATICS

It is recognized that the brain and natural intelligence are centric on information processing [Wang, 2002d/2003b/2007a; Wang et al., 2006]. The information metaphor is widely adopted in cognitive science, following the dominations of *structuralism, functionalism, associationism, connectionism,* and *behaviorism* [Leahey, 1997].

N. Stillings and M.H. Feinstein assumed that "the human mind is a complex system that receives, stores, retrieves, transforms, and transmits information [Stillings and Feinstein, 1987]." R. Harre perceived that cognitive science is the study of cognitive phenomena [Harre, 2002], while Michael Dawson considered "the central assumption for cognitive science is information processing [Dawson, 1998]." In summary, contemporary cognitive science is the study of the brain, the mind, and intelligent behavior that blends anthropology, computer science, psychology, neuroscience, linguistics, sociology, and philosophy.

Cognitive informatics is a cutting-edge and interdisciplinary research area that tackles the common root problems of modern informatics, intelligent science, computing, software engineering, AI, cognitive science, knowledge science, and neuropsychology. This subsection explores the emerging discipline of *cognitive informatics*. Cognitive informatics studies the internal information processing mechanisms and natural intelligence of the brain. The historical development of informatics from the classical information theory, contemporary informatics, to cognitive informatics, has been reviewed in Chapter 7.

Definition 9.3 *Cognitive Informatics* (CI) is a transdisciplinary enquiry of natural and machine intelligence, and their products in terms of information, knowledge, and behaviors.

Cognitive informatics is a new frontier that attempts to solve problems in two interconnected areas in a bi-directional and multidisciplinary approach. In one direction, cognitive informatics uses cognitive science theories to investigate informatics, computing, and software engineering problems, such as information and knowledge representation in the brain, the nature of computing, cognitive complexity of software, abstraction of software, and system behaviors. In the other direction, cognitive informatics uses computing theories and formal mathematical means to investigate cognitive science problems, such as memory, learning, and thinking.

Cognitive informatics is a discipline that forges links between a number of natural science and life science disciplines with informatics and computing science. The relationship between cognitive informatics and other natural sciences can be perceived as shown in Fig. 9.6.

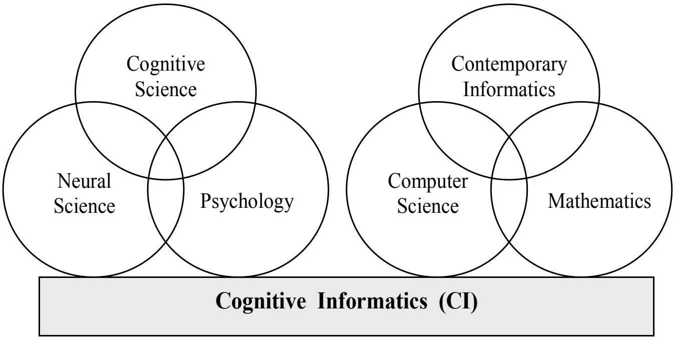


Figure 9.6 Relationship between cognitive informatics and related science disciplines

Fig. 9.6 shows that the foundations of cognitive informatics are based on multidisciplinary knowledge, such as those of informatics, natural sciences, and humanity. These foundations can be classified into three categories as shown in Table 9.3.

Table 9.3
Foundations of Cognitive informatics

Category No.	Category of sciences	Discipline of sciences
1	Informatics	
1.1		Modern information theory
1.2		Computing theory
1.3		Software science
1.4		Artificial intelligence
2	Natural sciences	
2.1		Cognitive science
2.2		Neurobiology
2.3		Psychology
2.4		Physiology
2.5		Mathematics
3	Humanity	
3.1		Philosophy
3.2		Linguistics

9.2.4 THE THEORETICAL FRAMEWORK OF COGNITIVE INFORMATICS

Cognitive informatics is a discipline that forges links between a number of natural science and life science disciplines with informatics and computing science. The structure of the theoretical framework of cognitive informatics is described in Fig. 9.7, which encompasses the fundamental theories of cognitive informatics, denotational mathematics for cognitive informatics, and the key application areas of cognitive informatics.

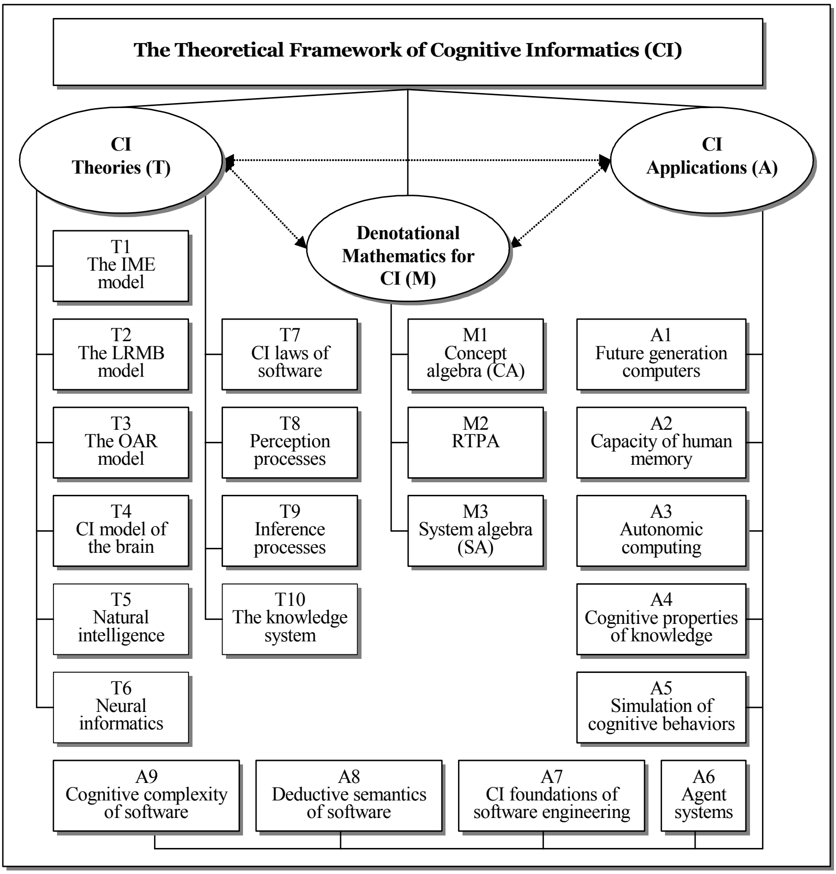


Figure 9.7 The theoretical framework of cognitive informatics

9.2.4.1 The Fundamental Theories of Cognitive informatics

The fundamental theories of cognitive informatics are developed in ten aspects resulting in the basic and transdisciplinary research in cognitive informatics [Wang, 2002d/07a], which encompass the Information-Matter-Energy (IME) model, the Layered Reference Model of the Brain (LRMB), the Object-Attribute-Relation (OAR) model of information representation in the brain, the cognitive informatics model of the brain, Natural Intelligence (NI), Neural Intelligence (NeI), the cognitive informatics laws of software, the mechanism of human perception processes, the cognitive processes of formal inferences, and the formal knowledge system. The remainder of this chapter and related parts throughout the book explains these fundamental theories of cognitive informatics and their interrelationships.

The denotational mathematics is an important part of the theories of cognitive informatics as introduced in Section 4.5, which provides rigorous and expressive means for formal reasoning in cognitive informatics studies. Three new types of denotational mathematics, *concept algebra* [Wang, 2006e], system algebra [Wang, 2006d], and RTPA [Wang, 2002a], are created for cognitive informatics to enable rigorous treatment of knowledge representation and manipulation in a formal and coherent framework.

The new structures of contemporary mathematics have extended the abstract objects under study in mathematics to a higher level on concepts, behavioral processes, and systems. RTPA has been intensively discussed in related sections of Chapters 4 and 6. System algebra will be presented in Chapter 10. Concept algebra may be referred to Chapter 15 and [Wang, 2004e]. A wide range of applications of the denotational mathematics in the context of cognitive informatics has been identified [Wang, 2002b/03c/06j].

9.2.4.2 The Domain of Cognitive Informatics

The key application areas of cognitive informatics can be divided into two categories [Wang, 2007a]. One category of applications, A2, A4, and A5 as shown in Fig. 9.7, uses informatics and computing techniques to investigate cognitive science problems, such as memory, learning, and reasoning. The other category including the remainder areas uses cognitive theories to investigate problems in informatics, computing, and software/knowledge engineering. Cognitive informatics focuses on the nature of information processing in the brain, such as information acquisition, representation, memory, retrieve, generation, and communication. Through the interdisciplinary approach and with the support of modern information and neuroscience technologies, mechanisms of the brain and the mind may be systematically explored within the framework of cognitive informatics.

Cognitive informatics covers a whole range of interdisciplinary research in subject areas including NI, autonomic computing, and NeI, as shown in Table 9.4.

Table 9.4
Subject Areas of Cognitive informatics

No.	Category	Description
1	**Natural Intelligence (NI)**	
1.1		Informatics models of the brain
1.2		Cognitive processes of the brain
1.3		Internal information processing mechanisms
1.4		Theories of natural intelligence
1.5		Intelligent foundations of computing
1.6		Descriptive mathematics for NI
1.7		Abstraction and means
1.8		Ergonomics
1.9		Informatics laws of software
1.10		Knowledge representation
1.11		Models of knowledge and skills
1.12		Language acquisition
1.13		Cognitive complexity of software
1.14		Distributed intelligence
1.15		Computational intelligence
1.16		Emotions/motivations/attitudes
1.17		Perception and consciousness
1.18		Hybrid (AI/NI) intelligence
2	**Autonomic Computing (AC)**	
2.1		Imperative vs. autonomic computing
2.2		Reasoning and inferences
2.3		Cognitive informatics foundations of AC
2.4		Memory models
2.5		Informatics foundations of software engineering
2.6		Fuzzy logic
2.7		Knowledge engineering
2.8		Pattern recognition
2.9		Agent technologies
2.10		Artificial intelligence
2.11		Software agent systems

2.12		Decision theories
2.13		Problem solving
2.14		Machine learning
2.15		Intelligent Internet techniques
2.16		Web contents cognition
2.17		Nature of software
2.18		Quantum computing
3	**Neural informatics (NeI)**	
3.1		Neural informatics foundations of information processing
3.2		Cognitive models of the brain
3.3		Functional modes of the brain
3.4		Neural models of memory
3.5		Neural networks
3.6		Neural computation
3.7		Cognitive linguistics
3.8		Neuropsychology
3.9		Bioinformatics
3.10		Biosignal processing
3.11		Cognitive signal processing
3.12		Gene analysis
3.13		Gene expression
3.14		Neural signal interpretation
3.15		Visual information representation
3.16		Visual information interpretation
3.17		Sensational cognitive processes
3.18		Human factors in systems

9.3 Cognitive Informatics Models of the Brain

The human brain is the most complicated organ in the universe and is constantly the frontier yet to be explored in an interdisciplinary approach. Investigation into the brain and its cognitive mechanism is a unique and the hardest problem in science that requires recursive and introspective mental power to explore the brain by the brain.

This section develops the cognitive informatics models of the brain. One of the focuses of this section is the relationship between the inherited life functions and the acquired life functions. Another focus is the memory-based theory of intelligence, which deems that the memory mechanisms are the foundation for any kind of natural intelligence. Without the establishment of a unified memory model, studies on the brain will never form a coherent theory.

9.3.1 THE LAYERED REFERENCE MODEL OF THE BRAIN (LRMB)

A variety of life functions and their cognitive processes have been identified in cognitive informatics, cognitive neuropsychology, cognitive science, and neurophilosophy. Based on the advances of research in cognitive informatics and related fields, this subsection presents a Layered Reference Model of the Brain (LRMB) [Wang et al, 2006] that explains the functional mechanisms and cognitive processes of the natural intelligence. In order to formally and rigorously describe a comprehensive and coherent set of mental processes and their relationships, the hierarchical LRMB model is established, which encompasses 39 cognitive processes at six layers known as the *sensation, memory, perception, action, meta cognitive,* and *higher cognitive layers* from the bottom up.

The following subsections discuss each of these six layers of cognitive processes and interactions between the layers in the context of the cognitive model of the brain as a real-time intelligent system.

9.3.1.1 The Architecture of LRMB

LRMB can be described as shown in Fig. 9.8. At the top level, the hierarchical life functions of the brain, the natural intelligent system (NI-Sys), can be divided into two categories: the *subconscious* and *conscious s*ubsystems. The former known as the NI *operating system* (NI-OS) encompasses the layers of sensation, memory, perception, and action (Layers 1 to 4). The latter known as the NI *applications* (NI-App) includes the layers of meta and higher cognitive functions (Layers 5 and 6).

The *subconscious layers* of the brain represented by *NI-OS* are inherited, fixed, and relatively mature when a person was born. Therefore, the subconscious cognitive function layers are not directly controlled and accessed by the conscious life function layers. This is why it used to be called the nonconscious life functions in psychology literature [Smith, 1993; Leahey, 1997; Payne and Wenger, 1998; Sternberg, 1998; Reisberg, 2001].

The *conscious layers* of the brain, represented by *NI-App*, are acquired, highly plastic, programmable, and can be controlled intentionally based on willingness (motivations), goals, and inferences. Although cognitive informatics puts more emphases on exploring the conscious *NI-App*, the interactions of *NI-App* with *NI-OS* profoundly shape the theory of LRMB.

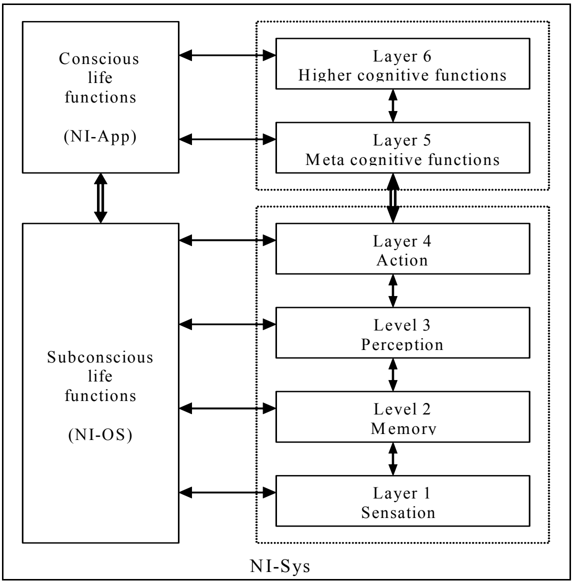


Figure 9.8 The Layered Reference Model of the Brain (LRMB)

A formal description of the high-level architecture of the LRMB model using RTPA is presented in Fig. 9.9. Detailed descriptions of individual layers of LRMB will be presented in the following subsection and may be referred to [Wang et al., 2006].

The Layered Reference Model of the Brain (LRMB)

```
§LRMBST ≜ §NI_Sys            // The Natural Intelligent system
        =   NI_OS            // The sub conscious NI operating system
          || NI_App          // The conscious NI applications
        = (  Layer1_Sensation
           || Layer2_Memory
           || Layer3_Perception
           || Layer4_Action
          )
        || (  Layer5_Meta_Cognitive_Functions
           || Layer6_Higher_Cogntive_Functions
          )
```

Figure 9.9 Formal description of the LRMB model of the brain

9.3.1.2 The Functional Layers of LRMB

Based on the architectural framework of LRMB, the six layers of cognitive processes of the brain and their relationships are descried below.

9.3.1.2.1 Layer 1: The Sensation Layer

Definition 9.4 The *sensation layer* of LRMB is a subconscious layer of life functions of the brain for detecting and acquiring cognitive information from the external world via physical and/or chemical means.

The sensation layer encompasses all input-oriented senses such as vision, audition, smell, tactility, and taste. The sensation layer is associated with a set of input-oriented temporary memory, known as SBM, as modeled in Section 9.2.2.3.

Definition 9.5 *Sensation* is a set of cognitive processes of the brain at the subconscious life functional layer that forms the interfaces between the internal and external worlds for information detection, transformation, and acquisition.

Sensations are mental states caused by the stimulation of sensory organs affected by either real-world entities or energy as well as their changes of statuses in the external world. The sensational cognitive processes at Layer 1 of LRMB encompass the basic cognitive life functions of *vision, audition, smell, tactility,* and *taste* as described in Table 9.5 [Wang et al, 2006; Wang, 2005c]. Tactility can be further divided into senses of *therme, pressure, weight, pain,* and *texture*; while taste can be categorized as that of *salt, sweet, bitter, sour,* and *pungency*.

9.3.1.2.2 Layer 2: The Memory Layer

Definition 9.6 The *memory layer* of LRMB is the fundamental layer of life functions of the brain functioning to: a) Retain and store information about both the external and internal worlds; b) Maintain a stable state of an animate system; c) Provide a working space of abstract inference; and d) Buffer programmed actions and motions to be executed by the body.

It is recognized that the natural intelligence is memory-based [Wang, 2002d/07a; Wang and Wang, 2006], and the memory layer is a fundamentally important part of the subconscious life functions. According to neural informatics as discussed in Section 9.2.2, the memory of brain encompasses the following types of memories: SBM, STM, LTM, and ABM, where LTM is the key to understand the mechanism of the natural intelligence.

Definition 9.7 *Memory* is a set of cognitive processes of the brain at the subconscious life function layer that retains the external or internal cognitive information in various memories of the brain, particularly in LTM.

The memory layer is the fundamental layer of life functions of the brain for maintaining a stable state of an animate system. The major means of interaction between the conscious and subconscious layers of LRMB are all forms of memories and the information retained in them.

9.3.1.2.3 Layer 3: The Perception Layer

Definition 9.8 The *perception layer* of LRMB is a subconscious layer of life functions of the brain for maintaining conscious life functions and for browsing internal abstract memories in the cognitive models of the brain.

The cognitive functions of the perception layer can be considered as the *thinking engine* of the brain and the kernel of the natural intelligence. Perception may also be considered as the *sixth sense*, supplementary to the five external sensations at Layer 1, known as vision, audition, smell, tactility, and taste, which implements self consciousness inside the abstract memories of the brain. Therefore, the perception layer is a core part of the subconscious life functions.

Definition 9.9 *Perception* is a set of internal sensational cognitive processes of the brain at the subconscious life function layer that detects, relates, interprets, and searches internal cognitive information in the mind.

The cognitive processes of perception encompass the self-consciousness, attention, emotions, attitude, sense of spatiality, and sense of motion as shown in Table 9.5. Perception is the internal sensory layer of the brain that almost all cognitive life functions rely on it. Perception is also an important cognitive function at the subconscious layers that determines personality. In other words, personality is a faculty of all subconscious life functions and experience cumulated via conscious life functions.

9.3.1.2.4 Layer 4: The Action Layer

Definition 9.10 The *action layer* of LRMB is a subconscious layer of life functions of the brain for output-oriented actions and motions that implement human behaviors such as a sequence of movement and a pre-prepared verbal communication.

The action layer is a part of the subconscious life functions. The action layer encompasses all motor control and execution functions such as looking, reading, and writing as shown in Table 9.5. Supplemented to the input-oriented SBM, ABM is an output-oriented temporary memory. The functional model of ABM is a set of parallel queues, each of them representing a sequence of actions, or a process. ABM was first identified in [Wang, 2002d; Wang and Wang 2006]. The action and the sensation layers form a closed-loop for implementing various life functions, particularly the cognitive life functions at the conscious layers.

Definition 9.11 *Actions* are a set of subconscious cognitive processes of the brain at the subconscious life function that executes both bodily (external) or mental (internal) actions via the motor systems of the body or the perceptional engine of the brain.

Note that there are mental perceptual actions inside the brain via the thinking engine of the mind. This observation is significant to explain how perceptive and thinking processes are carried out in the abstract or information world of the brain.

9.3.1.2.5 Layer 5: The Meta Cognitive Process Layer

Definition 9.12 The *meta cognitive process layer* of LRMB is a conscious layer of life functions of the brain that carries out the fundamental and elementary cognitive processes commonly used in higher cognitive processes.

The meta cognitive process layer is a part of the conscious life functions that can be controlled directly by the conscious mind (or the thinking engine) as mental applications.

Definition 9.13 A *meta cognitive function* is a fundamental and elemental cognitive process of the brain at the conscious life function layer that is commonly used (or applied) to support the higher layer cognitive life functions.

The meta cognitive functions at Layer 5 of LRMB encompass the basic cognitive processes of identify object, abstraction, concept establishment, search, categorization, memorization, selection, qualification, quantification, and comparison as shown in Table 9.5.

9.3.1.2.6 Layer 6: The Higher Cognitive Process Layer

Definition 9.14 The *higher cognitive process layer* of LRMB is a conscious layer of life functions of the brain that carries out a set of specific cognitive processes under the support of the meta cognitive processes.

The higher cognitive process layer is a part of the conscious life functions. More complicated and colorful life functions can be implemented by serial, parallel, and interleaved combinations of these cognitive processes in LRMB.

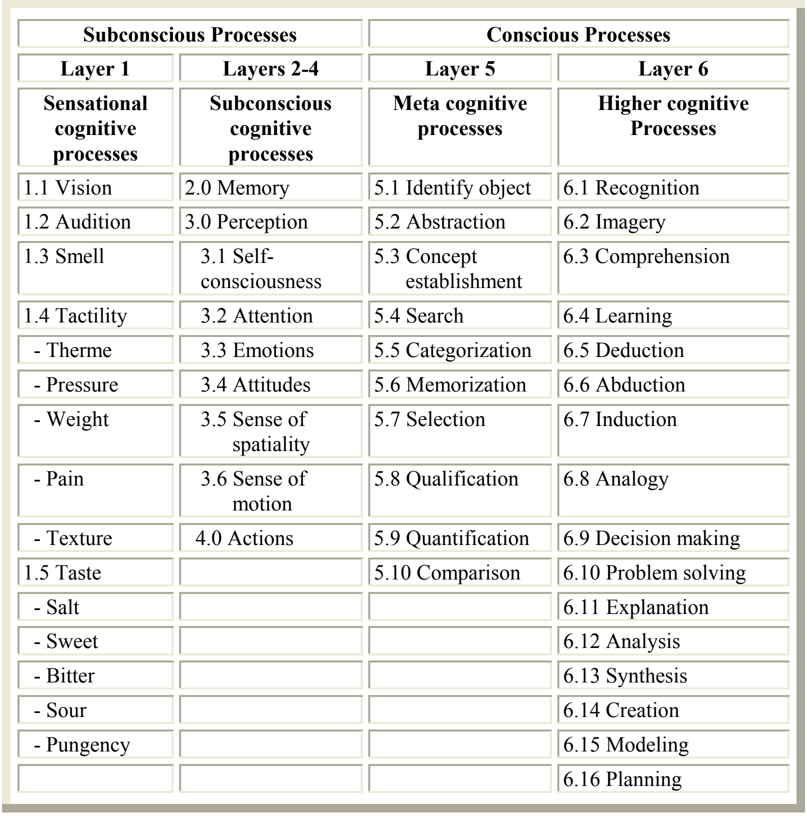
Definition 9.15 A *higher cognitive function* is an advanced cognitive process of the brain at the conscious life function layer that is developed and acquired to carry out commonly recurring life functions under the support of the meta cognitive process.

The higher cognitive functions at Layer 6 of LRMB include 16 processes such as recognition, imagery, comprehension, learning, deduction, induction, abduction, analogy, decision making, problem solving, explanation, analysis, synthesis, creation, planning, and modeling as shown in Table 9.5.

9.3.1.3 The Configuration of the Cognitive Processes of LRMB

In a summary, LRMB models 39 cognitive processes as shown in Table 9.5, which are categorized into the six layers and two subsystems. It is a great curiosity to explore the insides and processes of the brain and to explain its fundamental mechanisms by a set of cognitive processes. Formal descriptions of particular cognitive processes of LRMB in RTPA may be referred to [Wang, 2007h/07i; Wang and Gafurov, 2003; Chiew and Wang, 2004; Wang and Ruhe, 2007]. A comprehensive collection of detailed description of all the LRMB processes will be presented in *Cognitive Informatics: A Transdisciplinary Field Exploring Natural and Artificial Intelligence* [Wang, 2007j].

Table 9.5
Classification of Cognitive Processes in LRMB



Subconscious Processes		Conscious Processes	
Layer 1	**Layers 2-4**	**Layer 5**	**Layer 6**
Sensational cognitive processes	**Subconscious cognitive processes**	**Meta cognitive processes**	**Higher cognitive Processes**
1.1 Vision	2.0 Memory	5.1 Identify object	6.1 Recognition
1.2 Audition	3.0 Perception	5.2 Abstraction	6.2 Imagery
1.3 Smell	3.1 Self-consciousness	5.3 Concept establishment	6.3 Comprehension
1.4 Tactility	3.2 Attention	5.4 Search	6.4 Learning
- Therme	3.3 Emotions	5.5 Categorization	6.5 Deduction
- Pressure	3.4 Attitudes	5.6 Memorization	6.6 Abduction
- Weight	3.5 Sense of spatiality	5.7 Selection	6.7 Induction
- Pain	3.6 Sense of motion	5.8 Qualification	6.8 Analogy
- Texture	4.0 Actions	5.9 Quantification	6.9 Decision making
1.5 Taste		5.10 Comparison	6.10 Problem solving
- Salt			6.11 Explanation
- Sweet			6.12 Analysis
- Bitter			6.13 Synthesis
- Sour			6.14 Creation
- Pungency			6.15 Modeling
			6.16 Planning

9.3.2 COGNITIVE PROPERTIES OF INTERNAL INFORMATION

Almost all modern disciplines of science and engineering deal with information and knowledge. However, the three fundamental concepts of data, information, and knowledge are conventionally perceived quite differently in literature [Debenham, 1989; McDermid, 1991]. *Data* are directly acquired raw information, usually a quantitative abstraction of external objects and/or their relations. *Information* is meaningful data, or the subjective interpretation of data. Then, *knowledge* is the consumed information related to existing knowledge in the brain.

Based on the investigations in cognitive informatics, particularly the research on the object-attribute-relation model and the mechanisms of internal information representation, the above empirical classification of the cognitive levels of data, information, and knowledge may be revised. A cognitive informatics perception on the relationship among data (sensational inputs), actions (behavioral outputs), and their internal representations such as knowledge, experience, and skill, are that all of them are cognitive information. The taxonomy of cognitive information is determined by the types of inputs and outputs of information to and from the brain as described below.

Definition 9.16 The *Cognitive Information Model* (CIM) classifies cognitive information into four categories known as *knowledge, behavior, experience,* and *skill*, according to the types of input and output as either information (I) or action (A), i.e.:

a) Knowledge $K: I \rightarrow I$ (9.2)
b) Behavior $B: I \rightarrow A$ (9.3)
c) Experience $E: A \rightarrow I$ (9.4)
d) Skill $S: A \rightarrow A$ (9.5)

The CIM model provided in Definition 9.16 can be illustrated as shown in Table 9.6. According to Table 9.6, for a given cognitive process, if both I/O are abstract information, the internal information acquired is *knowledge*; if both I/O are empirical actions, the type of internal information is *skill*; and the remainder combinations between action/information and information/action produce *experience* and *behaviors,* respectively. Note in Table 9.6 that behaviors are a new type of cognitive information modeled inside the brain, which embodies an abstract input to an observable behavioral output [Wang et al., 2004; Wang, 2007a].

Table 9.6
The Cognitive Information Model (CIM)

		Type of output		**Ways of acquisition**
		Information	**Action**	
Type of input	**Information**	Knowledge (K)	Behavior (B)	*Direct or indirect*
	Action	Experience (E)	Skill (S)	*Direct only*

It is noteworthy that the approaches to acquire knowledge/instructions and experience/skills are fundamentally different. Although knowledge or behaviors may be acquired directly or indirectly, skills and experiences can only be obtained directly by hands-on activities. Further, the storage locations of the abstract information are different, where knowledge and experience are stored as abstract relations in LTM, while behaviors and skills are stored as wired neural connections in ABM.

The cognitive informatics theory developed in this section explains why people have to make the same mistakes in order to gain empirical experiences and skills, and why experience transfer is so hard and could not be gained by indirect reading. The cognitive informatics theories on classification of internal information will be used to explain a wide range of phenomena of learning and practices in software engineering in Section 9.5.

According to Table 9.6, the following law on information manipulation and learning for both human and machine systems can be derived.

The 28th Law of Software Engineering

Theorem 9.4 The *generic forms of cognitive information* state that there are four categories of internal information $\mathcal{I}$ in the brain known as *knowledge* ($\mathcal{I}_k$), *behaviors* ($\mathcal{I}_b$), *experience* ($\mathcal{I}_e$), and *skills* ($\mathcal{I}_s$), i.e.:

$$\mathcal{I} = (\mathcal{I}_k, \mathcal{I}_b, \mathcal{I}_e, \mathcal{I}_s) \tag{9.6}$$

Theorem 9.4 lays an important foundation for learning theories and pedagogy [Wang et al., 2004; Wang, 2007a]. Theorem 9.4 indicates that learning theories and their implementation in autonomic and intelligent systems should study all four categories of cognitive information acquisitions, particularly behaviors, experience, and skills rather than only focusing on knowledge.

Based on Theorem 9.4, the following corollaries on cognitive information acquisition can be derived.

Corollary 9.1 All the four categories of information can be acquired directly by an individual.

Corollary 9.2 Knowledge and behaviors can be learnt indirectly by inputting abstract information; while experience and skills must be learnt directly by hands-on or empirical actions.

To a certain extent, software engineering deals with instructive behaviors and their relations with knowledge, experience, and skills according to the CIM model.

9.3.3 NATURAL INTELLIGENCE VS. ARTIFICIAL INTELLIGENCE

Cognitive informatics adopts a compatible perspective on natural intelligence and artificial intelligence. It is logical to believe that natural intelligence should be fully understood before artificial intelligence can be scientifically studied. This subsection explores the nature of intelligence and the equivalence between natural and artificial intelligence. In this view, conventional machines are invented to extend human physical capability, while modern information processing machines such as computers, communication networks, and robots are developed for extending human intelligence, memory, and the capacity for information processing [Wang, 2006b/07a]. Therefore, any machine that may implement a part of human behaviors and actions in information processing has possessed some extent of intelligence.

9.3.3.1 The Nature of Intelligence

Intelligence is a driving force or an ability to acquire and use knowledge and skills, or to reason in problem solving. It was conventionally deemed that only human beings and advanced species possess intelligence. However, the development of computers, robots, and autonomic systems indicates that intelligence may also be created or implemented by machines and man-made systems.

Definition 9.17 *Intelligence*, in the narrow sense, is a human or a system ability that transforms information into behaviors; and in a broad sense, it is any human or system ability that autonomously transfers the forms of abstract information among *data, information, knowledge,* and *behaviors* in the brain.

In the above definition, the four abstract objects can be defined as follows based on Definition 9.16.

Definition 9.18 The abstract objects in the brain such as data (*D*), information (*I*), knowledge (*K*), and behavior (*B*) can be formally modeled as follows:

$$D \triangleq r_d : M \to S_k = \log_k M,\ k_{\min} = 2 \tag{9.7}$$

$$I \triangleq r_i : D \to C,\ r_i \in \Re \tag{9.8}$$

$$K \triangleq r_k : C_{n+1} \rightarrow (\mathop{\Large\times}\limits_{\mathrm{i=1}}^{\mathrm{n}} C_i),\ r_k \in \mathfrak{R} \tag{9.9}$$

$$\begin{aligned} B &\triangleq \wp \\ &= \mathop{\Large R}\limits_{k=1}^{m} (@e_k \hookrightarrow P_k) \\ &= \mathop{\Large R}\limits_{k=1}^{m} [@e_k \hookrightarrow \mathop{\Large R}\limits_{i=1}^{n-1} (p_i(k)\ r_{ij}(k)\ p_j(k))],\ j = i+1, r_{ij} \in \mathfrak{R} \end{aligned} \tag{9.10}$$

where C is a *concept* as given in Definition 15.3, $\mathfrak{R}$ is the set of process relations as defined in Theorem 4.7, and the behavior B is equivalent to a program $\wp$ or a set of interacting processes as given in Definition 5.53.

With the clarification of the intension and extension of the generic concept, intelligence, the terms of natural and artificial intelligence can be derived below.

Definition 9.19 *Natural intelligence* (NI) is a system of intelligent behaviors possessed or embodied by the brains of human beings and other advanced species.

Definition 9.20 *Artificial intelligence* (AI) is a system of intelligent behaviors possessed or implemented by machines or man-made systems.

9.3.3.2 Taxonomy of Intelligence

Intelligence can be formally modeled as a set of functions that transfers a pair of abstract objects in the brain or systems as given in Definitions 9.17 or 9.18.

The 29th Law of Software Engineering

Theorem 9.5 The *nature of intelligence* states that *intelligence* $\Im$ can be classified into four forms called the *perceptive intelligence* $\Im_{\mathrm{p}}$, *cognitive intelligence* $\Im_{\mathrm{c}}$, *instructive intelligence* $\Im_{\mathrm{i}}$, and *reflective intelligence* $\Im_{\mathrm{r}}$ as modeled below:

$$\begin{aligned} \Im \triangleq\ & \Im_{\mathrm{p}} : D \rightarrow I \ \ \text{(Perceptive)} \\ & \| \Im_{\mathrm{c}} : I \rightarrow K \ \ \text{(Cognitive)} \\ & \| \Im_{\mathrm{i}} : I \rightarrow B \ \ \text{(Instructive)} \\ & \| \Im_{\mathrm{r}} : D \rightarrow B \ \ \text{(Reflective)} \end{aligned} \tag{9.11}$$

According to Theorem 9.5 and Definitions 9.17, the narrow sense of intelligence is corresponding to the instructive and reflective intelligence; while the broad sense of intelligence includes all four forms of intelligence, particularly the perceptive and cognitive intelligence.

It is recognized [Wang, 2006b/2007a] that the basic approaches to implement intelligence can be classified as shown in Table 9.7. Observing Table 9.7, software for computation is the third approach to simulate and implement the natural intelligence by programmed logic. This indicates that the nature of software is the simulation and execution of partial human behaviors, and the extension of human capability, reachability, persistency, memory, and information processing speed.

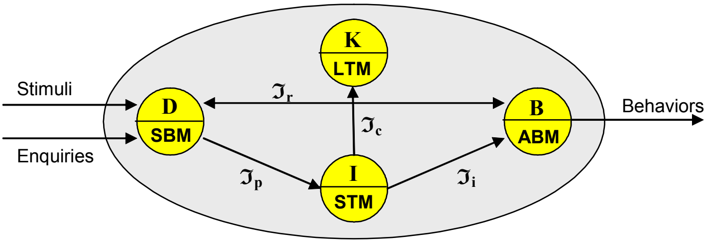
Table 9.7
Approaches to Implement Natural Intelligence and Artificial Intelligence

No.	Means	Approach	Category
1	Biological organisms	Naturally grown	NI
2	Silicon automata	Wired	AI
3	Computing systems	Programmed	AI
4	Other (in future)	Hybrid	NI + AI

9.3.3.3 The Model of Natural Intelligence

On the basis of the conceptual models developed in previous subsections, the mechanisms of natural intelligence can be described by a generic intelligence model (GIM) as given below.

Definition 9.21 The *Generic Intelligence Model* (GIM) describes the mechanisms of the natural intelligence, as shown in Fig. 9.10, according to Theorem 9.5 on the nature of intelligence.



$\mathfrak{I}_p$ – *Perceptive* intelligence $\mathfrak{I}_i$ – *Instructive* intelligence
$\mathfrak{I}_c$ – *Cognitive* intelligence $\mathfrak{I}_i$ – *Reflective* intelligence

Figure 9.10 The Generic Intelligence Model (GIM)

In the GIM model as shown in Fig. 9.10, different kind of intelligence is described as a driving force that transfers between a pair of abstract objects in the brain such as data (*D*), information (*I*), knowledge (*K*), and behavior (*B*).

It is noteworthy that each abstract object is physically stored in a particular type of memories as defined in Section 9.2.2.3 according to the CMM model given in Theorem 9.3. This is the neural informatics foundation of natural intelligence, and the physiological evidences of why natural intelligence can be classified into four forms as given in Theorem 9.5.

According to the GIM model, as well as Theorems 3.3 and 3.4, the natural and machine (artificial) intelligence share the same cognitive informatics foundation. In other words, they are compatible. Therefore, the studies on natural intelligence and artificial intelligence may be unified into a common framework.

9.3.3.4 Measurement of Intelligence

The measurement of intelligent capability of humans and systems can be classified into three categories known as intelligent quotation, intelligent equivalence, and intelligent capability as described in the following subsections.

9.3.3.4.1 Intelligent Quotient

The first measurement for mental intelligence is proposed in psychology known as the intelligent quotient based on the *Stanford-Binet intelligence test* [Binet, 1905; Terman and Merrill, 1961; Pinneau, 1961; Mackintosh, 1998]. Intelligent quotient is determined by six subtests where the passing of each subtest is counted for two equivalent months of mental intelligence.

Definition 9.22 The *mental age* A_m in an intelligent quotient test is the sum of a base age A_b and an extra equivalent age ΔA, i.e.:

$$\begin{aligned} A_m &= A_b + \Delta A \\ &= A_{\max} + \frac{2n_{sub}}{12} \\ &= A_{\max} + \frac{n_{sub}}{6} \quad [yr] \end{aligned} \tag{9.12}$$

where A_b is the maximum age A_{max} gained by a testee who passes all six subtests required for an certain age, and ΔA is determined by the number of passed subtests beyond A_{max} as merits, i.e., n_{sub}.

Definition 9.23 *Intelligent quotient* (IQ) is a ratio between the mental age A_m and the chronological (actual) age A_c, multiplied by 100, i.e.:

$$\begin{aligned} IQ &= \frac{A_m}{A_c} \bullet 100 \\ &= \frac{A_{\max} + \frac{1}{6} n_{sub}}{A_c} \bullet 100 \end{aligned} \tag{9.13}$$

Example 9.1 For a 9 year-old boy and 6.75 year-old (6 year and 9 month) girl, if they both pass all six subtests at the 7-year level, plus 12 subtests partially passes in higher age levels, what are their IQs, respectively?

As given above, for the boy: A_{c1} = 9, A_{max1} = 7, and n_{sub1} = 12; and for the girl: A_{c2} = 6.75, A_{max2} = 7, and n_{sub2} = 12. Their IQs, IQ_1 and IQ_2, can be determined according to Eq. 9.12 as follows:

$$\begin{aligned} IQ_1 &= \frac{A_{\max} + \frac{1}{6} n_{sub}}{A_c} \bullet 100 \\ &= \frac{7 + \frac{12}{6}}{9} \bullet 100 \\ &= 100.0 \end{aligned}$$

$$\begin{aligned} IQ_2 &= \frac{A_{\max} + \frac{1}{6} n_{sub}}{A_c} \bullet 100 \\ &= \frac{7 + \frac{12}{6}}{6.75} \bullet 100 \\ &= 133.3 \end{aligned}$$

According to Definition 9.12, an IQ score above 100 supposes a gifted intelligence. However, the measure is only sensitive to children but not sensitive to adults, because the differences between the mental and chronological ages for adults are not clear naturally. Another drawback in the IQ test is that the norms or benchmarks of the mental ages for determining *IQ* are not easy to objectively define, especially for adults, and were considered highly subjective. Third, the IQ test does not cover all forms of intelligence as defined in Theorem 9.5, particularly the instructive and reflective intelligent capabilities.

9.3.3.4.2 The Turing Test

The second measurement for comparative intelligence is proposed by Alan Turing based on the Turing test [Turing, 1950].

Definition 9.24 The *Turing intelligence equivalence* E_T is a ratio of conformance or equivalence evaluated in a comparative test between a system under test and an equivalent human-based system, where both systems are treated as a black box and the testers do not know which is the tested system, i.e.:

$$E_T = \frac{T_c}{T_c+T_u} \bullet 100\% \tag{9.14}$$

where T_c is the number of conformable results between the two systems a tester evaluated, and T_u the number of unconformable results.

Turing tests with the layout above are informally defined based on empirical experiments and subjective judgements of conformance by testers. The standard real human intelligent system as the reference system in the test is difficult to be defined and it is instable. Also, not all forms of intelligence may be tested by the black box settings such as the cognitive and reflective intelligent capabilities.

9.3.3.4.3 Wang's Intelligent Capability Metrics

Based on the investigation into the nature of intelligence and the GIM intelligence model [Wang, 2006b/06j/07a], a comprehensive measurement on human and system intelligence is proposed by the author known as the Wang's intelligent capability metrics as defined below [Wang, 2007i].

Definition 9.25 The *Intelligent Capability* $\mathfrak{C}_I$ is an average capability of the perceptive intelligence (C_p), cognitive intelligence (C_c), instructive intelligence (C_i), and reflective intelligence (C_r), i.e.:

$$\mathfrak{C}_I = \frac{C_\mathrm{p} + C_\mathrm{c} + C_\mathrm{i} + C_\mathrm{r}}{4} \tag{9.15}$$

where $\mathfrak{C}_I \geq 0$ and $\mathfrak{C}_I = 0$ represents no intelligence.

In Definition 9.25, the four forms of intelligent capabilities can be measured individually according to the following methods given in Definitions 9.26 through 9.29.

Definition 9.26 The *perceptive intelligent capability* C_p is the ability to transfer a given number of data objects or events N_d into a number of information objects in term of derived or related concepts, N_i, i.e.:

$$C_p = \frac{N_\mathrm{i}}{N_\mathrm{d}} \tag{9.16}$$

The perceptive intelligent capability is directly related to the association capability of a testee. The higher the ratio of C_p, the higher the capability of perceptive intelligence. If there is no concept that may be linked or derived for a given set of data or event, there is no perceptive intelligent capability.

Definition 9.27 The *cognitive intelligent capability* C_c is the ability to transfer a given number of information objects N_i in terms of associated concepts into a number of knowledge objects N_k in terms of relations between concepts, i.e.:

$$C_c = \frac{N_\mathrm{k}}{N_\mathrm{i}} \tag{9.17}$$

Definition 9.28 The *instructive intelligent capability* C_i is the ability to transfer a given number of information objects N_i in terms of associated concepts into a number of behavioral actions N_b in terms of number of processes at LRMB Layers 5 and 6, i.e.:

$$C_i = \frac{N_\mathrm{b}}{N_\mathrm{i}} \tag{9.18}$$

Definition 9.29 The *reflective intelligent capability* C_r is the ability to transfer a given number of data objects or events N_d into a number of behavioral actions N_b in terms of number of processes at LRMB Layers 5 and 6, i.e.:

$$C_r = \frac{N_\mathrm{b}}{N_\mathrm{d}} \tag{9.19}$$

On the basis of Definitions 9.25 through 9.29, a benchmark of average intelligent capabilities can be established with large set of test samples. Then, a particular testee's relative intelligent capability or intelligent merit may be derived based on the benchmark.

Definition 9.30 The *relative intelligent capability* $\Delta\mathfrak{C}_I$ is the difference between a testee's absolute intelligent capability $\mathfrak{C}_I$ and a given intelligent capability benchmark $\overline{\mathfrak{C}_I}$, i.e.:

$$\begin{aligned} \Delta\mathfrak{C}_I &= \mathfrak{C}_I - \overline{\mathfrak{C}_I} \\ &= \frac{1}{4}\left(\frac{N_\mathrm{i}}{N_\mathrm{d}} + \frac{N_\mathrm{k}}{N_\mathrm{i}} + \frac{N_\mathrm{b}}{N_\mathrm{i}} + \frac{N_\mathrm{b}}{N_\mathrm{d}}\right) - \overline{\mathfrak{C}_I} \end{aligned} \tag{9.20}$$

9.3.3.5 Theory of Learning and Knowledge Acquisition

According to the CIM model described in Section 9.3.3, the forms of information acquisition and learning of both humans and machine systems are determined by the taxonomy and cognitive properties of internal information inside the brain as stated in the following theorem.

The 28th Principle of Software Engineering

Theorem 9.6 The *generic forms of learning* state there are sufficiently four categories of learning $\mathcal{L}$ known as those of *knowledge* ($\mathcal{L}_\mathrm{k}$), *behaviors* ($\mathcal{L}_\mathrm{b}$), *experience* ($\mathcal{L}_\mathrm{e}$), and *skills* ($\mathcal{L}_\mathrm{s}$), i.e.:

$$\mathcal{L} = (\mathcal{L}_k, \mathcal{L}_b, \mathcal{L}_e, \mathcal{L}_s) \tag{9.21}$$

Theorem 9.6 indicates that learning theories and their implementations in autonomic and intelligent systems should study all four categories of cognitive information acquisition, particularly behaviors, experience, and skills rather than only focusing on knowledge.

Corollary 9.3 Knowledge and behaviors can be learnt indirectly by inputting abstract information; while experience and skills must be learnt directly by hands-on or empirical actions.

According to the object-attribute-relation (OAR) model (see Model 9.8), the result of knowledge acquisition or learning can be embodied by the updating of the existing OAR in the brain. In other words, learning is a dynamic composition of the existing OAR in LTM and the currently created sub-OAR as expressed below.

The 29th Principle of Software Engineering

Theorem 9.7 The *representation of learning results* states that the internal memory in the form of the *OAR* structure can be updated by a conjunction between the existing *OAR* and the newly created sub-*OAR* (*sOAR*), i.e.:

$$\begin{aligned} OAR'\mathbf{ST} &\triangleq OAR\mathbf{ST} \cup sOAR'\mathbf{ST} \\ &= OAR\mathbf{ST} \cup (O_s, A_s, R_s) \qquad (9.22) \end{aligned}$$

9.3.4 THE COGNITIVE MODEL OF THE BRAIN

As argued in the beginning of Section 9.2, although plenty of observations and studies on the brain at physiological and psychological levels have been cumulated, there is a lack of the functional model of the brain at the top level that provides a whole picture of the brain and natural intelligence.

This subsection develops the cognitive models of the brain and natural intelligence by studying relationships between the inherited and acquired life functions in the six layers of LRMB and between the memory components of the brain. It is found that the brain and the natural intelligence can be formally treated as a real-time information processing system [Wang and Wang, 2006] at the functional level in cognitive informatics with the framework of LRMB.

Model 9.5 The *Real-Time Intelligent System Model* of the brain, *NI-Sys*, can be described as a real-time natural intelligent system with an inherited operating system (thinking engine) *NI-OS* and a set of acquired life applications *NI-App*, i.e.:

$$\begin{aligned} NI\text{-}Sys \triangleq\ & NI\text{-}OS \\ & \|\ NI\text{-}App \qquad (9.23) \end{aligned}$$

where *NI-OS* represents the inherited life functions, *NI-App* the developed life functions, and || a parallel relation.

The relationship between *NI-OS* and *NI-App* has been illustrated in Fig. 9.8, where the Level 1 through Level 4 life functions belong to *NI-OS*, and Level 5 and Level 6 life functions are a fundamental part of *NI-App*.

The characteristics of *NI-OS* have been observed as follows [Wang and Wang, 2006]:

- Inherited
- Wired (by neural networks)
- Working subconsciously and automatically
- A real-time system
- Person-independent, common and similar
- Highly parallel and fault-tolerant
- With event/time/interrupt/goal/inference-driven mechanisms

In contrary to the *NI-OS*, the characteristics of *NI-App* have been identified as follows [Wang and Wang, 2006]:

- Acquired
- Partially wired (frequently used functions) and partially programmed (temporary functions)
- Working consciously
- Can be trained and programmed
- Person-specific

The *goal*-driven and *inference*-driven mechanisms are unique features of natural intelligence systems on top of the event-, time-, and interrupt-driven imperative computing mechanisms. The former are autonomously determined by internal states and willingness such as emotions, desires, and rational reasoning. A systematical comparison of the natural intelligent behaviors and the conventional imperative computer behaviors will be provided in Section 15.4.1 toward the development of software science and autonomic computing. Because the inference-driven behaviors have been modeled in Section 3.3, the goal-driven behaviors are described below.

Definition 9.31 A *goal,* denoted by $@g_k\mathbf{ST}$ in the system type $\mathbf{ST}$, is a triple, i.e.:

$$@g_k\mathbf{ST} = (P, \Omega, \Theta) \tag{9.24}$$

where $P = \{p_1, p_2, ..., p_n\}$ is a nonempty finite set of purposes or motivations, Ω is a set of constraints for the goal, and Θ is the environment of the goal.

Therefore, a goal-driven behavior is a cognitive process that is determined by an internal goal in the forms of emotions, desires, willingness, and/or results of rational reasoning.

On the basis of Model 9.5, the functional model of the brain is illustrated in Fig. 9.11 [Wang and Wang, 2006], which shows that the brain consists of the *NI-Sys* (*NI-OS* || *NI-App*), LTM, STM, SBM (connected with a set of sensors), and ABM (connected with a set of servos). In Fig. 9.11 the kernel of the brain is the natural intelligence system (*NI-Sys*), which is *the thinking engine* of the brain as described in the LRMB model.

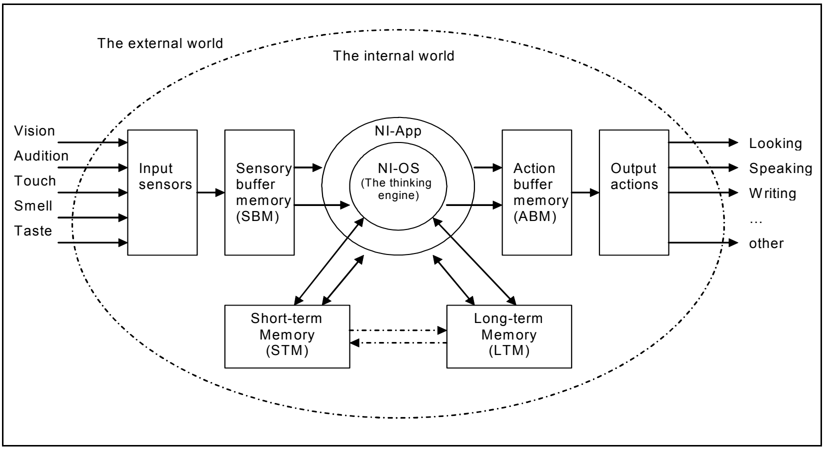


Figure 9.11 The functional model of the brain

Based on Fig. 9.11 and the CMM model, Model 9.5 can be extended as follows.

Model 9.6 The *functional model of the brain, BRAIN*, as a real-time system and a high-level logical model of the brain, describes the functional configuration of the brain and how the *NI-Sys* interacts with the memory system, i.e.:

$$\begin{aligned} BRAIN \triangleq\ & NI\_Sys \\ & \|\, CMM \\ =\ & (\ NI\_OS \\ & \quad \|\, NI\_App \\ & \quad) \\ & \|\, (\ LTM \\ & \quad \|\, STM \\ & \quad \|\, SBM \\ & \quad \|\, ABM \\ & \quad) \end{aligned} \tag{9.25}$$

Eq. 9.25 indicates that although the thinking engine *NI-Sys* is considered the center of the natural intelligence, the memories are essential to enable the *NI-Sys* to properly function, and to keep temporary and stable results stored and retrievable.

The corresponding biological organ of *NI-OS* in neurophysiology is the *thalamus* – a switching center located above the midbrain, which possesses tremendous amounts of connections to almost all parts of the brain, especially cerebral cortex, eyes, and visual cortex [Smith, 1993; Leahey, 1997; Payne and Wenger, 1998; Sternberg, 1998]. Thalamus, the physiological organ corresponding to *NI-OS*, has nearly matured when a person is born. However, the *NI-Ap*p is a set of acquired functions, in forms of knowledge and skills, wired and stored in the LTM in cerebral cortex.

NI-Sys interacts with LTM and STM in a bi-directional way, which forms the basic functionality of the brain as a thinking machine. STM provides working space for the *NI-Sys*, and LTM stores both cumulated information (knowledge) and wired and usually subconscious procedures (skills). It is also noteworthy that ABM plays an important role in the brain to plan and implement human behaviors [Wang and Wang, 2006].

The *NI-Sys* communicates with the external world through inputs and outputs (I/Os). The former are sensorial information, including vision, audition, touch, smell, and taste. The latter are action and behaviors of life functions, such as looking, speaking, writing, and driving. The actions and behaviors generated in the brain, either from *NI-OS* or *NI-App*, are buffered in the ABM before they are executed and outputted to implement the predetermined actions and behaviors.

Unlike a computer, the brain works in two approaches: the internal *goal-* and *inference*-driven processes (in *NI-OS*), and the external *event-*, *time-,* and *interrupt*-driven processes (in *NI-App*). The external information and events are the major sources that drive the brain, particularly for *NI-App* functions. In this case, the brain may be perceived as a passive system, at least when it is conscious, that is controlled and driven by external information. Even the internal willingness, such as goals, desires, and emotions, may be considered as derived information based on originally external information.

It is noteworthy that the subconscious life functions determine the majority of human behaviors and cognitive processes. Although Sigmund Freud identified the psychological effects of sex-related desires of human beings [Freud, 1895; Leahey, 1997], the other more important subconscious life functions as modeled in LRMB as shown in Table 9.5 were probably oversimplified [Wang et al., 2006]. Therefore, a study on the subconscious behaviors of the brain and their mechanisms may be the key to understand how the brain works.

The LRMB reference model and the functional model of the brain establish the foundation for explaining and analyzing the cognitive mechanisms of the brain. Based on them, the age-old problem of the relationship between the brain and the mind can be explained, where the *brain* is both an information processing organ and a real-time controller of the body; while the *mind* is an abstract model of oneself and a thinking engine. The mind, as a virtual model of a person in the brain, is partially programmed and partially wired. The former is evolved for the flexibility of life functions, while the latter is formed for the efficiency of frequently conducted activities, such as eating, writing, and driving.

Definition 9.32 The relationship between the brain and the mind can be analogized by:

$$Brain : mind = computer : program \tag{9.26}$$

The existence of the virtual model of human beings, the mind, can be proven by the following mental phenomena: (a) If one keeps eyes closed, one may still imagine or 'see' everything you learned and remembered, particularly, the visual information, such as the hands and legs. (b) It is reported that patients who lost a leg or arm may think or feel, from time to time, that they still have it as before, because the original cognitive model about the organ at the lower layer of the brain may not be eliminated whenever it had been physiologically created. However, the low-level model may be bypassed or patched at higher layer by conscious cognitive processes.

9.4 Cognitive Informatics Models of Knowledge Representation

Types and structures of memories have been formally modeled in Section 9.2.2, and the cognitive models of the brain have been formally described in Section 9.3. This section examines the cognitive informatics models of information and knowledge and their internal representation in the brain. A set of cognitive models will be developed such as the hierarchical neural cluster model of memory and the object-attribute-relation model of internal knowledge representation, which leads to the explanation of the forms of learning result representation and the estimation of the memory capacity of the brain.

9.4.1 THE HIERARCHICAL NEURAL CLUSTER (HNC) MODEL OF MEMORY

In contrary to the traditional *container* metaphor, the human memory mechanism can be described by a *relational* metaphor [Wang et al., 2003; Wang and Wang, 2006]. The new metaphor perceives that memory and knowledge are represented by the connections between neurons in the brain, rather than the neurons themselves as information containers. Therefore, the cognitive model of human memory, particularly LTM, can be modelled as follows.

Model 9.7 The functional model of LTM is a set of *Hierarchical Neural Clusters* (HNC) with partially connected *neurons* via *synapses*.

The HNC model can be illustrated as shown in Fig. 9.12, where the LTM consists of dynamic and partially interconnected neural networks, where a connection between a pair of neurons by a synapse represents a *relation* between two cognitive objects. The hierarchical and partially connected neural clusters are the foundation for permanent and dynamic information and knowledge representation in LTM.

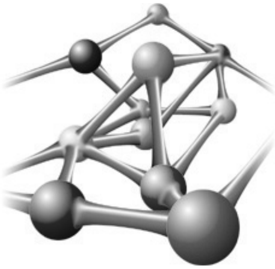
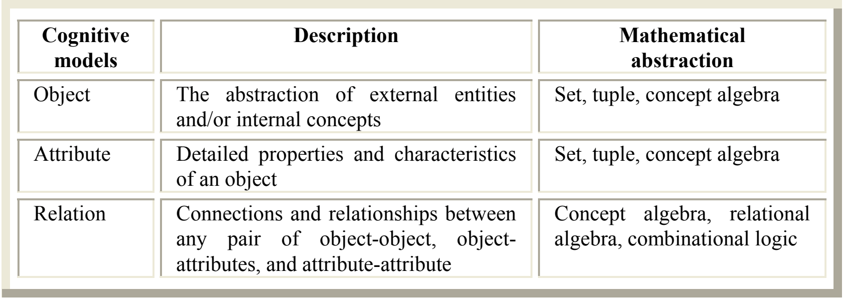


Figure 9.12 LTM: hierarchical and partially connected neural clusters

9.4.2 THE OBJECT-ATTRIBUTE-RELATION (OAR) MODEL OF INTERNAL INFORMATION REPRESENTATION

It is recognized that at the fundamental level, the brain models information by only four meta types [Wang, 2007f; Wang and Wang, 2006]: *object*, *attribute*, and *relation*, as shown in Table 9.8. However, the magnitude of connections among them is extremely high, which can be on the order of $10^{8,432}$ bits according to a recent study [Wang et al., 2003] described in Section 9.4.5.

Table 9.8
The Meta Cognitive Models of the Brain



Cognitive models	Description	Mathematical abstraction
Object	The abstraction of external entities and/or internal concepts	Set, tuple, concept algebra
Attribute	Detailed properties and characteristics of an object	Set, tuple, concept algebra
Relation	Connections and relationships between any pair of object-object, object-attributes, and attribute-attribute	Concept algebra, relational algebra, combinational logic

Based on this observation, an object-attribute-relation model of human memory is developed below, which presents a generic memory model of the brain [Wang, 2007f].

Model 9.8 The *Object-Attribute-Relation* (OAR) model of LTM can be described as a triple, i.e.:

$$OAR \triangleq (O, A, R) \tag{9.27}$$

where O is a set of objects identified by unique symbolic names, i.e.:

$$O \triangleq \{o_1, o_2, \ldots, o_i, \ldots, o_n\} \tag{9.28}$$

For each given object o_i, $1 \leq$ i $\leq n$, A_i is a set of attributes for characterizing the object, i.e.:

$$A_i \triangleq \{A_{i1}, A_{i2}, \ldots, A_{ij}, \ldots, A_{im}\} \tag{9.29}$$

Logically, each A_i may be defined by one of the following set of generic attributes and/or specific attributes:

$A_i \triangleq$ physical attributes
| chemical attributes
| cognitive attributes (image, sound, touch, smell, taste)
| economical attributes
| time-related attributes
| space-related attributes
| categories
| specifications

$$\begin{aligned} &\mid \text{measurements}\\ &\mid \text{usages}\\ &\mid \text{other specific attributes} \end{aligned} \tag{9.30}$$

For each given o_i, $1 \leq$ i $\leq n$, R_i is a set of relations between o_i and other objects or attributes of other objects, i.e.:

$$R_i \triangleq \{R_{i1}, R_{i2}, ..., R_{ik}, ..., R_{iq}\} \tag{9.31}$$

where R_{ik} is a relation r between two objects, o_i and $o_{i'}$, and their attributes A_{ij} and $A_{i'j}$, $1 \leq i \leq n$, $1 \leq j \leq m$, i.e.:

$$\begin{aligned} R_{ik} \triangleq\ & r\,(o_i\,,\, o_{i'}) \\ & \mid r\,(o_i, A_{i'j}) \\ & \mid r\,(A_{ij}, o_{i'}) \\ & \mid r\,(A_{ij}, A_{i'j}),\ \ 1 \leq k \leq q \end{aligned} \tag{9.32}$$

Typically, R_i may be defined by one of the following set of generic relations and/or specific ones:

$$\begin{aligned} R_i \triangleq\ & \text{categories} \\ & \mid \text{types} \\ & \mid \text{entities (real-world objects)} \\ & \mid \text{artifacts (abstract concepts)} \\ & \mid \text{others} \end{aligned} \tag{9.33}$$

where | denotes an alternative relation between defined items.

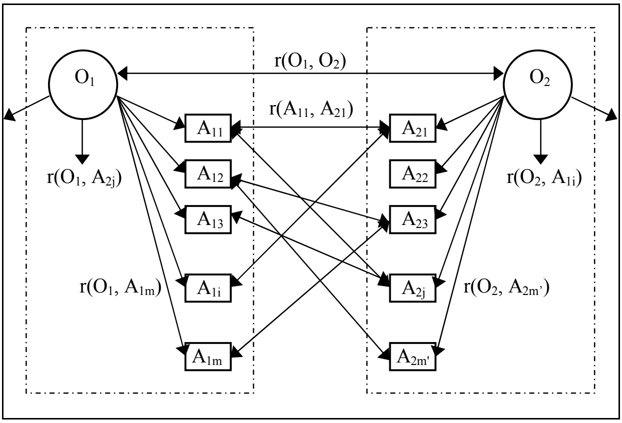


Figure 9.13 The OAR model of internal knowledge representation

An illustration of the OAR model between two objects is shown in Fig. 9.13. The relations between objects can be established via pairs of object-object, object-attribute, and/or attribute-attribute. The connections could be highly complicated, while the mechanism is so simple that it can be deducted to the physiological links of neurons via synapses in LTM.

Example 9.2 The OAR model for representing an object *tree, t*, in LTM of the brain can be expressed as follows:

$$\begin{aligned} t &\triangleq (O, A, R) \\ &= (t, A_t, R_t) \end{aligned}$$

where

$$o \triangleq t = \text{tree}$$

$A_t \triangleq$ sign

| real world reference (image)
| other sensorial attributes (sound, touch, smell, and taste)
| shape (category)
| phonetics (/tri:/)
| plant (category)
| having a trunk (specific attribute 1)
| with leaves (specific attribute 2)
| green (specific attribute 3)
| …

$R_t \triangleq$ forest

| wood
| environment
| furniture
| house
| bird
| …

It is noteworthy as in the OAR model that the internal information and knowledge is presented by the *relations* in the brain. The relational metaphor is totally different from the traditional container metaphor in neuropsychology and computer science, because the latter perceives that memory and knowledge are *stored* in individual neurons or memory cells and the neurons or cells function as containers.

Although the number of neurons in the brain is limited, the possible relations between them may result in an explosive number of combinations that represent knowledge in the human memory. Therefore, the OAR model is capable to explain the fundamental mechanisms of human memory creation, retention, and processing. It also explains why the cortex of the brain is twisted like spaghettis, because this physical configuration allows and increases possible synaptic links between different physiological groups of neurons that represent logically different internal knowledge and information. The establishment of an unusual synaptic link between such logically long-distance but physiologically nearby neurons indicates the mechanism of creation or invention.

9.4.3 THE EXTENDED OAR MODEL OF THE BRAIN

The OAR model developed in Section 9.4.2 provides an abstract conceptual model of LTM and the logical representation of knowledge and learning results [Wang, 2007f]. Mapping it onto the cognitive structure of the brain, an extended OAR model, *EOAR*, can be derived below and illustrated in Fig. 9.14 [Wang and Wang, 2006].

Model 9.9 The *Extended OAR model* of the brain, *EOAR*, states that the external world is represented by *real entities* (REs), and the internal world by *virtual entities* (VEs) and *objects* (Os). The internal world can be divided into two layers known as the *image layer* and the *abstract layer*.

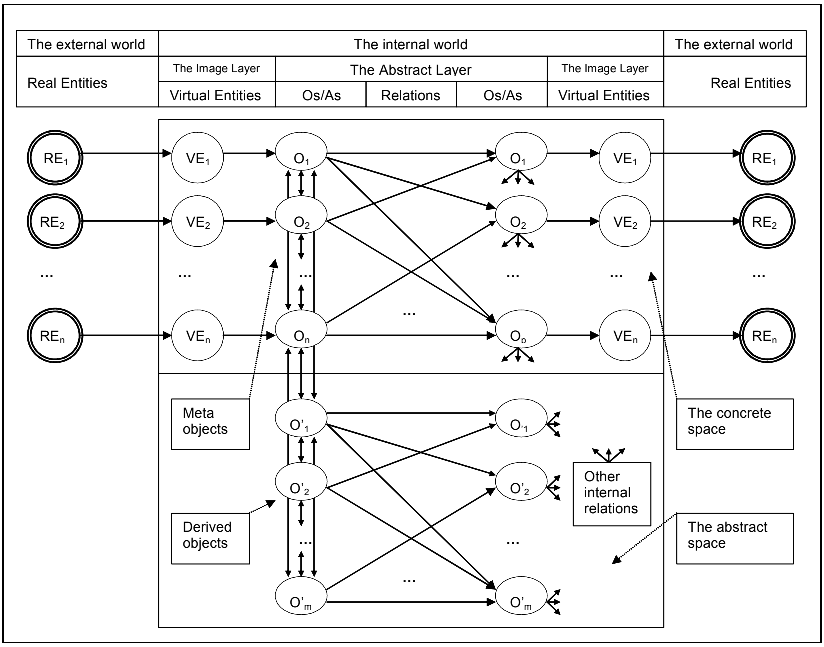


Figure 9.14 The EOAR model of the brain

The *virtual entities* are direct images of the external real-entities located at the image layer. The objects are abstract artifacts located at the abstract layer. There are *meta objects* (O) and *derived objects* (O'). The former are objects directly corresponding to the virtual entities; the latter are objects that are derived internally and have no direct connection with the virtual entities or images of the real-entities.

According to EOAR, there are two inference spaces in the brain: the *concrete* space and the *abstract* space. The former is the memory space for visual reasoning; the latter for abstract reasoning. Objects in both spaces can be further extended into a network of object, attributes, and relations according to the OAR model.

The *abstract space* is an advanced property of the human brain. Other species have no such abstract layer in their brains. This can be proved by the following test with a cat for instance. One may communicate with one's cat by a concrete real entity. However, if you want your cat to pay attention to an entity in the distance by pointing your finger at it, the cat will never understand your intention but simply look at your finger perplexedly! Thus, animals have no *indirect* or *abstract* thinking capability in addition to inherited reflective capability. In other words, the abstract space and the abstract inference capability is a unique power of human brains. The other advantage of the human brain is the tremendous capacity of LTM in cerebral cortex as described in Theorems 9.1 and 9.2.

The 30th Principle of Software Engineering

Theorem 9.8 The *principal intelligent advantages* state that, on the basis of two principal advantages known as the *qualitative* properties (Theorem 9.1) and *quantitative* properties (Theorem 9.2), humans gain the power as the most intelligent species in the world.

It is noteworthy that the cognitive model of the brain is looped. This means that an internal virtual entity is not only abstracted from the real-entity as shown in the left-hand side in Fig. 9.14, but also eventually connected to the entities in the right-hand side. This is the foundation of thinking, reasoning, and other high-level cognitive processes, in which internal information has to be related to the real-world entities, in order to enable the mental processes to be embodied and meaningful.

The EOAR model can be used to describe information representation and its relation to the external world. The external world is described in terms of real entities whereas the internal world is represented by virtual entities and objects. The internal world in its turn consists of two layers: the image and abstract layers. Virtual entity is an image of the real entity in internal world and located at the image layer. Objects in the abstract layer are grouped into two classes: the meta and derived objects. Meta objects have direct relations to virtual entities, while derived objects are represented internally in the abstract space without direct relations to the virtual entities.

The EOAR model reveals that the natural intelligence is memory-based. The EOAR model can be applied to explain the mental processes and cognitive mechanisms as identified in the LRMB model as developed in Section 9.3.1.

9.4.4 THE COGNITIVE MECHANISMS OF LONG-TERM MEMORY

In studying the mechanisms of generic computing machines, Allan Turing found that the basic functionality for computing might mimic the following fundamental capabilities of human brains [Turing, 1936/50]:

- A limited-length memory tape
- A simple addressing capability for searching information in the memory
- Inputs (read) from the memory tape
- Outputs (write) to the memory tape

This finding revealed that intelligence is *memory-based*. Further, it indicates that computing, a high-level artificial intelligence, can be divided into a sequence of *simple memory manipulations*, such as addressing, reading, and writing. Turing's findings can also be used to explain the natural intelligence in cognitive informatics and neuropsychology.

It is observed that the capacity of association cortex, the physiological organ of LTM, has increased dramatically as mankind evolved over time. This provides a foundation for retaining and manipulating information and knowledge in the brain.

On the basis of the cognitive models of the brain developed in the preceding subsection, the cognitive mechanisms and properties of human memory, particularly LTM, can be systematically examined and analyzed. This subsection demonstrates that based on the cognitive models of the brain and memories, a wide range of natural intelligent phenomena and their mechanisms may be formally explained.

9.4.4.1 Cognitive Properties of LTM

Corollary 9.4 The *cognitive properties of LTM* are identified as follows:

- It's directed
- New relations (synaptic connections) can be generated
- It can be reconfigured
- It can be traversed or searched
- It contains loops allowing searches may be carried out from an arbitrary node
- It cannot be sorted

It is noteworthy that LTM cannot be sorted because it is so complicated and it uses spatial synapse connections to represent information. Therefore, accesses to information in LTM have to be carried out by *content-sensitive* and thread-based search. Comparing computer memory and LTM, it can be seen that the former may be fast and randomly accessed by *address-sensitive* mechanisms, while the latter is far more stable and robust. In addition, for a given object, LTM can be accessed through multiple paths. This is one of the cognitive foundations for reasoning and imagination.

Potential approaches to represent information in LTM may be implemented as follows:

a) *A set of new synaptic links*: To represent new information as relations with existing objects. In this case, no new neuron is needed to be allocated.

b) *New neurons and synaptic links*: To represent new information at the leave of a neural cluster. In this case, new neurons belongs to an existing subcluster need to be allocated, and more synaptic links need to be generated within this cluster.

c) *A new neural cluster*: To represent a large set of coherent new information such as systematical knowledge learned in a new course by new neural clusters. In this case, significant numbers of new neurons and their clustering are needed. This is may be implemented by allocating a set of spare neurons in LTM [Squire et al., 1993; Gabrieli, 1998; Payne and Wenger, 1998]. Therefore the total memory capacity of adult brain will not be significantly changed rather than reallocated.

It can be seen that the third case identified above is the most difficult situation in learning and memorization where a set of totally new knowledge has to be physiologically represented and memorized. This explains why the productivity of creative work such as that of software engineering is conservative as stated in Theorem 1.7, because the allocation of neurons and the growing rate of synapses are constrained by natural laws.

The 30th Law of Software Engineering

Theorem 9.9 The *dynamic properties of neural clusters* state that the LTM is dynamic. New neurons (to represent *objects or attributes*) are assigning, and new synaptic connections (to represent *relations*) are creating and reconfiguring all the time in the brain.

9.4.4.2 When is Memory Created in LTM?

The investigation of this fundamental issue can be started by examining the following questions: Why do all mammals need sleep? What is the cognitive mechanism of sleeping?

Sleep as an important physiological and psychological phenomenon was perceived as innate, and few hypotheses and theories have been developed to explain the reason. This subsection explores the role of sleep in LTM establishment.

It is observed that when complicated and highly abstract subjects are taught in a class, students tend to get sleepy. The common sense explanation for this phenomenon is because of boredom or being tired. However, according to Theorem 9.9 and the following lemma, it is a natural and protective action of the brain, which tries to create a sleeping or nap period to remove information that occupies STM into LTM, in order to release more STM space for a current and complicated subject.

Lemma 9.1 *Information memorization in LTM*, as a process to create synaptic relations between neurons according to the OAR model, is functioning subconsciously during sleep.

Lemma 9.1 logically explains the following common phenomena: a) All mammals, including human beings, need to sleep; b) When sleeping, the blood supply to the brain reaches the peak, at about 1/3 of the total consumption of the entire body. However, during daytime the brain just consumes 1/5 of the total blood supply in the body [Smith, 1993; Rosenzmeig et al., 1999; Maquet, 2001; Stickgold et al., 2001]; and c)

According to the *NI-Sys* model, human beings are naturally a real-time intelligent information processing system. Since the brain is busy during daytime, it is logical to schedule the functions of LTM establishment at night, when more processing time is available, and fewer interruptions occur due to external events.

Lemma 9.1 is supported by the following observations and experiments.

Experiment 9.1 A group of UK scientists observed that stewardesses serving long-haul flights had bad memory in common [Wilson and Frank, 1999]. An initial explanation about the cause of this phenomenon was that the stewardesses have been crossing time zones too frequently.

Actually, the above hypothesis is only a partially correct explanation. According to Lemma 9.1, the memory problems of stewardesses are caused by the lack of quality sleep during night flights. As a consequence, the LTM could not be properly established.

Based on Lemma 9.1, the cognitive informatics *theory of sleep* can be derived as follows.

The 31st Principle of Software Engineering

Theorem 9.10 The *cognitive mechanism of sleep* states that sleeping is a subconscious process for LTM establishment, i.e.:

$$\textit{Cognitive purpose of sleep} = \textit{LTM establishment} \qquad (9.34)$$

Theorem 9.10 describes an important finding on one of the fundamental mechanisms of the brain and the cognitive informatics meaning of sleep. Of course, there are obvious physiological purposes of sleep as well, such as resting the body and saving energy.

Corollary 9.5 *Lack of sleep* results in bad memory, because the memory in LTM cannot be properly established.

It was commonly believed that the heart is the only organ in the human body that never takes a rest during the entire life. Corollary 9.5 reveals that so does the brain. The nonresting brain is even more important than the heart because the latter is subconsciously controlled and maintained by the former.

Corollary 9.6 The *subconscious cognitive processes* of the brain, *NI-OS,* do not sleep throughout human life.

The 31st Law of Software Engineering

Theorem 9.11 The *establishment cycle of LTM* states that the cycle of LTM establishment requires at least 24 hours, i.e.:

$$\textit{LTM establishment cycle} \geq 24 \ \ [\text{hrs}] \qquad (9.35)$$

where the 24-hour cycle includes any kind of combinations of awake, asleep, and siesta.

9.4.4.3 How is Memory Created in LTM?

Sleep is recognized as a subconscious process in cognitive informatics. Its cognitive and psychological purpose is to build and update LTM. LTM is updated by searching and analyzing the contents of STM and selecting useful (i.e., the most frequently used) information.

The 32nd Principle of Software Engineering

Theorem 9.12 The *mechanism of LTM establishment* states that the entire memory of information represented as an OAR model in the brain is updated by incorporating the sub-OARs formed in STM based on the following selective criteria:

a) A new sub-OAR in STM was more frequently used in the previous 24 hours;

b) A new sub-OAR in STM was related to the existing OAR in LTM at a higher level of the neural cluster hierarchy;

c) A new sub-OAR in STM was given special attention so that it obtained a higher retention weight.

Based on Theorem 9.12, the relationship between memorization (especially that of LTM) and learning becomes apparent. That is, learning is to gain knowledge or acquire skills, where the results of learning are retained in LTM or ABM, respectively.

9.4.5 THE MEMORY CAPACITY OF HUMAN BRAINS

According to the OAR model as shown in Figs. 9.13 and 9.14, information is represented in the brain by relations, which is a logical model of the synaptic connections among neurons. Therefore, the capacity of human memory is not only dependent on the number of neurons, but also the synaptic connections among them. This mechanism may result in a huge space of exponential combination to represent and retain information in LTM. This also explains why the magnitude of neurons in an adult brain seems stable, however, a tremendous amount of information can be remembered through out the entire life of a person.

On the basis of the OAR model, a mathematical model for estimating the upper limit of human memory capacity can be derived [Wang et al., 2003].

Model 9.10 The *human memory capacity model* states that, assuming there are *n* neurons in the brain, and in average there are *s* synaptic connections between a given neuron and a subset of the rest of them in the brain, the magnitude of the brain's memory capacity C_m can be expressed by the following mathematical model:

$$\begin{aligned} C_m &\triangleq \mathrm{C}_{\mathrm{n}}^{s} \\ &= \frac{n!}{s!(n-s)!} \quad \text{[bit]} \end{aligned} \tag{9.36}$$

where *n* is the total number of neurons, and *s* the number of average partial connections between neurons via synapses.

Model 9.10 indicates that the age-old memory capacity problem in cognitive science and neuropsychology can be reduced to a classical combinatorial problem, with the total potential relational combinations, C_n^s, among all neurons (n = 10^{11}) and their average synaptic connections (s = 10^3) [Marieb, 1992; Pinel, 1997; Gabrieli, 1998]. Therefore, the parameters of Eq. 9.32 can be determined as shown below:

$$\begin{aligned} C_m &\triangleq \mathrm{C}_{\mathrm{n}}^{s} \\ &= \frac{10^{11}!}{10^{3}!(10^{11}-10^{3})!} \quad \text{[bit]} \end{aligned} \tag{9.37}$$

Eq. 9.37 provides a mathematical explanation of the OAR model, which shows that the number of connections among neurons in the brain can be derived by the combination of a huge base and a large number of potential synaptic connections.

The above model seems a simple problem intuitively. However, it turns out to be extremely hard to calculate and is almost intractable using a typical computer, because of the exponential complexity or the recursive computational costs for such large *n* and *m*. However, using approximation theory, and a computational algorithm [Wang et al., 2003], Eq. 9.37 is solved and the result is obtained as:

$$\begin{aligned} C_m &\triangleq \mathrm{C}_{\mathrm{n}}^{s} \\ &= \frac{10^{11}!}{10^3!(10^{11}-10^3)!} \\ &= 10^{8,432} \quad [\text{bit}] \end{aligned} \tag{9.38}$$

Eq. 9.38 reveals that the magnitude of the memory capacity of the brain may reach an order as high as $10^{8,432}$ bits according to the OAR model. This forms the *quantitative* foundation of the natural intelligence. The finding on the magnitude of the human memory capacity reveals an important mechanism of the brain. That is, the brain does not create new neurons to represent new information, instead it generates new synapses between the existing neurons in order to represent new information. The observation in neurophysiology that the number of neurons is kept stable rather than continuous increasing in adult brains [Marieb, 1992; Pinel, 1997; Rosenzmeig et al, 1999] is observed evidences for supporting the relational model of information representation in human memory.

It is interesting to contrast the memory capacities between modern computers and human beings. The capacity of computer memory (mainly the hard disks) has been increased dramatically in the last few decades from a few kB to several GB (10^9 byte), even TB (10^{12} byte). Therefore, with an intuitive metaphor that 1 neuron = 1 bit, optimistic vendors of computers and memory chips perceived that the capacity of computer memory will, sooner or later, reach or even exceed the capacity of human memory [Sabloniere, 2002]. However, according to the finding reported in this subsection, the ratio, *r*, between the brain memory capacity (C_b) and the projected computer memory capacity (C_c) in the next ten years is as follows [Wang, et al., 2003]:

$$\begin{aligned} r &= C_b / C_c \\ &= 10^{8,432} / 8\times 10^{12} \\ &\approx 10^{8,432} / 10^{13} \\ &= 10^{8,419} \end{aligned} \tag{9.39}$$

This indicates that the memory capacity of a human brain is equivalent to at least $10^{8,419}$ modern computers. In other words, the total memory capacity of computers all over the world is far more less than that of a single human brain. Eq. 9.39 also shows the power of the OAR mechanism and configuration of the brain, which uses only 100 billion neurons and their relational combinations to represent and store up to $10^{8,432}$ bit information and knowledge.

The tremendous difference of memory magnitudes between human beings and computers demonstrates the efficiency of information representation, storage, and processing in human brains. Computers store data in a direct and unconsumed manner; while the brain stores information by relational neural clusters. The former can be accessed directly by explicit addresses and can be sorted; while the latter may only be retrieved by content-sensitive search and matching among neuron clusters where spatial connections and configurations themselves represent information.

Investigation into the cognitive models of information and knowledge representation in the brain and the capacity of the memory have been perceived to be one of the fundamental research areas that help to reveal the mechanisms and the potential of the brain. The result developed in this subsection has demonstrated that the magnitude (upper limit) of human memory capacity is excessively higher than those of computers in an order that we never realized. This new factor has revealed the tremendous quantitative gap between the natural and machine intelligence. The finding of this subsection has also indicated that the next generation computer memory systems can be built according to the relational OAR model rather than the traditional container metaphor, because the former is more powerful, flexible, efficient, and is capable of generating a mathematically unlimited memory capacity by using limited number of neurons in the brain or hardware cells in the next generation computers.

9.5 Cognitive Informatics for Software Engineering

The cognitive constraints of software engineering have been identified in Section 1.3.2 as one of the primary constraints of software engineering, which refer to those of intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures. On the basis of improved understanding of the cognitive properties of software as intelligent behaviors, this section describes the cognitive informatics principles of software engineering, which encompass cognitive constraints on software productivity, software engineering psychology, the cognitive foundation of software comprehension, software engineering skills and experiences, and software agent systems.

9.5.1 COGNITIVE CONSTRAINTS ON SOFTWARE PRODUCTIVITY

According to Theorem 1.6, conservative productivity is a basic constraint of software engineering due to cognitive complexity and due to the cognitive mechanism in which abstract artifacts need to be represented physiologically in the brain via growing synaptic neural connections.

The fact that before any program is composed, an internal abstract model must be created inside the brain [Wang, 2007a; Wang and Wang, 2006] reveals the most fundamental constraint of software engineering, i.e., software is generated and represented in the brain before it can be transferred into the computer. Because the growth of the human neural system is naturally constrained, as described by the 24-hour law in Theorem 9.11, it is very hard to dramatically improve the productivity of software development.

According to the statistics of software engineering literature [Boehm, 1987; Dale and Zee, 1992; Jones, 1981/1986; Livermore, 2005], the average productivity of software development was about 1,300 LOC/person-year in the 1970s, 2,500 LOC/person-year in the 1980s, and 3,000 LOC/person-year in the 1990s, where management, quality assurance, and supporting activities are included, and LOC is the unit of the symbolic size of software in terms of Line of Code. It is obvious that the productivity in software engineering has not been increased remarkably in the last four decades independent of programming language development. In other words, no matter what kind of programming language is used, as long as they are for human programming, there is no difference in principle. This assertion can be proved by asking the following question: Have you ever known an author in literature who is productive because he/she writes in a specific natural language?

Productivity of software development is the key among all the cognitive, time, and resources constraints in software engineering. The other constraints can be overcome as a result of the improvement of software engineering productivity.

Lemma 9.2 The *key approaches to improve software development productivity* are:

a) To explicitly express software architectures and behaviors in *denotational mathematics*;

b) To investigate the theories of rational software *engineering organization*; and

c) To design tools that lead to *automatic software code generation* based on the denotational system models.

All theories and approaches explored throughout this book put emphases on this ultimate goal of software engineering. The most significant and unique characteristic of software engineering lays on the need for the contemporary denotational mathematics in order to rigorously and explicitly model the architectures and behaviors of software systems and to reduce the cognitive complexity of software engineering, which are challenging the limitation of human cognitive capacity in large scale software development.

9.5.2 SOFTWARE ENGINEERING PSYCHOLOGY

Software engineering psychology is a part of the domain of cognitive informatics. It is perceived that the nature of software and software engineering is in many ways closer to cognitive psychology than engineering and technology, because software is intangible and complicated abstract artifacts created by human brains [Weinberg, 1971; Wang, 2004b]. The best software often takes advantages of human creativity.

There are two types of programmers in a psychological view: the *realistic* and *idealistic* ones. The former may be suitable for coding, testing, and quality control; while the latter may be good at solution seeking, Graphic User Interface (GUI) design, and carrying out tasks as system analysts. For both above categories, the following psychological requirements for software engineers are important in software engineering:

- Abstract-level thinking
- Imagination of static descriptions in terms of dynamic behaviors
- Organization capability
- Cooperation and team working attitude
- Long-term focus of attentions
- Preciseness

- Reliability
- Expressive capability in expressions and communications

There is a special phenomenon in software engineering that anybody who is able to use a programming language may claim that one can programming or even be a software engineer. This is just like that one who acquires reading and writing skills in a natural language may claim oneself as a writer; or one who is able to build a simple shelter or doghouse may claim oneself as a civil engineer.

It is stressed that knowing a programming language is not enough to be a qualified software engineer. So what else is needed? The following characteristics of software engineering practice may be considered as the basic requirements:

- How to reuse proven software components into a well defined architecture
- How to conform with standards and empirical best practice
- How to organize a coherent team in engineering approach
- How to control a complicated software engineering process

Hence, it is interesting to contrast and analyze the differences between professionals and amateurs in software engineering. Professional software engineers are persons with professional cognitive models and knowledge on software engineering. They are trained with:

- Fundamental knowledge that governs software and software engineering practices
- Basic principles and laws of software engineering
- Proven algorithms
- Problem domain knowledge
- Problem solving experience
- Program developing tools / environments
- Solid programming skills in multiple programming languages
- A global and insightful view on system development, including its required functionalities as well as exception handling and fault-tolerance strategies

However, amateurish programmers are persons who know only one or a couple of programming languages but lack awareness and training as those professionals identified above. Amateurs may be characterized as follows in their software engineering knowledge structure:

- Ad hoc structure of programming knowledge
- Limited programming experience and skills
- Eager to try what is directly required before a system architecture is designed
- Tend to focus on details without a global and systematic view

The key difference between professionals and amateurs is whether their knowledge and skills are wired or temporarily programmed in the brain. Professional software engineers possess wired skills in the brain for programming, and with a global view on software development. They focus not only on required functions, but also on exceptions handling and fault-tolerance in implementing the required functions. However, amateur programmers possess ad hoc programming knowledge, eager to try what is directly required, and tend to focus on details without a global and systematic view.

9.5.3 THE COGNITIVE FOUNDATION OF SOFTWARE COMPREHENSION

Comprehension is the action or capability of understanding. Comprehension is a higher cognitive process of the brain that searches relations between a given object or attribute and other objects, attributes, and relations in LTM, and establishes a representational model for the object or attribute by connecting it to appropriate clusters of memory. It is recognized that although knowledge and information are powerful, before any information can be possessed and processed it should be comprehended properly.

Comprehension can be modeled as a cognitive process that can be carried out by the following steps:

a) To search relations from *real entities* to *virtual entities* and/or existing *objects* and *attributes*.

b) To build a partial or adequate OAR model of the entity.

c) To wrap up the OAR model by classifying and connecting it to appropriate clusters of the LTM.

d) To memorize the new OAR model and its connections in LTM.

The above cognitive process of comprehension is informally described in Fig. 9.15. As shown in Fig. 9.15, the first step to comprehend a given real entity or concept is to search the corresponding virtual entity and its relations to objects in the abstract layer. Depending on the results of the search for relations, the next step can be different. The ideal search result is that adequate relations have been found. In this case, comprehension is almost reached. The other possible result is that a partial comprehension or incomprehension is obtained when a partial OAR model is built, or a very low level of comprehension is reached. A partial OAR model is a sub-OAR model, where no sufficient relations have been found. In an extremely case, for a totally new concept in comprehension, probably only an ID of the concept is existent. These different outcomes in comprehension indicate that everything is comprehensible; only the extent of comprehension is varying in a range from 100% to 0%.

If the findings are sufficient then the brain builds a sub-OAR model for the given object (box 7). When the model is built it needs to be connected to the most appropriate cluster in the entire OAR of the brain (box 8). Only after this step is comprehension achieved.

However, if the findings are not adequate after the search, the brain builds a partial OAR model with limited information and requires further actions to obtain additional information from external resources (box 10). For example, if one could not recall or do not know the meaning of a given word by existing knowledge, one may look for it in a dictionary or encyclopedia. The search from external resources may be a repetitive process. For instance, if one cannot find the meaning of the word in the dictionary, then someone may be asked for its meaning.

After searching several times in external resources, the brain checks again whether findings are adequate (box 11). If so, Steps 7 to 9 will be repeated. Otherwise, it is regarded that an incomprehension has been achieved (box 12) at this given moment, but still the results are remembered in LTM. In this situation a sub-OAR model is stored and it may be simply an ID for an unknown concept for future comprehension. For example, when reading one may come across a completely new word, and one could neither find it from any dictionary nor from other resources. Then, the word may simply be remembered by rote without knowing its meaning. One may remember it like “the word has seven letters”, “it starts with letter *k*”, “and it resembles the word …” etc, but without any significant attribute. For another example, when one looks at an abstract painting, one may see that some parts of the picture resemble a hand and other parts resemble something else, but what the whole picture expresses cannot be understood. In both examples the brain may still able to build a partial OAR model with a few insignificant attributes and relations based on the limited clues. The final step in the comprehension process is to memorize the OAR model in LTM by which the comprehension process is completed.

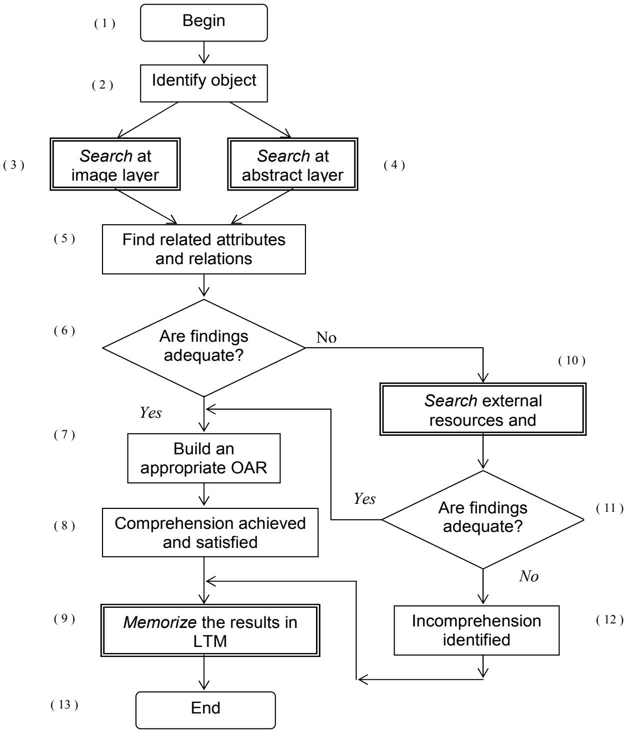


Figure 9.15 The cognitive model of the comprehension process

On the basis of the explanation of the comprehension process of the brain, a formal description of the cognitive process of comprehension in RTPA is presented in Fig. 9.16. The relationship between the comprehension process and other meta cognitive processes, such as search, memorization, and knowledge representation, is also identified in the process.

Definition 9.33 *Program comprehension* is a cognitive process to understand a given software system in dimensions of architecture, static behaviors, and dynamic behaviors, and their relationships.

Program comprehension is a cognitive process to understand a given software system in terms of the architecture, static behavior, and dynamic behavior dimensions as well as their relationships and interactions. The comprehension of software architectures can be supported by concept algebra and system algebra [Wang, 2006e/06d], which focuses on data objects and frameworks of a software system. The comprehension of software static and dynamic behaviors can be supported by RTPA, which focuses on individual behaviors and the interactions between computing actions and the data objects.

The Cognitive Process of Comprehension

```
Comprehension (I:: TheConcept$\mathbf{S}$;
                    O:: OAR(TheConcept$\mathbf{S}$)$\mathbf{ST}$, Lcomprehension$\mathbf{N}$) $\triangleq$
{
 o$\mathbf{S}$ := TheConcept$\mathbf{S}$
 $\to$ (
      Scope$\mathbf{S}$ := ImageLayerOfLTM$\mathbf{S}$
            $\rightarrowtail$ Search (I:: o$\mathbf{S}$, Scope$\mathbf{S}$;  O:: $A_r\mathbf{ST}$, $R_r\mathbf{ST}$)
    || Scope$\mathbf{S}$ := AbstartcLayerOfLTM$\mathbf{S}$
            $\rightarrowtail$ Search (I:: o$\mathbf{S}$, Scope$\mathbf{S}$;  O:: $A_r\mathbf{ST}$, $R_r\mathbf{ST}$)
    )
 $\to A_r\mathbf{ST} = \{a_1, a_2, \ldots, a_n\}$
 $\to R_r\mathbf{ST} = \{o_1, o_2, \ldots, o_m\}$
 $\to$ ( � $(A_r\mathbf{ST} \subseteq A_{meta}\mathbf{ST}) \wedge (R_r\mathbf{ST} \subseteq O_{meta}\mathbf{ST}) \wedge (A_r\mathbf{ST} = \phi \vee R_r\mathbf{ST} = \phi)$
        $\to$ PL1$\mathbf{S}$: OAR(o$\mathbf{S}$)$\mathbf{ST}$ = {o$\mathbf{S}$, $A_r\mathbf{ST}$, $R_r\mathbf{ST}$}
    | � ~
        $\to$ Scope$\mathbf{S}$ := ExternalResources$\mathbf{S}$
        $\rightarrowtail$ Search (I:: o$\mathbf{S}$, Scope$\mathbf{S}$; O:: $A'_r\mathbf{ST}$, R(r$\mathbf{ST}$)
        (  �  $(A_r\mathbf{ST} \subseteq A_{meta}) \wedge ( (R_r\mathbf{ST} \subseteq O_{meta}) \wedge (A_r\mathbf{ST} = \phi \vee R_r\mathbf{ST} = \phi)$
              $\curvearrowright$ PL1$\mathbf{S}$
         | � ~
              $\to$ OAR(o$\mathbf{S}$)$\mathbf{ST}$ = {o$\mathbf{S}$, $A_r\mathbf{ST}$, $R_r\mathbf{ST}$}
        )
    )
  $\rightarrowtail$ Memorization (OAR(o$\mathbf{S}$)$\mathbf{ST}$)

  $\to \mathop{\mathrm{R}}\limits_{i=1}^{n}$ (OAR($R_i\mathbf{ST}$)$\mathbf{ST}$ = {$o_i\mathbf{S}$,  $A_i\mathbf{ST} \cup o_i\mathbf{S}$,  $R_i\mathbf{ST} \cup o_i\mathbf{S}$}
}
```

Figure 9.16 Formal description of the comprehension process in RTPA

9.5.4 SOFTWARE ENGINEERING SKILLS AND EXPERIENCES

As revealed in Theorem 9.4 and Table 9.6, although knowledge can be acquired indirectly in learning, experiences and skills must be obtained directly by empirical actions.

In discussing "what makes a good software engineer" in a panel, Marcia Finfer (1989) believed: "the answer, in my opinion, is simply the combination of both innate skill and significant experience in building real systems against a set of functional and performance requirements and a given budget and schedule." This shows that professional experience is a primary factor of software engineers, where an experience of problem complexity beyond 5,000LOC as given in Definition 1.2 is a necessary benchmark. Also, the possession of fundamental principles and laws of software engineering is essential towards excellent software engineers.

The acquisition of professional skills may be described by a cognitive process. For example, in a complex building, if a newcomer is guided through once, he or she may still have difficulty to manage to remember the ways in the beginning. Because an abstract model of the building has yet to be built in his/her LTM by wired neural networks, which takes time (see the 24-hour law in Theorem 9.11). The acquisition of skills for driving is another example that explains skill acquisition according to cognitive informatics principles.

It is observed that programming skills and software engineering experiences can not be transferred directly from person to person without hands on practice. Therefore, it is curious to seek what made skill and experience transfer hard in software engineering below.

The means of experience repository in software engineering can be categorized into the following types:

- Best practices
- Know-how
- Lessons learnt
- Failure reports
- New technology trial reports

All the above items of software engineering experience seem to be hard to gain indirectly by reading. The major cognitive reasons explaining these phenomena can be described as follows:

- The brain has no direct skill or experience transfer mechanism. Before it is acquired and possessed, any skill has to be physiologically modeled in the ABM, and any experience has to be reprogrammed into an action process in ABM.
- The only way for skill and experience acquisition is *learn by doing*, or *trial and error*. People, usually, have to make the same mistakes, at least to simulate them, in order to learn and remember a specific experience.

- Each brain is unique as described in Theorem 1.2, because of individual physiological differences, cognitive style differences, personality differences, and learning environment differences.

Therefore, although computers, as external or extended memory and information processing systems for the brain, provide a new possibility for people to learn things faster than ever, the internal representation of abstract knowledge or active behaviors such as skills and experiences must still rely on wired inter-connections among neural clusters and obey the same cognitive laws of the brain as described in Theorems 9.4 through 9.11.

9.5.5 SOFTWARE AGENT SYSTEMS

Definition 9.34 A *software agent* is an intelligent software system that autonomously carries out robotistic and interactive applications based on goal-driven mechanisms [Wang, 2003d/07f].

The theoretical foundation of agent systems is cognitive informatics. Because a software agent may be perceived as an application-specific virtual brain according to Theorems 3.4 and 9.5, functions of an agent are mirrored human behaviors. The fundamental characteristics of agent-based systems are autonomic computing, goal-driven action-generation, and self-learning.

Machine perception is a basic capability required for a software agent system, where perception refers to the capability for thinking and interpreting data and information acquired from external world and events based on existing internal knowledge and how the data and information may be transformed into behaviors based on cognition [Matlin, 1994; Chorafas, 1998].

LRMB [Wang et al., 2006] may be used as a reference model for agent-based technologies. This is a fundamental view toward the formal description and modeling of the architectures and behaviors of agent systems, which are designed to do something repeatable in context, to extend human capability, reachability, and/or memory capacity.

It is found that both human and software behaviors can be described by a 3-D representative model comprising *action*, *time*, and *space*. For agent system behaviors, the three dimensions are known as *mathematical operations*, *event/process timing*, and *memory manipulation* [Wang, 2003d]. The 3-D behavioral space of agents can be formally described by RTPA [Wang, 2002a] that serves as an expressive notation system for describing thoughts and notions of dynamic system and human behaviors as a series of actions and cognitive processes.

Recent research reveals that the foundations of agent technologies [Wang, 2007f] are rooted in cognitive informatics theories and autonomic computing methodologies. Along with the latest development of cognitive informatics and autonomic computing, self-organizing, self-managing and non-imperative autonomic agent systems are emerging. An autonomic agent system is an intelligent software system that takes rational actions in the pursuit of its agenda via goal-, inference-, and event-driven behaviors. Because cognitive informatics investigates the internal information processing mechanisms and processes of the brain and natural intelligence, its research results underpin engineering applications of autonomic agent systems.

9.6 Cognitive Complexity of Software

One of the central problems in software engineering is its inherited complexity. Since software is the product of human intelligence, cognitive informatics plays an important role in understanding the fundamental characteristics of software complexity.

The existing measures for software complexity can be classified into two categories: the macro and the micro measures of software complexity. Two major macro complexity measures of software were proposed by Basili and Kearney, respectively. The former considered software complexity as 'the resources expended [Basili, 1980a/80b].' The latter perceived that the complexity is the extent of difficulty in programming [Kearney et al., 1986].

The micro measures are based on program code disregarding comments and stylistic attributes. This type of measures typically depends on program sizes, program flow graphs, or module interfaces such as Halstead's *software science metrics* [Halstead, 1977] and the *cyclomatic complexity* of McCabe [McCabe, 1976]. However, Halstead's software science purely calculates the numbers of operators and operands, but does not consider the internal structures of software components; while McCabe's cyclomatic measure focuses on the internal structures of software flow graphs without considering the data objects and I/O structures of software systems.

In cognitive informatics, it is found that the functional complexity of software in design and comprehension is dependent on three fundamental factors known as the *internal processes*, *inputs,* and *outputs* as modeled in the system theory in Section 10.5.5. The new measure for software cognitive complexity is a measure of cognitive and psychological complexity of software as a human intelligent artifact. The cognitive complexity measure takes into account both internal process structures of a software system and the I/O data objects under its processing [Wang, 2005j/06c; Shaw and Wang, 2003].

This section describes the complexity of software by examining the cognitive weights of BCS's in software systems as modeled in Section 5.4.1. A new concept of software cognitive complexity is introduced, which provides a foundation for cross-platform analysis of complexities, sizes, and comprehension efforts of software specifications and implementations in the phases of design, implementation, and/or maintenance in software engineering.

9.6.1 THE RELATIVE COGNITIVE WEIGHTS OF GENERIC SOFTWARE STRUCTURES

To design and comprehend a given program, the focuses are naturally put on the control logic of a software system represented by BCS's and the behaviors they may operate on the data objects. BCS's are a set of essential flow control mechanisms that are used for building logical architectures of software, as described in Section 5.4.1, independent of programming languages.

Definition 9.35 The *cognitive weight of software* is the extent of difficulty or relative time and effort for comprehending the function and semantics of a given program.

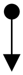
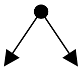
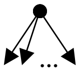
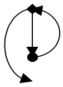
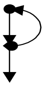
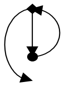
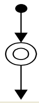
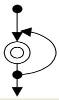
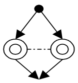
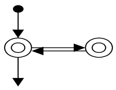
It is almost impossible to measure the cognitive weights of a program at statement level because of their variety and language dependency. However, it is found that it is feasible if the focus is put on the BCS's of the software systems [Wang, 2005j], because there are only ten BCS's in programming no matter what kind of programming language is used. Detailed definitions of the BCS's and their syntaxes may be referred to Section 5.4.1 and Table 9.9.

Definition 9.36 The *relative cognitive weight of a BCS,* $w_{BCS}(i)$, $1 \le i \le 10$, is the relative time or effort spent on comprehending the function and semantics of a BCS against that of the sequential BCS.

$$w_{BCS}(i) = \frac{t_{BCS}(i)}{t_{BCS}(1)}, \quad 1 \le i \le 10 \tag{9.40}$$

where $t_{BCS}(1)$ is the relative time spent on the sequential BCS.

Table 9.9
The Relative Cognitive Weights of BCS's

BCS (i)	**Description**	**Nota-tion**	**Structural model**	**RTPA model**	**Calibrated weight w_{BCS}(i)**
1	Sequence	$\rightarrow$		$P \rightarrow Q$	1
2	Branch	$\vert$		�$exp\mathbf{BL} = \mathbf{T} \rightarrow P$ $\vert$ �$\sim \rightarrow Q$	3
3	Switch	$\vert$ … $\vert$		�$\mathrm{exp}\mathbb{T}$ **=** $\mathrm{i} \rightarrow P_i$ $\vert \sim \rightarrow \oslash$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$	4
4	While-loop	R^*		$\overset{\mathbf{F}}{\underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{R}} P$	7
5	Repeat-loop	R^+		$P \rightarrow \overset{\mathbf{F}}{\underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{R}} P$	8
6	For-loop	R^i		$\overset{n\mathbf{N}}{\underset{i\mathbf{N}=1}{R}} P(i\mathbf{M})$	7
7	Function call	$\rightarrowtail$		$P \rightarrowtail F$	7
8	Recursion	$\circlearrowleft$		$\overset{0}{\underset{i\mathbf{N}=n\mathbf{N}}{R}} P^{i\mathbf{M}} \circlearrowleft P^{i\mathbf{M}-1}$	11
9	Parallel	$\vert\vert$		$P \vert\vert Q$	15
10	Interrupt	↯		P ↯ Q	22

Definition 9.37 The *unit of cognitive weight of BCS's,* CU, is the relative time spent on the *sequential* BCS, i.e.:

$$\begin{aligned} w_{BCS}(1) &= \frac{t_{BCS}(1)}{t_{BCS}(1)} \\ &= 1 \quad [\mathrm{CU}] \end{aligned} \tag{9.41}$$

The ten categories of BCS's described above are profound architectural attributes of software systems. These BCS's and their variations are modeled and illustrated in Table 9.9, where their equivalent cognitive weights (w_{BCS}(i)) for determining a component's functionality and complexity are defined based on a set of psychological experiments in cognitive informatics [Wang, 2005j].

9.6.2 PSYCHOLOGICAL EXPERIMENTS ON THE COGNITIVE WEIGHTS

The method of the psychological experiments [Osgood, 1953; Wang, 2005j] for calibrating the relative cognitive weights of the ten BCS's is based on the axiom that the relative time spent on comprehending the function and semantics of a BCS is proportional to the relative cognitive weight of effort for the given BCS.

Although different persons may comprehend the set of the ten BCS's in different speeds according to their experience and skill in a given programming language, the relative effort or the relative weight spent on each type of the BCS's are statistically stable, assuming the relative weight of the *sequential* BCS is one CU according to Definition 9.36.

Definition 9.38 The generic *psychological experimental method* for establishing a benchmark of the cognitive weights of the ten BCS's can be conducted in the following steps:

a) Record the start time T_1 in mm:ss.

b) Read the given program Test_1:

```
int Test1 (int A=1, B=2) {
    return A + B;
}
```

c) Answer: A + B = ?

d) Record the end time T_2 in mm:ss.

e) Calculate the relative cognitive weight of the sequential construct BCS_1 according to Eq. 9.40, i.e., $t_{BCS}(1) = (T_2 - T_1)$, and $w_{BCS}(1) = t_{BCS}(1) / t_{BCS}(1) \equiv 1$.

Applying the generic experimental method as given in Definition 9.38 by simply replacing the testing program in Step (b), a set of cognitive psychological experiments can be carried out on the ten BCS's by empirical studies in software engineering [Wang, 2005j]. The following are examples adopted in the experiments.

Experiment 9.2 The relative cognitive weights of the *branch BCS* (ITE), $w_{BCS}(2)$, is determined using the following program in Java:

```
int Test2 (int A=2, B=3) {
     if A >= B
        return A - B
        else return B - A
}                                                    (9.42)
```

Experiment 9.3 The relative cognitive weights of the *for-loop BCS* (R^i), $w_{BCS}(6)$, is determined using the following program in Java:

```
int Test4 (int A=2, B=3) {
     for (int i=0; i<3; i++) {
          A := A + i
          }
     return A + B
}                                                    (9.43)
```

9.6.3 CALIBRATION OF THE RELATIVE COGNITIVE WEIGHTS OF BCS'S

The cognitive psychological experiments as designed in the previous subsection have been carried out in undergraduate and graduate classes in software engineering. Based on 126 experiment results, the equivalent cognitive weights of the ten fundamental BCS's are calibrated as summarized in Table 9.9.

On the basis of the calibrated cognitive weights of BCS's, the cognitive complexity of software can be rigorously analyzed. Detailed methodology for measuring the cognitive complexities of software will be provided in Section 10.7 with comparative studies against existing software complexity measures.

9.7 Summary

Cognitive informatics (CI) has been recognized as a new frontier that studies internal information processing mechanisms and processes of the brain, and their applications in computing and the IT industry. Cognitive informatics has been described as a profound transdisciplinary research field that tackles the common root problems of modern informatics, computation, software engineering, AI, cognitive science, nueropsychology, and life sciences.

Large-scale software systems have been recognized as highly complicated systems that mankind has ever handled or experienced. Although software as a unique abstract artifact does not obey any known physical laws, it is constrained by the laws of informatics, cognitive informatics, mathematics, and systems.

This chapter has described the cognitive informatics and intelligent behavioral metaphor of software and software engineering. The theories of cognitive informatics and its potential impacts on, and applications in, information-based sciences and engineering disciplines, particularly software engineering, have been explored. As a result, the **cognitive informatics foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Cognitive Informatics Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 9. Cognitive Informatics Foundations of SE

■ Cognitive Informatics
- Cognitive philosophy
- Neural informatics foundations of the brain
  - Neurons and synapses
  - Physiological structure of the brain
  - Cognitive models of memories

- The emergence of cognitive informatics
- The theoretical framework of cognitive informatics
  - The fundamental theories of cognitive informatics
  - The domain of cognitive informatics

■ Cognitive Informatics Models of the Brain

- The layered reference model of the brain (LRMB)
  - The structure of LRMB
  - Cognitive layers of LRMB
  - The configuration of the cognitive processes of LRMB
- Cognitive properties of internal information
- Natural intelligence vs. artificial intelligence
  - The nature of intelligence
  - Taxonomy of intelligence
  - The model of natural intelligence
  - Measurement of intelligence
  - Theory of learning and knowledge acquisition
- The cognitive model of the brain
  - The functional model of the brain
  - The cognitive mechanisms of the brain

■ Cognitive Informatics of Knowledge Representation

- The hierarchical neural cluster (HNC) model of memory
- The object-attribute-relation (OAR) model of internal information representation
- The extended OAR model of the brain
- The cognitive mechanisms of long-term memories
- The memory capacity of human brain

■ Cognitive Informatics for Software Engineering

- Cognitive informatics properties of SE
- SE psychology
- The cognitive foundation of software comprehension
- SE skills and experiences

■ Cognitive Complexity of Software

- The relative cognitive weights of generic software structures
- Psychological experiments on the cognitive weights
- Calibration of the relative cognitive weights of BCS’s
- Measurement of cognitive complexity of software

SIGNIFICANT FINDINGS OF THIS CHAPTER

- The **quantitative advantage of human brain:** The magnitude of the memory capacity of the brain is tremendously larger than that of the closest species.

- The **qualitative advantage of human brain:** The possession of the abstract layer of memory and the abstract reasoning capacity makes human brain profoundly powerful on the basis of the quantitative advantage.

- **Cognitive Models of Memories:** *Memory* is the foundation for maintaining a stable state of an animate system. It is also the foundation for any form of natural and machine intelligence.

  - Latest discoveries in neuroscience and cognitive informatics indicate that **LTM** is dynamically reconfiguring, particularly at the lower levels of the neural clusters. This explains the mechanisms of memory establishment, enhancement, and evolution that are functioning everyday in the brain.

  - A new type of memory known as **ABM** is identified recently that denotes the memory functions for the output-oriented actions, skills, and behaviors, such as a sequence of movement and a pre-prepared verbal sentence.

- **LTM:** The HNC model indicates that the LTM is dynamic. New neurons (to represent *objects or attributes*) are assigning, and new connections (to represent *relations*) are creating and reconfiguring all the time in the brain.

  - **Knowledge** in LTM, as synaptic connections between neurons according to the OAR model, is dynamically grown during sleeping.

- **Properties of LRMB:** The **subconscious layers** of the brain represented by *NI-OS* are inherited, fixed, and relatively mature when a person is born. Therefore, the subconscious cognitive function layers are not directly controlled and accessed by the conscious life function layers. The **conscious layers** of the brain, represented by *NI-App*, are acquired, highly plastic, programmable, and can be controlled intentionally based on willingness, goals, and inferences.

- **Inherited and acquired brain functions:** According to the logical model of the brain, *genes* may only explain things at the level of *inherited brain functions*, rather than at the level of *acquired brain functions*, because the latter cannot be directly represented by genes in order to be inherited;

instead they should be represented as internal knowledge, behaviors, experience, and skills.

- **Intelligence**, in the *narrow sense*, is a human or a system ability that transforms information into behaviors; and in a *broad sense*, it is any human or system ability that autonomously transfers the forms of abstract information between *data, information, knowledge,* and *behaviors* in the brain.

  - **Natural intelligence** $\Im$ can be classified into four forms called the *perceptive intelligence* ($\Im_p$: $D \rightarrow I$), *cognitive intelligence* ($\Im_c$: $I \rightarrow K$), *instructive intelligence* ($\Im_i$: $I \rightarrow B$), and *reflective intelligence* ($\Im_r$: $D \rightarrow B$), where D stands for data.

- The **representation of learning results:** The consequence of learning is represented by a updating of the internal memory in the form of the *OAR* structure by a conjunction between the existing *OAR* and the newly created sub-*OAR* (*sOAR*).

- **Cognitive mechanisms of the brain:** a) LTM establishment is a subconscious process; b) The long-term memory is established during sleeping; c) The general acquisition cycle of LTM requires at least 24 hours; e) The mechanism of LTM establishment is to update the entire memory of information represented as an OAR model in the brain; and f) Eye movement and dreams play an important role in LTM creation.

- **The *human memory capacity model* of the brain:** Assuming there are n neurons in the brain, and in average there are s connections between a given neuron and a subset of the rest of them in the form of synapses, the magnitude of the brain's memory capacity can be determined with the total potential relational combinations C_n^s, among all neurons ($n = 10^{11}$) and their average synapses ($s = 10^3$), which results in $10^{8,432}$ bits.

  - The tremendous difference of memory magnitudes between human beings and computers demonstrates the efficiency of information representation, storage, and processing in the human brains. Computers store data in a direct and unconsumed manner; while the brain stores information by relational neural clusters. The former can be accessed directly by explicit addresses and can be sorted; while the latter may only be retrieved by content-sensitive search and matching among neuron clusters where spatial connections and configurations themselves represent information.

- **Cognitive informatics properties of software engineering:** The fact that before any program is composed, an internal abstract model must be created inside the brain reveals the most fundamental constraint of software engineering, i.e., software is generated and represented in the brain before it can be transferred into a computer. Because the growth of the human neural system is naturally constrained as described by the 24-hour law, it is very hard to dramatically improve the productivity of software development. This is identified as one of the basic constraints of software engineering known as conservative productivity due to the cognitive mechanism in which abstract artifacts need to be represented physiologically in the brain via growing synaptic neural connections.

- The **cognitive complexity weights** of software are a new functional complexity measure based on the ten BCS's in programming. The **relative cognitive weight of a BCS**, w_{BCS}, is the relative time or effort spent on comprehending the function and semantics of a BCS against that of the sequential BCS. The calibrated w_{BCS}(*Sequence, branch, switch, while-loop, repeat-loop, for-loop, function call, recursion, parallel, interrupt*) = (1, 3, 4, 7, 8, 7, 7, 11, 15, 22).

- **Psychological requirements for software engineers:** a) Abstract-level thinking; b) Imagination of dynamic behaviors with static descriptions; c) Organization capability; d) Cooperative attitude in team work; e) Long-term focus of attentions; f) Preciseness; g) Reliability; and h) Expressive capability in communication.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Cognitive informatics

- **Cognitive informatics** (CI) is a transdisciplinary enquiry of natural and machine intelligence, and their products in terms of information, knowledge, and behaviors.

    - Cognitive informatics covers a whole range of interdisciplinary research in subject areas including **natural intelligence** (NI), **autonomic computing** (AC), and **neural informatics** (NeI).

    - The theories of cognitive informatics and neural informatics explain a number of important questions in the study of NI. Enlightening results derived in cognitive informatics have led to the determination of the magnitude and expected **capacity of human memory.**

• **Neural Informatics** (NeI) is a new interdisciplinary enquiry of the biological and physiological representation of information and knowledge in the brain at the neuron level and their abstract mathematical models.

• Neural informatics is a branch of cognitive informatics, where memory is recognized as the foundation and platform of any natural or artificial intelligence.

Cognitive informatics models of the brain

• **The human brain** is the most complicated organ in the universe and is constantly the frontier yet to be explored in an interdisciplinary approach. Investigation into the brain and its cognitive mechanism is a unique and the hardest problem in science that requires **recursive** and **introspective** mental power to explore the brain by the brain.

• **The Cognitive Models of Memory (CMM):** CMM states that the architecture of human memory is parallel configured by the Sensory Buffer Memory (SBM), Short-Term Memory (STM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM).

• **The Layered Reference Model of the Brain (LRMB):** LRMB encompasses 39 cognitive processes at six layers known as the *sensation*, *memory*, *perception*, *action*, *meta cognitive*, and *higher cognitive* layers from the bottom up.

• The hierarchical life functions of the brain, the **natural intelligent system** (NI-Sys), can be divided into two categories: the **subconscious and conscious subsystems**. The former known as the **NI operating system** (NI-OS) encompasses the layers of sensation, memory, perception, and action (Layers 1 to 4). The latter known as the **NI applications** (NI-App) includes the layers of meta and higher cognitive functions (Layers 5 and 6).

• **Sensation** is a set of cognitive processes of the brain at the subconscious life functional layer that forms the interfaces between the internal and external worlds for information detection, acquisition, and input into the brain. The **sensation layer** of LRMB is a subconscious layer of life functions of the brain for detecting and acquiring cognitive information from the external world via physical and/or chemical means.

- **Memory** is a set of cognitive processes of the brain at the subconscious life function layer that retains the external or internal cognitive information in various memories of the brain, particularly in LTM. The **memory layer** of LRMB is the fundamental layer of life functions of the brain functioning to: a) retain and store information about both the external and internal worlds; b) maintain a stable state of an animate system; c) provide a working space of abstract inference; and d) buffer programmed actions and motions to be executed by the body.

- **Perception** is a set of internal sensational cognitive processes of the brain at the subconscious life function layer that detects, relates, interprets, and searches internal cognitive information in the mind. The **perception layer** of LRMB is a subconscious layer of life functions of the brain for maintaining conscious life functions and for browsing internal abstract memories in the cognitive models of the brain.

- **Action** is a set of subconscious cognitive processes of the brain at the subconscious life function that executes both bodily (external) or mental (internal) actions via the motor systems of the body or the perceptional engine of the brain. The ***action layer*** of LRMB is a subconscious layer of life functions of the brain for output-oriented actions and motions that implement human behaviors such as a sequence of movement and a pre-prepared verbal sentence.

- A **meta cognitive function** is a fundamental and elemental cognitive process of the brain at the conscious life function layer that is commonly used (or applied) to support the higher layer cognitive life functions. The **meta cognitive process layer** of LRMB is a conscious layer of life functions of the brain that carries out the fundamental and elementary cognitive processes commonly used in higher cognitive processes.

- A **higher cognitive function** is an advanced cognitive process of the brain at the conscious life function layer that is developed and acquired to carry out commonly recurring life functions under the support of the meta cognitive process. The **higher cognitive process layer** of LRMB is a conscious layer of life functions of the brain that carries out a set of specific cognitive processes under the support of the meta cognitive processes.

- **The Cognitive Information Model (CIM):** CIM classifies cognitive information into four categories, according to their types of I/O information, known as *knowledge* (K: $I \to I$), *behavior* (B: $I \to A$)*, experience* (E: $A \to I$)*,* and *skill* (S: $A \to A$), where I stands for information and A for actions.

- The CIM model lays an important foundation for learning theories and pedagogy. It reveals that **software engineering deals with instructive behaviors** and their relations with knowledge, experience, and skills.

- **The generic forms of cognitive information:** There are four categories of internal information $\mathcal{I}$ in the brain known as *knowledge* ($\mathcal{I}_k$), *behaviors* ($\mathcal{I}_b$), *experience* ($\mathcal{I}_e$), and *skills* ($\mathcal{I}_s$), i.e., $\mathcal{I} = (\mathcal{I}_k, \mathcal{I}_b, \mathcal{I}_e, \mathcal{I}_s)$. All the four categories of information can be acquired directly by an individual. However, knowledge and behaviors can be learnt indirectly based on abstract information. Instead, experience and skills must be learnt directly by hands-on or empirical actions.

- **Intelligence: Natural intelligence** (NI) is a system of intelligent behaviors possessed or implemented by the brains of human beings and other advanced species; **Artificial intelligence** (AI) is a system of intelligent behaviors possessed or implemented by machines or man-made systems.

- **Measurement of Intelligence:** There are three methods for measuring the natural and artificial intelligence.

  - **Intelligent quotient** (IQ) is a ratio between the mental age A_m and the chronological (actual) age A_c, multiplied by 100, where A_m is the sum of a base age A_b and an extra equivalent age ΔA.

  - **Turing intelligence equivalence** E_T is a ratio of conformance or equivalence evaluated in a comparative test between a system under test and an equivalent human-based system, where both systems are treated as a black box and the testers do not know which the tested system is.

  - **Wang's intelligent capability metric** $\mathcal{C}_I$ is a normalized sum of abilities of the *perceptive intelligence* (C_p), *cognitive intelligence* (C_c), *instructive intelligence* (C_i), and *reflective intelligence* (C_r).

  - The *perceptive intelligent capability* C_p is the ability to transfer a given number of data objects or events N_d into a number of information objects in term of derived or related concepts N_i.

  - The *cognitive intelligent capability* C_c is the ability to transfer a given number of information objects N_i into a number of knowledge objects N_k.

- The *instructive intelligent capability* C_i is the ability to transfer a given number of information objects N_i into a number of behavioral actions N_b.

- The *reflective intelligent capability* C_r is the ability to transfer a given number of data objects or events N_d into a number of behavioral actions N_b.

- The **relative intelligent capability** $\Delta\mathfrak{C}_I$ is the difference between a testee's absolute intelligent capability $\mathfrak{C}_I$ and a given intelligent capability benchmark $\overline{\mathfrak{C}_I}$.

- **Theory of learning:** The *generic forms of learning* $\mathcal{L}$ can be classified into those of *knowledge* ($\mathcal{L}_\mathrm{k}$)*, behaviors* ($\mathcal{L}_\mathrm{b}$)*, experience* ($\mathcal{L}_\mathrm{e}$)*,* and *skills* ($\mathcal{L}_\mathrm{s}$)*, i.e.,* $\mathcal{L} = (\mathcal{L}_k, \mathcal{L}_b, \mathcal{L}_e, \mathcal{L}_s)$.

- **The functional model of the brain:** A high-level logical model of the brain describes the functional configuration of the brain and how the *NI-Sys* interacts with the memory system. It revealed that intelligence is *memory-based*.

Cognitive informatics models of knowledge representation

- **The cognitive informatics model for internal knowledge representation:** The functional model of LTM is a set of *Hierarchical Neural Clusters* (HNC) with partially connected *neurons* via *synapses*. In contrary to the traditional **container** metaphor, the human memory mechanism can be described by a **relational** metaphor, in which memory and knowledge are represented by the connections between neurons in the brain, rather than the neurons themselves as information containers.

- **The *Object-Attribute-Relation* (OAR) model:** The OAR model of LTM can be described as a triple, i.e., *OAR* = (*O, A, R*), where *O* is a set of objects identified by unique symbolic names, *A* is a set of attributes for characterizing the object, and *R* is a set of relations between o_i and other objects or attributes of other objects.

- **The *Extended OAR Model* (EOAR) of the brain:** *EOAR* states that the external world is represented by *real entities*, and the internal world by *virtual entities* and *objects*. The internal world can be divided into two layers known as the *image layer* and the *abstract layer*.

- The EOAR model can be used to describe information representation and its relation to the external world. It can also be applied to explain the mental processes and cognitive mechanisms as identified in the LRMB model.

- The **principal intelligent advantages** state that, on the basis of two principal advantages known as the *qualitative* properties and *quantitative* properties, humans gain the power as the most intelligent species in the world.

Cognitive informatics for software engineering

- **Productivity** of software development is the key among all the cognitive, time, and resources constraints in software engineering. The other constraints can be overcome as a result of the improvement of software engineering productivity. All theories and approaches explored throughout this book put emphases on this **ultimate goal of software engineering**.

- The **key approaches to improve software development productivity** are:

  a) To explicitly express software architectures and behaviors in *denotational mathematics*;

  b) To investigate the theories of rational software *engineering organization*; and

  c) To design tools that lead to *automatic software code generation* based on the denotational system models.

- **Software engineering psychology:** There are two types of programmers: the *realistic* and *idealistic* ones. The former may be suitable for coding, testing, and quality control; while the latter may be good at solution seeking, GUI design, and carrying out tasks as system analysts.

- **Psychological requirements for software engineers**: abstract-level thinking; imagination of static descriptions in terms of dynamic behaviors; organization capability; cooperation and team working attitude; long-term focus of attentions; preciseness; reliability; and expressive capability in communications.

- **Program comprehension:** It is a cognitive process to understand a given software system in dimensions of architecture, static behaviors, and dynamic behaviors, and their relationships.

- The **cognitive process of comprehension**: a) To search relations from *real entities* to *virtual entities* and/or existing *objects* and *attributes*; b) To build a partial or adequate *OAR* model of the entity; c) To wrap up the OAR model by classifying and connecting it to appropriate clusters of the LTM; d) To memorize the new OAR model and its connections in LTM.

- **Software engineering skills and experiences:** All software engineering experience and skills, such as best practices, know-how, lessons learnt, failure reports, and new technology trial results, are hard to gain indirectly, because: a) The brain has no direct experience or skill transfer mechanism, hence it has to be physiologically acquired from the external world; b) The only way to gain experience is *learn by doing*, or *trial and error*. Hence people have to make the same mistakes, at least to simulate them, in order to learn and remember them; c) Each brain is unique because of individual physiological differences, cognitive style differences, personality differences, and environment differences.

- A **software agent** is an intelligent software system that autonomously carries out robotistic and interactive applications based on goal-driven mechanisms. The theoretical foundation of agent systems is rooted in cognitive informatics theories and autonomic computing methodologies, particularly the LRMB model and denotational mathematics.

  - Along with the latest development of cognitive informatics and autonomic computing, self-organizing, self-managing and non-imperative autonomic agent systems are emerging. An **autonomic agent system** is an intelligent software system that takes rational actions in the pursuit of its agenda via goal-, inference-, and event-driven behaviors.

Cognitive complexity of software

- **The cognitive complexity weights of software:** One of the central problems in software engineering is the inherited complexity.

  - The **cognitive weight of software** is the extent of difficulty or relative time and effort for comprehending the function and semantics of a given program.

  - It is almost impossible to measure the cognitive weights of a program at statement level because of their variety and language dependency. However, it is found that it is feasible if the focus is put on the **BCS's**, the control logic of software systems, because there are only ten BCS's in programming no matter what kind of programming language is used.

- Software cognitive complexity provides a foundation for cross-platform analysis of complexities, sizes, and comprehension efforts of software specifications and implementations in the phases of design, implementation, and/or maintenance in software engineering. Detailed theories will be completed in Section 10.7.

Questions and Research Opportunities

9.1 What is cognitive informatics? How has it emerged from classic and contemporary informatics – the science of information?

9.2 What are the relationships of cognitive informatics with NI, AI, and cognitive science?

9.3 Summarize the fundamental theories of cognitive informatics, particularly the laws and principles for software engineering stated in the theorems of this chapter.

9.4 On the basis of the Software Engineering Constraint Model (SECM, Theorem 1.5), discuss why cognitive informatics plays an important role in building the foundations for software engineering.

9.5 Why is denotational mathematics a coherent part of the theoretical framework of cognitive informatics?

9.6 Discuss how cognitive informatics may be used in explaining fundamental software engineering issues such as software comprehension and software experience/skill transformation.

9.7 What are the qualitative and quantitative advantages of the human brain with regard to the other species?

9.8 Explain the roles of synapses in the neural networks of the brain according to the *Hierarchical Neural Clusters* (HNC) model and the logical model of OAR.

9.9 According to NeI and the OAR model, try to logically explain why brain neurons are the only type of cells in human body that does not go through reproduction rather than remains alive throughout the entire human life.

9.10 On the basis of the OAR model, try to explain the physiological and logical mechanisms of creation or invention.

9.11 Why acquired information and knowledge cannot be passed on and inherited through genes? How may highly professional skills that are acquired throughout a person's life be passed on to peers or next generation?

9.12 According to the HNC model and the properties of LTM, discuss why human memory can be searched based on content-sensitive mechanisms, but cannot be sorted internally.

9.13 Which form of *cognitive information* is with inputs of data and outputs of actions that can be both directly and indirectly acquired?

9.14 What are the forms of intelligence? Are the natural and machine intelligence different? Why?

9.15 Why is memory the foundation of both the natural and machine intelligence? Can a form of intelligence exist without memory? Why?

9.16 What is the taxonomy of human memories? Why should the action buffer memory be considered as an independent category of memory? Without it what kind of life functions cannot be carried out?

9.17 According to the *Layered Reference Model of the Brain* (LRMB), why can the natural intelligence be reduced and described by 6 layers and 39 fundamental cognitive processes?

9.18 Given a goal to find a certain restaurant, explain how the task may be carried out by composing multiple LRMB processes at different layers. A block diagram may be used to represent your answer.

9.19 According to the LRMB model, try to suggest any possible basic process(es) that is not modeled in LRMB and cannot be composedly described by multiple fundamental processes identified in it.

9.20 Define the four categories of information one deals with in software engineering according to Theorem 9.4 and the *Cognitive Information Model* (CIM).

9.21 The CIM model classifies cognitive information into four categories, according to their types of I/O information, known as knowledge (*information, information*), behavior (*information, action*), experience (*action, information*), and skill (*action, action*).

According to CIM, what is the category of a program or a software system? What is the category of a programmer's acquired ability for programming?

9.22 Why may the functional model of the brain be treated as a real-time intelligent system?

9.23 Discuss what the fundamental mechanisms of natural intelligence of the brain are, and how internal information is represented, processed, and utilized.

9.24 What will be the next generation architectures of computers that may learn from the human brains and natural intelligence?

9.25 Discuss what can a computer do while human beings cannot? What can a computer do better than human beings?

9.26 How do a faculty of subconscious and conscious life functions interact in the brain?

9.27 Where is the thinking engine? Or where is the organ in the brain that physiologically drives thinking and perception?

9.28 How is the thinking engine triggered or directed?

9.29 Are all thinking mechanisms consciously or intentionally controllable?

9.30 How can consciousness be the product of physiological and mental processes of the brain?

9.31 Explain why experience transfer is very difficult in programming and software engineering on the basis of Corollary 9.2.

9.32 According to Theorem 9.5 and the GIM model, explain why programming may be considered, to some extent, as a process to create machine intelligence.

9.33 Why do all mammals need sleep? What is the cognitive mechanism of sleeping?

9.34 When is memory established in the long-term memory? What are the roles of sleep in memory creation? What does the 24-hour law (Theorem 9.11) explain?

9.35 According to the OAR model, the entire knowledge maintained and represented in the brain is a hierarchical OAR structure. Try to use the OAR model to explain the following mechanisms:

(a) How is existing knowledge extended or updated in the OAR with A_{ext}, O_{ext}, and/or R_{ext}?

(b) How is new knowledge created in the OAR with A_{new}, O_{new}, and/or R_{new}?

9.36 Why is software productivity, as that of any other creative work, is conservative that forms a basic constraint for software engineering?

9.37 Why has software productivity not been significantly improved in software engineering in the last four decades? What are key approaches to improve software development productivity by dealing with the cognitive complexity of software engineering?

9.38 According to the cognitive model of the comprehension process, explain why no comprehension is logically a comprehension.

9.39 Referring to Experiment 9.3, try to design an psychological experiment for calibrating your relative cognitive weight of the while-loop BCS, i.e., $w_{BCS}(4)$. Then, compare your result with Table 9.9.

9.40 According to the experiment method provided in Definition 9.38 and the detail test programs reported in [Wang, 2005j], try to conduct a complete set of the psychological experiments for calibrating the relative cognitive weights of all ten BCS's in a group and analyze your results with Table 9.9.

9.41 Read the following classic article in software engineering:

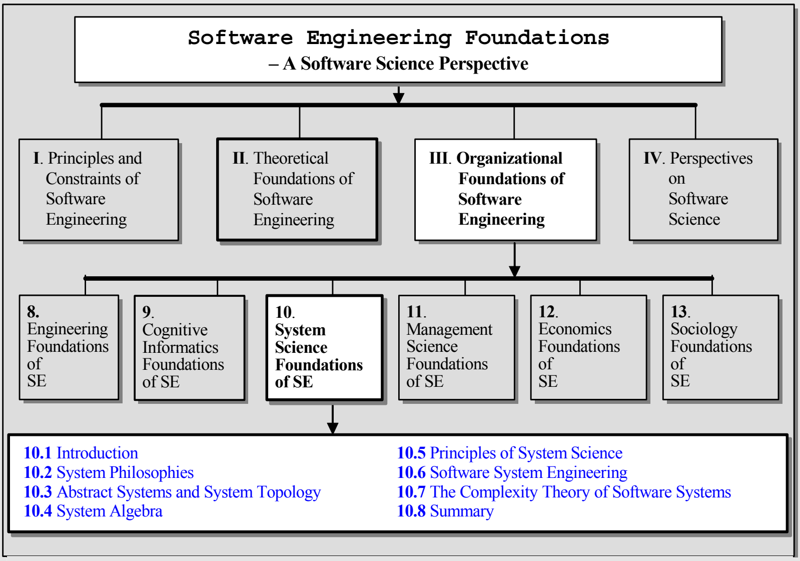
> John L. McCarthy (1987), Generality in Artificial Intelligence, The 1971 Turing Award Lecture, *Communications of the ACM*, 30(12), pp. 1029-1035.

Discuss the following topics in a group:

- About the author.
- What was the generality of AI according to the author in the 1970s?
- What is the relationship between AI and cognitive informatics (CI)?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 10

SYSTEM SCIENCE FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE

10. System Science Foundations of SE

Knowledge Structure

❍ System philosophies

- The system metaphor for modeling complex entities
- Holism
- Systematic thinking

❍ Abstract systems and system topology

- Mathematical models of abstract systems
- Taxonomy of systems
- Magnitudes of systems

❍ System algebra

- Algebraic relations of systems
- Algebraic operations of systems

❍ Principles of system science

- System fusions
- Work done by systems
- System equilibrium and organization
- System dissimilation
- System functions and behaviors
- The maximum output of systems
- System synchronization and coordination

❍ Software system engineering

- The abstract model of computing systems
- The ISO/IEC 15288 system engineering model for SE
- SE phenomena as system engineering problems
- The hierarchical model of software systems

❍ The complexity theory of software systems

- Computational complexity
- Cognitive complexity of software systems
- Cohesion and coupling complexity of software systems
- Control flow complexity
- Software cognitive complexity analysis

Learning Objectives

- To know the system philosophy for dealing with complex entities and objects in software engineering.
- To understand the concept of abstract systems and their topology as a mathematical model for general and concrete systems.
- To understand the new mathematical structure of system algebra for modeling and manipulating abstract and software systems.
- To gain knowledge on fundamental system principles and their formal explanations, particularly system gains and abstract work done by systems.
- To understand the abstract model of generic computing systems and the hierarchical model of generic software systems.
- To understand the complexity theory of software systems, particularly cognitive complexity analyses for software engineering.
- To be able to explain software engineering phenomena by system theories.

"Problems that are created by our current level of thinking can't be solved by that same level of thinking."

Albert Einstein (1879-1956)

"*The more science becomes divided into specialized disciplines, the more important it becomes to find unifying principles.*"

Herman Haken (1977)

10.1 Introduction

System theories are one of the three emerging sciences, with information science (covered in Chapter 7) and cybernetics (covered and extended in Chapter 9), developed in the 1940s. System science and its application in systems engineering are a branch of knowledge that studies the top-level objects and phenomena in the physical, information, and social worlds, namely *systems*, across all science and engineering disciplines.

The concept of systems can be traced back to the 17th Century when Rene Descartes (1596-1650) noticed the interrelationship among scientific disciplines as systems. Descartes developed analytic geometry that integrated algebra, geometry, and physics on the same mathematical foundation. Then, the general system notion was founded by Ludwig von Bertalanffy in the 1920s [von Bertalanffy, 1952].

Definition 10.1 An *abstract system* is a collection of coherent and interactive entities that possesses stable functions and clear boundary with external environment.

This chapter treats systems as discrete entities and studies the generic rules and theories of abstract systems. Systems treated as continuous systems may be referred to *synergetics* [Haken, 1977/83; Haken et al., 1995], *hypercycle* theory [Eigen and Schuster, 1979], and *dissipative structures* [Prigogine and Stengers, 1984/97].

The classic theories at the system level developed in the 20th century are as follows:

- *Information theory* (C.E. Shannon, 1948)
- *Cybernetics* (N. Wiener, 1948)
- *Systems theory* (L. von Bertalanffy, 1952)

With a similar point of view, management science, economics, and sociology, which will be explored in Chapters 11, 12, and 13, respectively, are special branches of system science that study objects and phenomena at different levels of human coordinative work and social organizations.

Because software engineering is one of the most complicated systems that humans have ever dealt with, it is naturally an ideal testbed for evaluating existing system theories and their enhancements. Treating software engineering and large-scale software project via system engineering is also a promising trend in dealing with the problems, complexities, quality assurances, and human factors in software engineering.

This chapter describes the system metaphor of software and software engineering. It explores theories of systems science, as well as underlying principles and modeling techniques of system engineering. Applications of system theories and system engineering methodologies in software engineering are discussed, and interesting software engineering phenomena as system engineering issues are addressed. In the remainder of this chapter, the system science foundations of software engineering will be presented in six sections. Section 10.2 presents classic system philosophies and the ways of system thinking for modeling complex entities. Section 10.3 introduces a new mathematical structure of abstract systems known as system algebra. Section 10.4 describes principles of system theories with a formal and rigorous treatment. Section 10.5 discusses properties of abstract systems such as generic architectures, equilibrium, synchronization, and dissimilation. Section 10.6 applies system theories and system engineering techniques into software engineering. The systematical perception on software and software system models is presented in Section 10.7.

10.2 System Philosophies

Many preeminent scientists working in different disciplines intend to recognize that nature is a coherent system with perfect *harmony* and *integrity* [Ellis and Fred, 1962]. Rene Descartes (1596-1650) first proposed the *notion of system* in his investigation into analytic geometry. Ludwig von Bertalanffy found the *generic system theory* on the basis of his study in biologic systems [von Bertalanffy, 1952], which is then extensively studied in [Hall and Fagan, 1956; Boulding, 1956/74; Ashby, 1956/58a/62/70/72; Rapoport, 1962; Schedrovitzk, 1962; Makridakis and Faucheux, 1971/73; Bunge, 1978/81; Gaines, 1972/78/84; Takahara and Takai, 1985; Klir, 2001].

Steven Weinberg wrote: ‘Our job in physics is to see things simply, to understand a great many complicated phenomena, in terms of a few simple principles.’ The system philosophy is based on the observation that the nature is built by a small number of basic components and particles, and governed by a limited set of basic laws. Even all living things are configured by almost the same cells, chromosomes, and DNAs.

In *Philosophy of Physics* (1936), Max Planck expressed:

> “Modern physics has taught us that the nature of any system cannot be discovered by dividing it into its component parts and studying each part by itself, since such a method often implies the *loss of important properties* of the system. We must keep our attention fixed on the whole and on the inter-connection between the parts. … The whole is never equal simply to the sum of its various parts.”

This is an excellent representation of the basic philosophy of system science theory, which suggests that systems must be viewed as a whole with each part linking to all the other parts.

10.2.1 THE SYSTEM METAPHOR FOR MODELING COMPLEX ENTITIES

The discipline of system science is an inquiry into the general principles and rules commonly shared by different kinds of systems. The system metaphor is one of the most widely used concepts and notions in almost all disciplines of science, engineering, and society. Whenever an object of study is getting complicated, the system metaphor can be adopted as a powerful modeling and analysis means to deal with the complexity of the object under investigation.

A system can be as small as two dependent components or as large as the universe. The scopes and magnitudes of systems may vary extremely from a few components to billions of components. According to the system philosophy, the *universe* may be defined as a system of all systems. Otherwise, the readers might have difficulties to answer: Where is the boundary of the universe? What are things outside the universe?

Lemma 10.1 The principle of *generic constraints* states that any system is constrained by a set of common conditions, properties, and rules, which are obeyed by all components inside the system, but not by those outside it.

The generic system theory treats everything as a system, and it perceives that a system belongs to other super system(s) and contains more subsystems. A generic system can be described recursively in a hierarchical structure as illustrated in Fig. 10.1

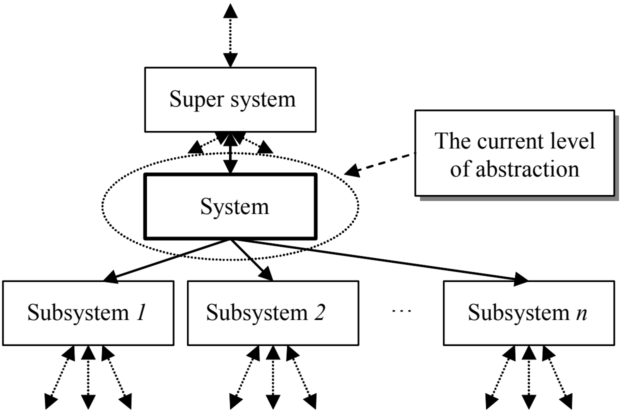


Figure 10.1 A hierarchical view of system structures

For example, the hierarchical levels of living systems of the universe can be decomposed into the following layers from bottom up:

- Cell
- Organ
- Organism / individual
- Group / team
- Organization
- Society / community
- Supranational system / earth
- The universe

Based on Fig. 10.1, the following lemma on system abstraction can be derived.

Lemma 10.2 The *generality principle of system abstraction* states that a system can be represented as a whole in a given level *k* of abstraction and reasoning without knowing the details at levels below *k*.

10.2.2 HOLISM

The word *holism* is originated from Greek *holos* meaning the whole. Holism is a philosophical view that perceives a phenomenon and system with wholeness in an integrated, synthetic, and systematic approach. Holism was well entrenched in Greek philosophy as well as in the classic Chinese philosophy of Taoism.

The most influential statement of holism is as follows:

> "The whole is more than the sum of its parts."

That has been widely accepted since Aristotle's time in 400BC [Klir, 2001]. According to holism, complex organisms and systems as a whole possess special properties when its elements and their interactions reach or beyond a certain critical mass, which cannot be found from any of the individual elements.

Albert Einstein (1879-1956) once pointed out:

> "Problems that are created by our current level of thinking cannot be solved by that same level of thinking."

Bertrand Russell, British philosopher and logician, presented similar thought in the 1960s, concerning that problems about the whole of a given order of logic have to be solved in a higher order of logic. Russell also proposed for using objects and classes to describe the logical world in 1900 [Russell, 1961], which is considered the philosophical foundation of modern object-oriented technology in software engineering.

Taking the systematic view on entities and their relations in a system as a whole, a common problem known as "local maximum" may be avoided in the development of knowledge. As it is said there is no number two in *sciences*, while there is no number one in *engineering*. The rational is that sciences do not recognize a repetitive discovery or reinvention, but engineering cannot prove or claim, in technical and economical terms, whether a specific design or implementation is the best or the optimal solution among a large number of potential ones. This is particularly true in software engineering.

10.2.3 SYSTEMATIC THINKING

The principle of systematic thinking extends the system philosophy to strategic and tactic applications in sociology, management science, economics, engineering, and everyday life. To illustrate this philosophy, the following historic story about King Qi's horse racing may be taken as an example. About 2000 years ago, King Qi collected the best horses in his kingdom. He liked horse racing very much and he expected to win every time. However, once he lost to Ji Tian, a wizard of that time.

The horses were categorized in three classes, i.e., for the King: K_1, K_2, and K_3; and for Tian: T_1, T_2, and T_3. If they were raced in the way: $K_1 – T_1$, $K_2 – T_2$, and $K_3 – T_3$, there was no surprise that the King had have to win, as illustrated in Fig. 10.2, because he possessed the best horses in each class.

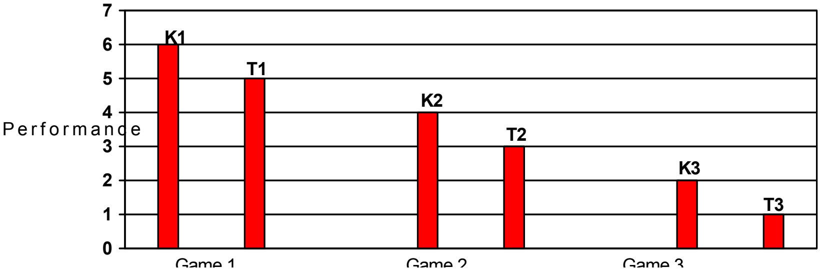


Figure 10.2 Horse Racing: King Qi vs. Ji Tian – System strategy (I)

However, the wizard Ji Tian won using the following systematical strategy as shown in Fig. 10.3. Tian dispatched his third class horse (T_3) against King Qi's first class (K_1), and of course allowed the King to win the first race. Then, in the following two races, Tian used his first (T_1) and second (K_2) class horses against the King's second (K_2) and third (K_3) class horses, respectively. Eventually, Tian won the three-match game for the first time in the history of the kingdom.

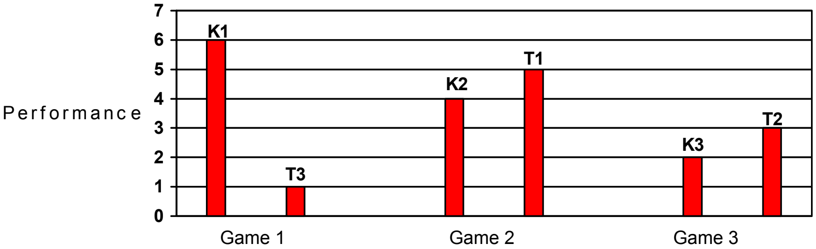


Figure 10.3 Horse Racing: King Qi vs. Ji Tian – System strategy (II)

This story tells a useful operational strategy in system theory, which has then been taken as an excellent paradigm of systematic thinking. However, readers will find that the contemporary system theories in system science and engineering are far more complicated and abstract than the story.

10.3 Abstract Systems and System Topology

This section creates the mathematical models of abstract systems and explores their properties, structures, and behaviors. The topological properties of abstract systems universally shared by all systems, such as system sizes, magnitudes, and complexities, are analyzed. The structural properties of abstract systems are studied on the hierarchical architectures. As a result, the universal system organization tree is introduced.

10.3.1 MATHEMATICAL MODELS OF ABSTRACT SYSTEMS

This subsection introduces the concept of abstract systems [Wang, 2006d] and describes how the properties of abstract systems may be formally modeled and studied.

Abstract systems can be classified into two categories known as the closed and open systems. Most practical and useful systems in nature are open systems in which there are interactions between the system and its environment. However, in order to develop the theoretical framework of abstract systems, the closed systems in which there is no interaction with external environment will be introduced first in the following subsection.

10.3.1.1 The Mathematical Model of Closed Systems

The axiom of the abstract system theory is based on the OAR model [Wang and Wang, 2006; Wang, 2007g] as defined in Chapter 9, in which the architecture of a system object O_s can be modeled by a set of attributes A and a set of binary relations R among the attributes, i.e.:

$$O_s = (A, R) \tag{10.1}$$

Encompassing both architectures and behaviors of a system on the basis of Eq. 10.1, an abstract closed system without interactions with the environment can be formally described as follows.

Definition 10.2 A *closed system* $\widehat{S}$ is a 4-tuple, i.e.:

$$\widehat{S} = (C, R, B, \Omega) \tag{10.2}$$

where

- C is a nonempty set of components of the system, $C = \{c_1, c_2, \ldots, c_n\}$.
- R is a nonempty set of relations between pairs of the components in the system, $R = \{r_1, r_2, \ldots, r_m\}$, $R \subseteq C \times C$.
- B is a set of behaviors (or functions), $B = \{b_1, b_2, \ldots, b_p\}$.
- Ω is a set of constraints on the memberships of components, the conditions of relations, and the scopes of behaviors, $\Omega = \{\omega_1, \omega_2, \ldots, \omega_q\}$.

An abstract closed system, $\widehat{S} = (C, R, B, \Omega)$, can be illustrated as shown in Fig. 10.4.

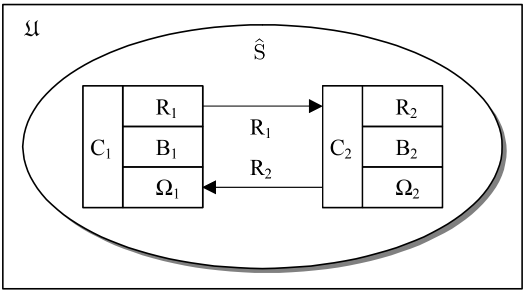


Figure 10.4 The abstract model of a closed system

Lemma 10.3 A closed system $\widehat{S} = (C, R, B, \Omega)$ is an *asymmetric* (directed) and *reflective* system because the relations R in it are constrained by the following rules:

$$\text{(a) } \forall a, b \in C \wedge a \neq b \wedge r \in R,\ r(a, b) \not\Rightarrow r(b, a) \tag{10.3}$$

$$\text{(b) } \forall c \in C,\ r(c, c) \in R \tag{10.4}$$

Definition 10.3 The *maximum number of binary relations* $n_r = \#R$ between all pairs of the $n_c = \#C$ components in a closed system $\widehat{S} = (C, R, B, \Omega)$ can be determined as follows:

$$\begin{aligned} n_r &= \#R \\ &= \#(C \times C) \\ &= n_c^{\ 2} \end{aligned} \tag{10.5}$$

if all reflective self-relations are ruled out in n_r, the partially connected relations n'_r is obtain as:

$$n'_r = n_c(n_c - 1) \tag{10.6}$$

10.3.1.2 The Mathematical Model of Open Systems

Most practical systems in the real world are not closed. That is, they need to interact with the external world known as the *environment* Θ in order to exchange energy, matter, and/or information. Such systems are called open systems. Typical interactions between an open system and the environment are inputs and outputs.

Observe that the relations of a closed system are defined on the Cartesian product of internal components. For an open system that has interactions with external environment, the set of relations R needs to be extended to include both internal relations R^c and external (input/output) relations R^i and R^o, i.e.:

$$R = R^c \cup R^i \cup R^o \tag{10.7}$$

Based on the above discussion, an abstract open system can be defined below.

Definition 10.4 An *open system S* is a 7-tuple, i.e.:

$$\begin{aligned} S &= (C, R, B, \Omega, \Theta) \\ &= (C, R^c, R^i, R^o, B, \Omega, \Theta) \end{aligned} \tag{10.8}$$

where the extensions of entities beyond the closed system are as follows:

- Θ is the environment of S with a nonempty set of components C_Θ outside C, i.e., $C_\Theta \cap C = \varnothing$.
- $R^c \subseteq C \times C$ is a set of internal relations.
- $R^i \subseteq C_\Theta \times C$ is a set of external input relations.
- $R^o \subseteq C \times C_\Theta$ is a set of external output relations.

An abstract open system, $S = (C, R^c, R^i, R^o, B, \Omega, \Theta)$, can be illustrated as shown in Fig. 10.5.

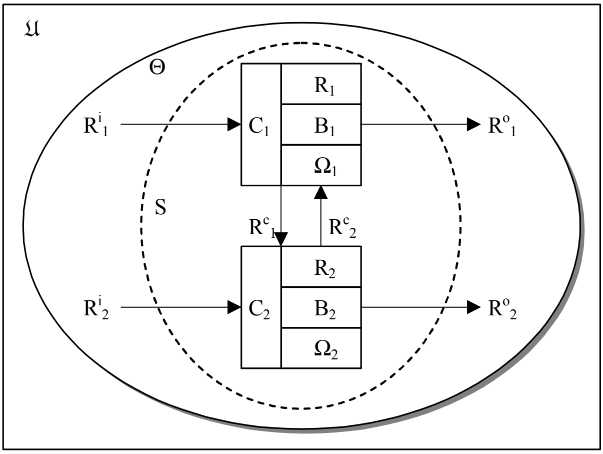


Figure 10.5 The abstract model of an open system

Example 10.1 A digital clock, *Clock*, can be described as an open system S_1 as follows:

$$Clock = S_1(C_1, R^c_1, R^i_1, R^o_1, B_1, \Omega_1, \Theta_1)$$

where

- The set of components:
  $C_1 = \{Processor, Keypad, LEDs, ClockPulse\}$

- The set of internal relations:
  $R^c_1 \subseteq C_1 \times C_1 = \{Input\ (Keypad, Processor),$
  $Tick\ (ClockPulse, Processor),$
  $Output\ (Processor, LEDs)\}$

- The set of input relations:
  $R^i_1 \subseteq C_{\Theta 1} \times C_1 = \{SetTime\ (User, Keypad)\}$

- The set of output relations:
  $R^o_1 \subseteq C_1 \times C_{\Theta 1} = \{ShowTime\ (LEDs, User)\}$

- The set of behaviors:
  $B_1 = \{SetTime, ShownTime, Tick\}$

- The set of constraints:
  $\Omega_1 = \{Time = \text{hh} \times \text{mm} \times \text{ss}\}$

- The environment:
  Θ_1 = {User}

where the behaviors of the digital clock system defined in B_1 can be further refined by a set of processes in RTPA.

Example 10.2 The alarm subsystem, *Alarm*, of a digital clock can be described as an *open system* S_2 as follows:

$$Alarm = S_2(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2)$$

where

- The set of components:
  $C_2 = \{Processor,\ Keypad,\ LEDs,\ Bell\}$

- The set of internal relations:
  $R^c{}_2 \subseteq C_2 \times C_2 = \{Input\ (Keypad,\ Processor),$
  $AlarmCheck\ (Time,\ Alarm),$
  $AlarmRelease\ (Keypad,\ Processor),$
  $Output\ (Processor,\ LEDs),$
  $Ring\ (Processor,\ Bell)\}$

- The set of input relations:
  $R^i{}_2 \subseteq C_{\Theta 2} \times C_2 = \{SetAlarm\ (User,\ Keypad)\}$

- The set of output relations:
  $R^o{}_2 \subseteq C_2 \times C_{\Theta 2} = \{ShowAlarm\ (LEDs,\ User)\}$

- The set of behaviors:
  $B_2 = \{SetAlarm,\ ShownAlarm,\ CheckAlarm,\ Ring,\ AlarmRelease\}$

- The set of constraints:
  $\Omega_2 = \{Alarm = \text{hh} \times \text{mm}\}$

- The environment:
  Θ_2 = {User}

where the behaviors of the alarm system defined in B_2 can be further refined by a set of processes in RTPA.

Lemma 10.4 An open system $S(C, R^c, R^i, R^o, B, \Omega, \Theta)$ is an *asymmetric* and *reflective* system because its relations R^c, R^i, and R^o are constrained by the following rules:

$$\text{(a) } \forall a, b \in C \wedge \mathrm{a} \neq \mathrm{b} \wedge r \in R^c, r(a,b) \not\Rightarrow r(b,a) \tag{10.9}$$

$$\text{(b) } \forall c \in C, r(c, c) \in R^c \tag{10.10}$$

$$\text{(c) } \forall a \in C \wedge \forall x \in C_\Theta \wedge r \in R^i, \ r(x,a) \not\Rightarrow r(a,x) \tag{10.11}$$

$$\text{(d) } \forall a \in C \wedge \forall x \in C_\Theta \wedge r \in R^o, \ r(a,x) \not\Rightarrow r(x,a) \tag{10.12}$$

Definition 10.5 The *total number of potential binary relations* n_r in an open system $S(C, R^c, R^i, R^o, B, \Omega, \Theta)$ is determined by the numbers of internal relations R^c, and external relations R^i and R^o, i.e.:

$$\begin{aligned} n_r &= \#R^c + \#R^i + \#R^o \\ &= n_c^{\,2} + 2\,\#C \bullet \#C_\Theta \end{aligned} \tag{10.13}$$

According to Definitions 10.3 and 10.4, it is apparent that either a closed or an open system may result in a huge number of relations n_r when the number of components possessed in them is considerably large.

10.3.2 TAXONOMY OF SYSTEMS

Systems as complex entities may be classified into various categories according to the key characteristics of their components (C), relations (R), behaviors (B), constraints (Ω), and/or environments (Θ). A summary of the system taxonomy is shown in Table 10.1.

There are combined categories of systems that fall in two or more categories, such as a dynamic nonlinear system and a discrete fuzzy social system. The types of systems may also classified by their magnitudes, which will be discussed in Section 10.3.3.

Detailed definitions of system classifications and their characteristics are described in the following subsections.

10.3.2.1 Concrete and Abstract Systems

Definition 10.6 A *concrete system* is a real and specific system with natural entities and certain functions.

Definition 10.7 An *abstract system* is a virtual or theoretical system that is modeled by mathematics or computing simulations.

Table 10.1
Taxonomy of Systems

No	**System**	**Key Characteristics**			
		Components (C)	**Relations (R)**	**Behaviors (B)**	**Environment (Θ)**
1	Concrete	Natural or real entities			
2	Abstract	Mathematical or virtual entities			
3	Physical	Natural entities			
4	Social	Humans			
5	Finite	$\#(C) \neq \infty$			
6	Infinite	$\#(C) = \infty$			
7	Closed		$R^i = \varnothing \wedge$ $R^o = \varnothing$		
8	Open		$R^i \neq \varnothing \wedge$ $R^o \neq \varnothing$		
9	Static			Invariable	
10	Dynamic			Variable	
11	Linear			Linear functions	
12	Nonlinear			Nonlinear functions	
13	Continuous			Continuous functions	
14	Discrete			Discrete functions	
15	Precise			Precise functions	
16	Fuzzy			Fuzzy functions	
17	Determinate			Response predictable to same stimulates	
18	Indeterminate			Response unpredictable to same stimulates	
19	White-box	Observable	Transparent	Observable	
20	Black-box	Unobservable	Non-transparent of internal relations	Observable	
21	Intelligent			Autonomic	Adaptive
22	Non-intelligent			Imperative	Nonadaptive
23	Maintainable	Fixable		Recoverable	
24	Non-maintainable	Nonfixable		Nonrecoverable	

An abstract system is usually used as a theoretical model of concrete systems, particularly for the purpose of formal treatment and the study of the generic properties shared by all concrete systems. In this view, a concrete system is an application or special case of the abstract system.

Observing the taxonomy of systems as given in Table 10.1, it can be seen that although there are numerous and various systems for different purposes in the real-world, their abstract models, categories, and properties are limited. This is the theoretical and empirical foundation for the establishment of system algebra as a generic mathematical means for the formal treatment of real-world systems as presented in Section 10.4 [Wang, 2006d].

10.3.2.2 Physical and Social Systems

Definition 10.8 A *physical system* is a natural and nonhuman system in which physical entities interact for certain purposes.

Definition 10.9 A *social system* is an organized human system in which groups of people interact for certain social purposes.

The physical systems can be mechanical, chemical, thermodynamic, electrical, and biological. The social systems such as social organizations, economic systems, and man-machine hybrid system are the most complicated and dynamic systems. The modeling and analysis of economical and social systems, as well as their applications in software engineering, will be presented in Chapters 12 through 13, respectively.

10.3.2.3 Finite and Infinite Systems

Definition 10.10 The finite system is a system with a certain number of components, i.e.:

$$n_c = \#\mathrm{C} < \infty \Rightarrow S(C, R^c, R^i, R^o, B, \Omega, \Theta) \text{ is } \textit{finite} \tag{10.14}$$

Definition 10.11 The infinite system is a system with an unlimited number of components, i.e.:

$$n_c = \#\mathrm{C} = \infty \Rightarrow S(C, R^c, R^i, R^o, B, \Omega, \Theta) \text{ is } \textit{infinite} \tag{10.15}$$

The ultimate concrete infinite system in the physical world is the universe, because as defined, anything that anybody may ever identify is included in the universal system.

Definition 10.12 The *universe* $\mathfrak{U}$ is an infinite system with unlimited sets of components U, as well as unlimited relations R_U, behaviors B_U, and constraints Ω_U, i.e.:

$$\mathfrak{U} = (U, R_{U,} B_U, \Omega_U) \tag{10.16}$$

where U encompasses any component c ever identifiable in the physical world, i.e., $\forall\, c, c \in U$.

There are many infinite *abstract* systems characterized by an infinite component set, such as the system with relations between a set of all natural numbers, or that of all points between [0, 1].

Lemma 10.5 There is only one *physical* infinitive system, i.e., $\mathfrak{U}$ (U, $R_{\mathrm{U},}$ B_{U}, Ω_{U}), but multiple abstract or mathematical infinitive systems exist.

Lemma 10.6 The evaluation *criterion* for whether a given system is *infinite* is that if the set of components C_{U} is infinitive.

Definition 10.13 The *empty system* $\mathfrak{O}$ is the smallest finite system in which the sets of components $C_\varnothing$, relations $R_\varnothing$, behaviors $B_\varnothing$, and constraints $\Omega_\varnothing$ are empty, i.e.:

$$\begin{aligned}\mathfrak{O} &= (C_\varnothing, R_{\varnothing,} B_\varnothing, \Omega_\varnothing)\\ &= (\varnothing, \varnothing, \varnothing, \varnothing)\end{aligned} \tag{10.17}$$

Definition 10.14 A *finite system* is a system between $\mathfrak{U}$(U, $R_{\mathrm{U},}$ B_{U}, Ω_{U}) and $\mathfrak{O}$($\varnothing$, $R_{\varnothing,}$ $B_\varnothing$, $\Omega_\varnothing$), which is characterized by $0 < \#C < \infty$.

Corollary 10.1 The evaluation *criterion* for whether a given system is *empty* is that if the set of components $C_\varnothing$ is empty.

Lemma 10.7 There is only one empty system in both the physical and abstract worlds, i.e., $\mathfrak{O}$($C_\varnothing$, $R_{\varnothing,}$ $B_\varnothing$, $\Omega_\varnothing$).

Corollary 10.2 The *relationship* between the infinitive universal system and the empty system $\mathfrak{O}$ is complementary, i.e.:

$$\overline{\mathfrak{U}} = \mathfrak{O} \tag{10.18a}$$

or

$$\overline{\mathfrak{O}} = \mathfrak{U} \tag{10.18b}$$

10.3.2.4 Closed and Open Systems

The definitions of closed and open systems have been given in Definitions 10.2 and 10.4. According to the definitions, it is hardly find any practical usage of a closed system except its theoretical value as a primitive model of systems.

Although in physics there are ideal theoretical models of closed systems, such as an isolated kinematical system and an adiabatic chamber with idea gas, there is no concrete closed system except the universe. Actually, in mathematical senses, there are only two concrete closed systems, the universal and empty systems, as described in the following theorem.

Corollary 10.3 There are only two concrete closed systems in the physical world that are the universal system $\mathfrak{U}(C_U, R_{U,} B_U, \Omega_U)$ and the empty system $\mathfrak{O}(C_\varnothing, R_{\varnothing,} B_\varnothing, \Omega_\varnothing)$.

The universal system $\mathfrak{U}(C_U, R_{U,} B_U, \Omega_U)$ is closed because there is no external environment outside $\mathfrak{U}$. The empty system $\mathfrak{O}(C_\varnothing, R_{\varnothing,} B_\varnothing, \Omega_\varnothing)$ is closed because there are no relations both internal and external.

Lemma 10.8 A closed system is *conservative* and there is always a unique static stable state.

Lemma 10.9 An open system is *anticonservative* and in which at least one dynamic equilibrium state exists.

10.3.2.5 Static and Dynamic Systems

Definition 10.15 A *static system* is a system that its behaviors are invariable over time or with the change of the environment.

Definition 10.16 A *dynamic system* is a system that its behaviors are variable over time or with the change of the environment.

10.3.2.6 Linear and Nonlinear Systems

Definition 10.17 A *linear system* is a system that its behaviors are modeled by linear functions.

Definition 10.18 A *nonlinear system* is a system that its behaviors are modeled by nonlinear functions.

10.3.2.7 Continuous and Discrete Systems

Definition 10.19 A *continuous system* is a system that its behaviors are modeled by continuous functions.

Definition 10.20 A *discrete system* is a system that its behaviors are modeled by discrete or digital functions.

Corollary 10.4 Continuous and discrete systems are equivalent because any continuous system can be simulated by a discrete system on the basis of behavioral equivalence.

10.3.2.8 Precise and Fuzzy Systems

Definition 10.21 A *precise system* is a system that its behaviors are modeled by precise functions.

Definition 10.22 A *fuzzy system* is a system that its behaviors are modeled by fuzzy logic.

Some works in fuzzy theories prefer to call a precise system as a *crisp* system. From a point of view in the time dimension, precise verses fuzzy systems may better represent the cognitive or theoretical maturity towards a target system under study. That is, in the beginning, a system under study may be fuzzy, but in the end, as understanding improves along theoretical and technical advances, it becomes precise. Also, there exists a unified membership function in extended set theory [Wang, 2007a] that treats the precise, fuzzy, and rough sets and their membership functions as its special cases.

10.3.2.9 Determinate and Indeterminate Systems

Definition 10.23 A *determinate system* is a system that its behaviors are predictable for the same stimulus from the environment.

Definition 10.24 An *indeterminate system* is a system that its behaviors are unpredictable for the same stimulus from the environment due to internal states, memory, and long-term feedback.

10.3.2.10 White-Box and Black-Box Systems

Definition 10.25 A *white-box system* is a system that both its internal architectures including the components and their relations, and external behaviors are transparent to and observable from the environment.

Definition 10.26 A *black-box system* is a system that only a part of its behaviors are observable from the environment, but the components and their relations are not transparent.

10.3.2.11 Intelligent and Nonintelligent Systems

Definition 10.27 An *intelligent system* is a system that its behaviors are determined autonomously by internal goals and motivations, and it is adaptive to the environment through learning and cumulated knowledge.

Definition 10.28 A *nonintelligent system* is a system that its behaviors are determined by imperative instructions, and has no ability to adapt to the environment.

10.3.2.12 Maintainable and Nonmaintainable Systems

Definition 10.29 A *maintainable system* is a system that its conditions, functions, or performance can be recovered or resumed after malfunctions or unsatisfied performances by maintenance or service during its lifecycle.

Definition 10.30 A *non-maintainable system* is a system that its conditions, functions, or performance cannot be recovered or resumed due to malfunctions and degrading, or its maintenance and service are infeasible technically or economically, during its lifecycle.

10.3.3 MAGNITUDES OF SYSTEMS

Abstract and real-world systems may be very small or extremely large [Rosen, 1977; Qian et al, 1990]. Therefore, a formal model of system magnitudes is needed to classify the size properties of systems and their relationship with other basic system attributes. In order to derive such a model, a set of measures on system sizes, magnitudes, and complexities is introduced in this subsection.

10.3.3.1 System Sizes, Magnitudes, and Complexities

Definition 10.31 The *size of a system S_s* is the number of components encompassed in the system, i.e.:

$$\begin{aligned} S_s &= \#C \\ &= n_c \end{aligned} \tag{10.19}$$

Definition 10.32 The *magnitude of a system M_s* is the number of asymmetric binary relations among the n_c components of the system including the reflexive relations, i.e.:

$$\begin{aligned} M_s &= \#R = n_r \\ &= \#(C \times C) \\ &= n_c^{\,2} \end{aligned} \tag{10.20}$$

If all self-reflective relations are ruled out in n_r, the pure number of binary relations M'_s in the given system is determined as follows:

$$\begin{aligned} M'_s &= M_s - n_c \\ &= n_c^{\,2} - n_c \\ &= n_c(n_c - 1) \end{aligned} \tag{10.21}$$

Lemma 10.10 The *pure number of binary relations M'_s* equals to exactly two times of the number of pairwise combinations among n_c, i.e.:

$$\begin{aligned} M'_s &= n_c(n_c - 1) \\ &= 2 \bullet \frac{n_c(n_c - 1)}{2} \\ &= 2 \bullet C_{n_c}^{2} \end{aligned} \tag{10.22}$$

where the factor 2 represents the asymmetric binary relation *r*, i.e., a*r*b ≠ b*r*a.

The magnitude of a system determines its complexity. The complicities of systems can be classified based on if they are fully or partially connected. The former is the theoretical upper-bound complexity of systems in which all components are potentially interconnected with each other in all n-nary ways, $1 \leq n \leq n_c = \#C$. The latter is the more typical complexity of systems where components are only pairwisely connected.

Definition 10.33 The *complexity of a fully connected system* C_{max} is a closure of all possible *n*-nary relations R^*, $1 \leq n \leq n_c$, among all components of the given system $n_c = \#C$, i.e.:

$$\begin{aligned} C_{max} &= R^* \\ &= 2\sum_{k=0}^{n_r} \mathrm{C}_{n_r}^{k} \\ &= 2 \bullet 2^{n_r} \\ &\approx 2 \bullet 2^{n_c 2} \\ &= 2^{n_r+1} \\ &= 2^{M_s} \end{aligned} \tag{10.23}$$

where C_{max} is also called the *maximum complexity* of systems.

According to Eq. 10.23, the closure of all possible *n*-nary relations R^* may easily result in an extremely huge degree of complexity for a system with few components. For example, when $n_c = 10$, $C_{max} = 2^{100}$. This explains why most of the real-world systems are really too hard to be modeled and handled.

It is noteworthy that almost all functioning systems are partially connected, because a fully connected system may not represent or provide anything meaningful. Therefore, the complexity of partially connected systems can be simplified as follows.

Definition 10.34 The *complexity of a partially connected system* C_r is determined by the number of asymmetric binary relations M'_s of the system, i.e.:

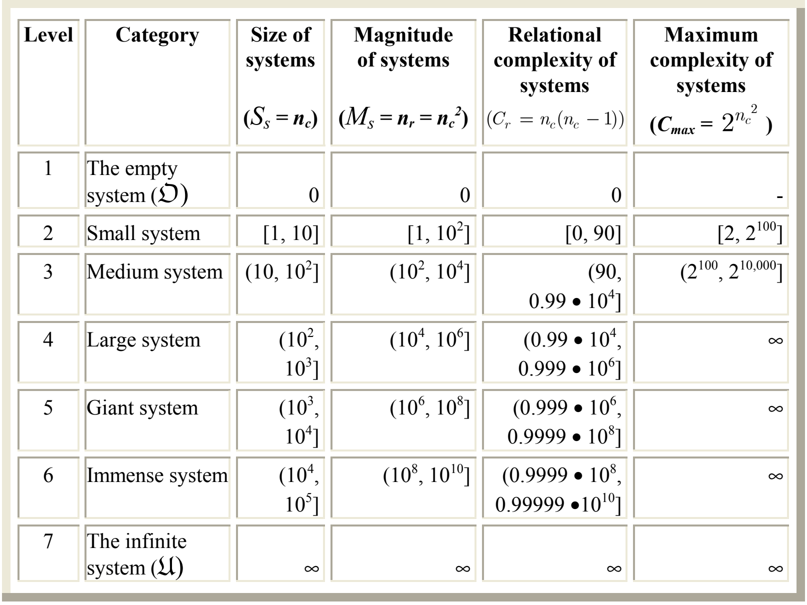
$$\begin{aligned} C_r &= M'_s \\ &= 2 \bullet \mathrm{C}_{n_c}^{2} \\ &= n_c(n_c - 1) \end{aligned} \tag{10.24}$$

where C_r is simply called the *relational complexity* of systems.

10.3.3.2 Taxonomy of System Magnitudes

The taxonomy of system magnitudes [Wang, 2006d] can be classified at seven levels known as the *empty, small, medium, large, giant, immense,* and *infinite systems* from the bottom up. A summary of the relationships between system magnitudes, sizes, internal relations, and complexities can be described in the *system magnitude model* as shown in Table 10.2.

Table 10.2
The System Magnitude Model



Level	Category	Size of systems $(S_s = n_c)$	Magnitude of systems $(M_s = n_r = n_c^2)$	Relational complexity of systems $(C_r = n_c(n_c - 1))$	Maximum complexity of systems $(C_{max} = 2^{n_c^2})$
1	The empty system ($\mathfrak{O}$)	0	0	0	-
2	Small system	[1, 10]	$[1, 10^2]$	[0, 90]	$[2, 2^{100}]$
3	Medium system	$(10, 10^2]$	$(10^2, 10^4]$	$(90, 0.99 \bullet 10^4]$	$(2^{100}, 2^{10,000}]$
4	Large system	$(10^2, 10^3]$	$(10^4, 10^6]$	$(0.99 \bullet 10^4, 0.999 \bullet 10^6]$	∞
5	Giant system	$(10^3, 10^4]$	$(10^6, 10^8]$	$(0.999 \bullet 10^6, 0.9999 \bullet 10^8]$	∞
6	Immense system	$(10^4, 10^5]$	$(10^8, 10^{10}]$	$(0.9999 \bullet 10^8, 0.99999 \bullet 10^{10}]$	∞
7	The infinite system ($\mathfrak{U}$)	∞	∞	∞	∞

Table 10.2 indicates that the complexity of a small system may easily be out of control of human cognitive manageability. This leads to the following theorem.

The 32nd Law of Software Engineering

Theorem 10.1 The *holism complexity of systems* states that within the 7-level magnitudes of systems, known as the *empty, small, medium, large, giant, immense,* and *infinite* systems, almost all systems are too complicated to be cognitively understood or mentally handled as a whole, except small systems or those that can be decomposed into small systems.

According to Theorem 10.1, the basic principle for dealing with complicated systems is system decomposition or modularity, in which the complexity of a lower level subsystem must be small enough to be cognitively manageable. Details of system decomposition theories and the art of system architectures will be developed in the following sections.

10.3.4 HIERARCHICAL ARCHITECTURES OF SYSTEMS

The hierarchical architecture of systems is very much in line with the notion of system philosophy and the hierarchical abstraction of systems that perceive a given system possesses certain subsystems and belongs to certain super system(s). From an organizational point of view, a system can be perceived as a *closure* at a certain level of hierarchical architecture of the abstract or conceptual world. Therefore, a system may be a part of a larger system at the top level except $\mathfrak{U}$, or consists of a number of smaller systems at the lower level except $\mathfrak{D}$. For example, a computer is an electronic information processing system, which may belong to a larger networked system, and at the same time it may contain a number of subsystems such as those of hardware, software, file management system, etc.

It has been empirically observed that the tree-like architecture is a universal hierarchical prototype of systems across disciplines of not only science and engineering, but also sociology and living systems. This subsection explores the theories behind the universal phenomena in system science, which explain why systems have to adopt tree structures in organization, and what the advantages of hierarchical trees in system organizations are.

The discussion may be started from analyzing the underlying reasons that force systems to take hierarchical structures. They are:

- The complexity of an unstructured system can easily grow out of control.
- The efficiency of an unstructured system can be very low.
- The gain of system by coordination may diminish when the overhead for doing so is too high in unstructured systems.

For example, according to the system magnitude model given in Table 10.2, when an unstructured group is getting too large, say $n_c \geq 100$, its relative complexity $C_r = 9{,}999$, and its absolute complexity can be as high as $C_{max} = 2^{10{,}000}$. It is obviously too complicated to be controllable. It is also an indirect proof of the coordinative work organization theory about the limit of group sizes as given in Theorem 8.10.

This example demonstrates the need for a suitable hierarchical structure for dealing with the complexities of large-scale systems. The most ideal structure of organized systems is known as the complete tree.

Definition 10.35 A *complete n-nary tree* $T_c(n, N)$ is a normalized tree in which each node of T_c can have at most n children, each level k of T_c from top-down can have at most n^k nodes, and all levels have allocated the maximum number of possible nodes, except on the rightmost subtrees and at the leave level where there are N nodes, $N \leq n^k$.

It is noteworthy in Definition 10.35, a tree said to be *complete* means that all levels of the tree have been allocated the maximum number of possible nodes, only two types of exceptions are allowed at the leave level and the rightmost subtress. The advantage of complete trees is that the configuration of any complete n-nary tree $T_c(n, N)$ is determined by only two attributes: the unified fan-out n and the number of leave nodes N at the bottom level. For instance, two complete trees $T_{c1}(n_1, N_1) = T_{c1}(2, 3)$ and $T_{c2}(n_2, N_2) = T_{c1}(2, 7)$ are as shown in Fig. 10.6.

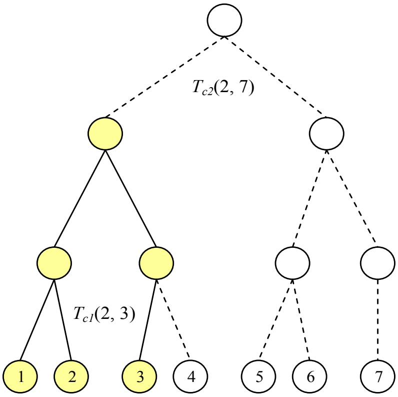


Figure 10.6 Growth of complete binary trees

Definition 10.36 A *normalized system* is a hierarchically structured system where no direct interconnections between nodes belong to different subtrees, and communications between such nodes should be coordinated through a common higher-level parent node.

The 33rd Law of Software Engineering

Theorem 10.2 The *generic topology of normalized systems* states that systems tend to be normalized into a hierarchical structure in the form of a complete n-nary tree.

Systems are forced to be with tree-like structures in order to maintain equilibrium, evolvability, and optimal predictability. The advantages of tree structures of systems can be formally described in the following corollary.

Corollary 10.5 *Advantages of the normalized tree* architecture of systems are as follows:

(a) *Equilibrium*: Looking down from any node at a level of the system tree, except at the leave level, the structural property of fan-out or the number of coordinated components are the same and evenly distributed.

(b) *Evolvablility*: A normalized system does not change the existing structure for future growth needs.

(c) *Optimal predictability*: There is an optimal approach to create a unique system structure $T_c(n, N)$ determined by the attributes of the unified fan-out n and the number of leave nodes N at the bottom level.

10.3.5 THE SYSTEM ORGANIZATION TREE

Based on the model of complete trees, the topology of normalized systems can be implemented by a system organization tree. A structural model of the system organization tree is presented in this subsection for formally describing the hierarchical architectures of normalized systems.

Definition 10.37 A *System Organization Tree* (SOT) is an n-nary complete tree in which all leave nodes represent a *component* and the remainder, all nodes beyond the leave level, represent a *subsystem*.

For instance, a ternary SOT, $SOT(n, N) = SOT(3, 24)$, is shown in Fig. 10.7. Since an SOT is a complete tree, when the leaves (components) do not reach the maximum possible numbers, the right most leaves and subtrees of the SOT will be left open.

According to Definition 10.37, SOT is an ideal model that implements the topology of a normalized system where: a) No direct interconnections between nodes of different subtrees; and b) Communication needs between those nodes belong to different subtrees may go through a common higher-level parent node known as the *manager node*. A set of useful topological properties of SOT is identified as summarized in the following corollary [Wang, 2006d].

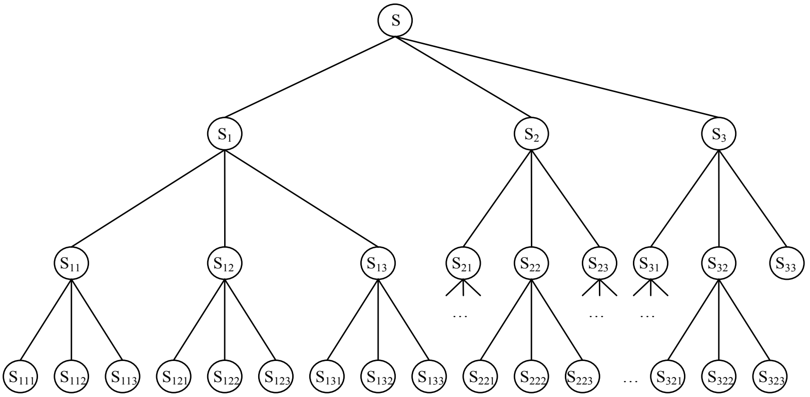


Figure 10.7 A ternary system organization tree *SOT*(3, 24)

Corollary 10.6 An *n*-nary *system organization tree SOT*(*n*, *N*) with the total number of leaves nodes *N* possesses the following properties:

(a) The maximum number of fan-out of any node $\overline{n}_{fo}$:

$$\overline{n}_{fo} = n = L_0 \tag{10.25}$$

(b) The maximum number of nodes at a given level *k*, n_k:

$$n_k = n^k \tag{10.26}$$

(c) The depth of the *SOT*, *d*:

$$d = \left\lceil \frac{\log N}{\log n} \right\rceil \tag{10.27}$$

(d) The maximum number of nodes in the *SOT*, N_{SOT}:

$$N_{SOT} = \sum_{k=0}^{d} n^k \tag{10.28}$$

(e) The maximum number of *components* (on all leaves) in the *SOT*, *N*:

$$N = n^d \tag{10.29}$$

(f) The maximum number of *subsystems* (nodes except all leaves) in the *SOT*, N_m:

$$N_m = N_{SOT}\text{-}N\text{-}1 = \sum_{k=1}^{d\text{-}1} n^k \tag{10.30}$$

It is noteworthy that the determination of the fan-out n, which represents the optimal size of a group in an organization (L_0), is not arbitrary according to Theorem 8.7. The optimization of n will be discussed in Sections 11.2 and 13.4.2. A wide range of applications of SOT in optimizing system organizations has been found. An SOT can be used to model and analyze the architectures and efficiencies of system organizations, for examples, in management science (Section 11.2) and sociology (Section 13.4).

10.3.6 SYSTEM COHESION AND COUPLING

Cohesion and coupling are a pair of important properties of systems. Both cohesion and coupling of a system can be described as a relative ratio of internal and external relations of the system, where the concept of the border of the system is used to distinguish whether a given relation is internal or external.

10.3.6.1 The Border of Systems

The border of a system in topology is a closure that can be described by the intersections of the interior and the exterior of the system.

Definition 10.38 The *interior of a system S, IN,* is a set of components C_S that are fully included in S, i.e.:

$$IN(S) = \{c \mid c \in C_S\} \tag{10.31}$$

Definition 10.39 The *exterior of a system S, ET,* is a set of components C'_S that are excluded in S but are related or interacting with S, i.e.:

$$ET(S) = \{x \mid x \in C'_S \wedge x \notin C_S \wedge r(x, c) \in R^i{}_S \wedge r(c, x) \in R^o{}_S\} \tag{10.32}$$

where $R^i{}_S$ and $R^o{}_S$ represent the sets of input and output relations of systems.

The exterior of a system is also called the *environment.*

In a normalized system, there must be no component that belongs to more than one system.

Definition 10.40 The *border of a system S, B*(S)*,* is a closure of all internal components that separate the interior and exterior of the system, i.e.:

$$\begin{aligned} B(S) &= IN(S) \cap ET(S) \\ &= \{C_S{}^* \mid C_S \not\sqsubseteq S\} \end{aligned} \tag{10.33}$$

According to Definition 10.40, an open system is fused together by the internal relations and is linked with the environment by the external relations cross the border in both ways. Therefore, cohesion and coupling of a system can be defined based on the relative ratio of these relations as analyzed in the following subsections.

10.3.6.2 System Cohesion and Coupling

Assume $\#R^c(S)$ is the number of internal relations of system S, $\#R^i(S)$ the number of input relations, $\#R^o(S)$ the number of output relations, and $\#R(S)$ the total number of both internal and external relations. Then, system cohesion and coupling can be defined below, respectively.

Definition 10.41 The *cohesion of a system S*, $CH(S)$, is defined as a ratio between its number of internal relations $\#R^c(S)$ and the total relations of the system $\#R(S)$, i.e.:

$$\begin{aligned} CH(S) &= \frac{\#R^c(S)}{\#R(S)} \bullet 100\% \\ &= \frac{\#R^c(S)}{\#R^c(S) + \#R^i(S) + \#R^o(S)} \bullet 100\% \end{aligned} \tag{10.34}$$

Definition 10.42 The *coupling of a system S*, $CP(S)$, is defined as a ratio between its number of external relations $\#R^i(S) + \#R^o(S)$ and the total relations of the system $\#R(S)$, i.e.:

$$\begin{aligned} CP(S) &= \frac{\#R^i(S) + \#R^o(S)}{\#R(S)} \bullet 100\% \\ &= \frac{\#R^i(S) + \#R^o(S)}{\#R^c(S) + \#R^i(S) + \#R^o(S)} \bullet 100\% \end{aligned} \tag{10.35}$$

It is noteworthy that system cohesion and coupling are not independent. The relationship between them can be described in the following corollary.

Corollary 10.7 The *cohesion and coupling* of any open system S are complementary, i.e.:

$$CH(S) + CP(S) = 100\% \tag{10.36}$$

The above corollary can be proven based on Definitions 10.41 and 10.42 as given below.

$$
\begin{aligned}
CH(S)+CP(S) &= \frac{\#R^{c}(S)}{\#R(S)} \bullet 100\% + \frac{\#R^{i}(S)+\#R^{o}(S)}{\#R(S)} \bullet 100\% \\
&= \frac{\#R^{c}(S)+\#R^{i}(S)+\#R^{o}(S)}{\#R(S)} \bullet 100\% \qquad (10.37)\\
&= 100\%
\end{aligned}
$$

Therefore, when either cohesion or coupling of a given system is known, the other one can be determined as a complement according to Eq. 10.36.

10.4 System Algebra

The mathematical models of abstract systems have been established in Section 10.3. On the basis of the discussions on the notions, metaphors, philosophies, and topology of systems, this section presents a new mathematical structure known as system algebra [Wang, 2006d], which provides a denotational means for the manipulation of abstract systems and the analysis of complex systems. System algebra is also the foundation for the formal treatment of principles and properties of abstract systems in Section 10.5.

Definition 10.43 *System algebra* is an abstract mathematical structure that provides an algebraic treatment of abstract systems and rules of relational and algebraic operations for forming complex systems.

10.4.1 RELATIONAL OPERATIONS OF SYSTEMS

This subsection describes the relational operations of abstract systems, which provides a formal treatment of system equivalence and comparability. The relations between closed and open systems are analyzed separately. Then, relations and equivalence between them will be discussed.

10.4.1.1 Algebraic Relations of Closed Systems

Relationships between two systems can be equivalent, independent, being subsystem, and being super system. The evaluations of these four types of relationships can be carried out based on the following definitions.

Definition 10.44 Two systems $\widehat{S_1}$ and $\widehat{S_2}$ are *equivalent*, denoted by =, if all sets of components, relations, behaviors, and constraints are identical, i.e.:

$$\begin{aligned} &C_1 = C_2 \wedge R_1 = R_2 \wedge B_1 = B_2 \wedge \Omega_1 = \Omega_2 \Rightarrow \\ &\qquad \widehat{S_1}\ (C_1, R_1, B_1, \Omega_1) = \widehat{S_2}\ (C_2, R_2, B_2, \Omega_2) \end{aligned} \tag{10.38}$$

Definition 10.45 Two systems $\widehat{S_1}$ and $\widehat{S_2}$ are *independent*, denoted by $\not{\mathcal{R}}$, if their component sets are disjoint, i.e.:

$$\begin{aligned} &C_1 \cap C_2 = \varnothing \Rightarrow \\ &\qquad \widehat{S_1}\ (C_1, R_1, B_1, \Omega_1)\ \not{\mathcal{R}}\ \ \widehat{S_2}\ (C_2, R_2, B_2, \Omega_2) \end{aligned} \tag{10.39}$$

It is noteworthy that, by definition, there is no related or overlapped closed systems.

Definition 10.46 A *subsystem* $\hat{S}'$ is a system that is encompassed in another system $\hat{S}$, denoted by $\sqsubseteq$, i.e.:

$$\begin{aligned} &\hat{S}'(C', R', B', \Omega') \sqsubseteq \hat{S}\ (C, R, B, \Omega) \Leftrightarrow \\ &\qquad C' \subseteq C \wedge R' \subseteq R \wedge B' \subseteq B \wedge \Omega' \subseteq \Omega \end{aligned} \tag{10.40}$$

The above definition indicates that a subsystem of a given closed system is a coherent component of the system where the component's relations, behaviors, constraints, and environment are integrated into the system.

Definition 10.47 A *super system* $\hat{S}$ is a system that encompasses one or more subsystems S', denoted by $\sqsupseteq$, i.e.:

$$\begin{aligned} &\hat{S}\ (C, R, B, \Omega) \sqsupseteq \hat{S}'(C', R', B', \Omega') \Leftrightarrow \\ &\qquad C' \subseteq C \wedge R' \subseteq R \wedge B' \subseteq B \wedge \Omega' \subseteq \Omega \end{aligned} \tag{10.41}$$

According to Definition 10.47 the composition of systems can be carried out when all four sets in the tuple that determine a system or subsystem are merged. Further discussion on the mechanisms of system compositions will be presented in Section 10.4.2.

10.4.1.2 Algebraic Relations of Open Systems

Relationships between two open systems can be equivalent, independent, overlapped, related, being subsystem, and being super system.

The evaluations of these relationships can be carried out based on the following definitions.

Definition 10.48 Two open systems S_1 and S_2 are *equivalent*, denoted by =, if all sets of components, relations, behaviors, constraints, and environments are identical, i.e.:

$$\begin{aligned} &C_1 = C_2 \wedge R^c{}_1 = R^c{}_2 \wedge R^i{}_1 = R^i{}_2 \wedge R^o{}_1 = R^o{}_2 \wedge B_1 = B_2 \wedge \Omega_1 = \Omega_2 \wedge \Theta_1 = \Theta_2 \\ &\quad \Rightarrow S_1\,(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1) = \\ &\quad\quad S_2\,(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \end{aligned} \tag{10.42}$$

Definition 10.49 Two open systems S_1 and S_2 are *independent*, denoted by $\not{R}$, if both their component sets and external relation sets are disjoint, i.e.:

$$\begin{aligned} &C_1 \cap C_2 = \varnothing \wedge R^i{}_1 \cap R^i{}_2 = \varnothing \wedge R^o{}_1 \cap R^o{}_2 = \varnothing \Rightarrow \\ &\quad S_1\,(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1)\, \not{R} \\ &\quad S_2\,(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \end{aligned} \tag{10.43}$$

Definition 10.50 Two open systems S_1 and S_2 are *overlapped*, denoted by Π, if their component sets are overlapped, i.e.:

$$\begin{aligned} &C_1 \cap C_2 \neq \varnothing \Rightarrow \\ &\quad S_1\,(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1)\, \Pi \\ &\quad S_2\,(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \end{aligned} \tag{10.44}$$

Definition 10.51 Two open systems S_1 and S_2 are *related*, denoted by R, if there is at least a coupled I/O relation satisfying $\forall a \in C_1$, $\forall b \in C_2$, $r(a, b) \in R^o{}_1 \cap R^i{}_2$ or $r(b, a) \in R^o{}_2 \cap R^i{}_1$, i.e.:

$$\begin{aligned} &R^o{}_1 \cap R^i{}_2 \neq \varnothing \vee R^o{}_2 \cap R^i{}_1 \neq \varnothing \Rightarrow \\ &\quad S_1\,(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1)\, R \\ &\quad S_2\,(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \end{aligned} \tag{10.45}$$

Definition 10.52 A *subsystem S'* is a system that is encompassed in another system S, denoted by ⊑, i.e.:

$$\begin{aligned} &S'(C', R^{c'}, R^{i'}, R^{o'}, B', \Omega', \Theta') \sqsubseteq S(C, R^c, R^i, R^o, B, \Omega, \Theta) \Leftrightarrow \\ &\quad C' \subseteq C \wedge R^{c'} \subseteq R^c \wedge R^{i'} \subseteq R^i \wedge R^{o'} \subseteq R^o \wedge \\ &\quad B' \subseteq B \wedge \Omega' \subseteq \Omega \wedge \Theta' = \Theta \end{aligned} \tag{10.46}$$

The above definition indicates that a subsystem of either an open or closed system is an open system. In other words, the decomposition of any system results in multiple open subsystems.

Definition 10.53 A *super system S* is a system that encompasses one or more subsystems *S'*, denoted by $\sqsubseteq$, i.e.:

$$\begin{aligned} S'(C', R^{c'}, R^{i'}, R^{o'}, B', \Omega', \Theta') \sqsubseteq S(C, R^c, R^i, R^o, B, \Omega, \Theta) \Leftrightarrow \\ C' \subseteq C \wedge R^{c'} \subseteq R^c \wedge R^{i'} \subseteq R^i \wedge R^{o'} \subseteq R^o \wedge \\ B^{'} \subseteq B \wedge \Omega' \subseteq \Omega \ \wedge \Theta' = \Theta \end{aligned} \tag{10.47}$$

According to Definition 10.53 the composition of two or more open systems can be carried out when all seven sets in the tuple that determine a system or subsystem are merged. Further discussion on system composition will be presented in Section 10.4.2.

10.4.1.3 Relations between Closed and Open Systems

The previous subsections analyzed the relations of closed systems and open systems separately. However, it is noteworthy that closed and open systems are transformable, when the environment of open systems is treated as a super system.

Based on the definitions of closed systems (Eq. 10.2) and open systems (Eq. 10.8), the above notion can be described in the following theorem and corollaries.

The 33rd Principle of Software Engineering

Theorem 10.3 The *equivalence between open and closed systems* states that an open system S and a closed system $\widehat{S}$ ($\Theta_{\widehat{S}} = C \not\sqsubseteq \widehat{S}$) in the same context is transformable when their environments Θ_S and $\Theta_{\widehat{S}}$ are taken into consideration, respectively, i.e.:

$$\begin{cases} \widehat{S} = \mathrm{S} \sqcup \Theta_{\mathrm{S}} \\ \mathrm{S} = \widehat{S} \sqcup \Theta_{\hat{\mathrm{S}}} \end{cases} \tag{10.48}$$

According to Theorem 10.3, the following properties of equivalence between closed and open systems can be derived.

Corollary 10.8 Any subsystem $\widehat{S_k}$ of a closed system $\widehat{S}$ is an open system *S*, i.e.:

$$\forall \widehat{S_k} \subseteq \widehat{S} \;\Rightarrow\; R^i{}_k \neq \varnothing \wedge R^o{}_k \neq \varnothing \wedge \Theta_k = C_s \setminus C_k \neq \varnothing \qquad (10.49)$$

Corollary 10.9 Any supersystem *S* of a given set of *n* open systems S_k, plus their environments Θ_k, $1 \leq k \leq n$, is a closed system, i.e.:

$$\forall S_k, \Theta_k, \; \widehat{S} = \bigsqcup_{k=1}^{n} (S_k \sqcup \Theta_k) \Rightarrow R^i{}_S = \varnothing \wedge R^o{}_S = \varnothing \wedge \Theta_S = \varnothing \qquad (10.50)$$

10.4.2 ALGEBRAIC OPERATIONS OF SYSTEMS

System algebra provides a powerful means to manipulate abstract systems as a mathematical entity. A set of algebraic operations on systems, such as system conjunction, disjunction, difference, composition, and decomposition, is defined in the following subsections based on algebraic rules.

10.4.2.1 System Conjunction

An operation on incremental union of multiple sets of relations is introduced first as a preparation before defining system conjunction.

Definition 10.54 An *incremental union* $\mho$ of two sets of relations R_1 and R_2 from different systems S_1 and S_2 are a normal union of A_1 and A_2 plus a newly generated incremental set ΔR_{12}, i.e.:

$$\begin{aligned} R &= R_1 \mho R_2 \\ &= R_1 \cup R_2 \cup \Delta R_{12}, \quad R_1 \mathsf{E}\, S_1, R_2 \mathsf{E}\, S_2, \Delta R_{12} \mathsf{E}\, S \end{aligned} \qquad (10.51)$$

where E denotes a special *membership* relation of a set in a system.

Eq. 10.51 reveals an important property of systems relations, known as the incremental union, which indicates that the merge of two systems results in new relations and/or behaviors (functions).

Definition 10.55 The *conjunction* of two closed systems $\widehat{S_1}$ and $\widehat{S_2}$, denoted by $\sqcup$, results in a super system $\widehat{S}$ that is formed by union of both sets of components and constraints, as well as *incremental union* of both sets of relations and behaviors, respectively, i.e.:

$$\begin{aligned}
&\widehat{S_1}\ (C_1, R_1, B_1, \Omega_1) \sqcup \widehat{S_2}\ (C_2, R_2, B_2, \Omega_2) \\
&\quad \triangleq \widehat{S}\ (C_1 \cup C_2, R_1 \mho R_2, B_1 \mho B_2, \Omega_1 \cup \Omega_2) \\
&\quad = \widehat{S}\ (C_1 \cup C_2, R_1 \cup R_2 \cup \Delta R_{12}, B_1 \cup B_2 \cup \Delta B_{12}, \Omega_1 \cup \Omega_2) \\
&\quad = \widehat{S}\ (C, R, B, \Omega) \qquad (10.52)
\end{aligned}$$

The 34th Law of Software Engineering

Theorem 10.4 The *system gain of functionality* states that system conjunction or composition between two systems S_1 and S_2 creates *new relations* ΔR_{12} and/or *new behaviors* (functions) ΔB_{12} that are solely a property of the newly established super system S, which can be determined by the sizes of the two intersected component sets $\#C_1$ and $\#C_2$, i.e.:

$$\begin{aligned}
\Delta R_{12} &= \#R - (\#R_1 + \#R_2) \\
&= (\#(C_1 + C_2))^2 - ((\#C_1)^2 + (\#C_2)^2) \\
&= 2\ (\#C_1 \bullet \#C_2) \qquad (10.53)
\end{aligned}$$

The discovery in Theorem 10.4 reveals that the mathematical explanation of system gains is the newly generated relations ΔR_{12} and/or behaviors ΔB_{12} during the conjunction of two systems or subsystems. The empirical awareness of this key system property has been intuitively or empirically described in the literature in system engineering for centuries. However, Theorem 10.4 is the first rigorous explanation of the mechanism of system gains during the incremental system conjunctions and compositions.

Theorem 10.4 may be used to predict the maximum numbers of newly established relations and behaviors when two systems are conjoined. According to Eq. 10.53, the maximum *incremental* or *system gain* equals to the number of by-directly interconnection between all components in both S_1 and S_2, i.e., $2(\#C_1 \bullet \#C_2)$. It is noteworthy that if C_1 and C_2 are disjoint, there is no incremental gain in system conjunctions.

More generally, Theorem 10.4 and Definition 10.55 can be extended to open systems as below.

Definition 10.56 The *conjunction* of two open systems S_1 and S_2, denoted by $\sqcup$, results in a super system that is formed by *incremental conjunctions* of both sets of relations and behaviors, respectively, as well as simple conjunctions of sets of components, constraints, and environments, i.e.:

$$\begin{aligned} & S_1(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1) \sqcup S_2(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \\ & \quad \triangleq S(C_1 \cup C_2, R^c{}_1 \cup R^c{}_2 \cup \Delta R^c{}_{12}, R^i{}_1 \cup R^i{}_2, R^o{}_1 \cup R^o{}_2, \\ & \qquad B_1 \cup B_2 \cup \Delta B_{12}, \Omega_1 \cup \Omega_2, \Theta_1 \cup \Theta_2) \\ & \quad = S(C, R^c, R^i, R^o, B, \Omega, \Theta) \end{aligned} \tag{10.54}$$

The operation of open system conjunction is illustrated in Fig. 10.8, where the generation of the new relations $\Delta R^c{}_{12} = \Delta R^c{}_1 + \Delta R^c{}_2$ in S after the conjunction of S_1 and S_2 can be observed.

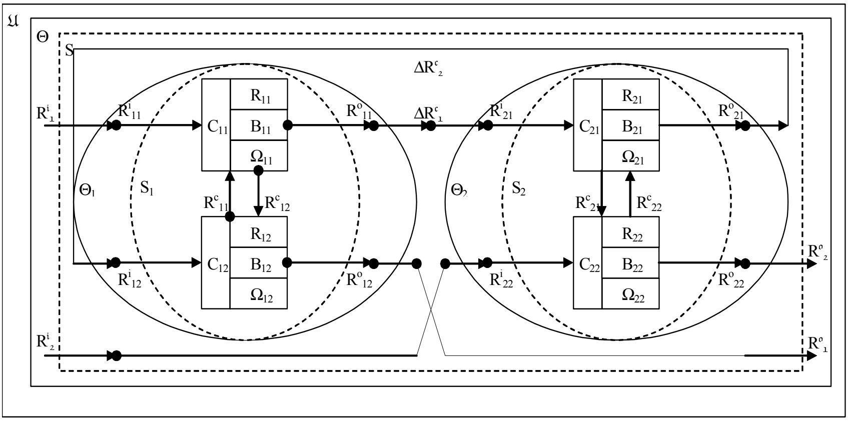


Figure 10.8 The conjunction of two open systems

Example 10.3 According to Definition 10.56, the conjunction of the two open systems $S_1(Clock)$ and $S_2(Alarm)$ as given in Examples 10.1 and 10.2 results in a new system $S(Alarm\_Clock)$ as follows:

$$\begin{aligned} S(C, R^c, R^i, R^o, B, \Omega, \Theta) &= S_1(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1) \sqcup \\ & \qquad S_2(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \\ &= S(C_1 \cup C_2, R^c{}_1 \cup R^c{}_2 \cup \Delta R^c{}_{12}, R^i{}_1 \cup R^i{}_2, R^o{}_1 \cup R^o{}_2, \\ & \qquad B_1 \cup B_2 \cup \Delta B_{12}, \Omega_1 \cup \Omega_2, \Theta_1 \cup \Theta_2) \end{aligned}$$

where

- The set of components:
  $C = C_1 \cup C_2$
  $= \{Processor, Keypad, LEDs, ClockPulse\} \cup \{Processor, Keypad, LEDs, Bell\}$
  $= \{Processor, Keypad, LEDs, ClockPulse, Bell\}$

- The set of internal relations:

$$\begin{aligned}
R^c &= R^c{}_1 \cup R^c{}_2 \cup \Delta R^c{}_{12} \\
&= \{Input(Keypad,\ Processor), \\
&\qquad Tick(ClockPulse,\ Processor), \\
&\qquad Output(Processor,\ LEDs) \\
&\quad\ \} \cup \\
&\quad\ \{Input(Keypad,\ Processor), \\
&\qquad AlarmCheck(Time,\ Alarm), \\
&\qquad AlarmRelease(Keypad,\ Processor), \\
&\qquad Output(Processor,\ LEDs), \\
&\qquad Ring(Processor,\ Bell) \\
&\quad\ \} \cup \\
&\quad\ \boldsymbol{\{Select(Clock,\ Alarm)\}} \qquad\qquad //\ \Delta R^c{}_{12} \\
&= \{Input(Keypad,\ Processor), \\
&\qquad Tick(ClockPulse,\ Processor), \\
&\qquad AlarmCheck(Time,\ Alarm), \\
&\qquad AlarmRelease(Keypad,\ Processor), \\
&\qquad Output(Processor,\ LEDs), \\
&\qquad Ring(Processor,\ Bell), \\
&\qquad Select(Clock,\ Alarm) \\
&\quad\ \}
\end{aligned}$$

- The set of input relations:

$$\begin{aligned}
R^i &= R^i{}_1 \cup R^i{}_2 \\
&= \{SetTime(User,\ Keypad),\ SetAlarm(User,\ Keypad)\}
\end{aligned}$$

- The set of output relations:

$$\begin{aligned}
R^o &= R^o{}_1 \cup R^o{}_2 \\
&= \{ShowTime(LEDs,\ User),\ ShowAlarm(LEDs,\ User)\}
\end{aligned}$$

- The set of behaviors:

$$\begin{aligned}
B &= B_1 \cup B_2 \cup \Delta B_{12} \\
&= \{SetTime,\ ShownTime,\ tick\} \cup \\
&\quad\ \{SetAlarm,\ ShownAlarm,\ CheckAlarm,\ Ring,\ AlarmRelease\} \cup \\
&\quad\ \boldsymbol{\{SelectClock,\ SelectAlarm\}} \qquad\qquad //\ \Delta B_{12}
\end{aligned}$$

- The set of constraints:

$$\begin{aligned}
\Omega &= \Omega_1 \cup \Omega_2 \\
&= \{Time = \mathrm{hh} \times \mathrm{mm} \times \mathrm{ss},\ Alarm = \mathrm{hh} \times \mathrm{mm}\}
\end{aligned}$$

- The environment:

$$\begin{aligned}
\Theta &= \Theta_1 \cup \Theta_2 \\
&= \{\mathrm{User}\}
\end{aligned}$$

Note that newly generated relations *Select*(*Clock, Alarm*), as well as behaviors *SelectClock* and *SelectAlarm* in system *S*(*Alarm_Clock*), do not belong to either subsystem S(*Clock*) or S(*Alarm*).

10.4.2.2 System Difference

Definition 10.57 The *difference* between a closed system *S* and a subsystem $\widehat{S_1}$, denoted by ⊟, results in a subsystem $\widehat{S_2}$ that is formed by the differences of sets of components and constraints, difference of sets of relations minus both R_1 and ΔR_{12}, and difference of sets of behaviors minus both B_1 and ΔB_{12}, i.e.:

$$\begin{aligned}&\widehat{S}\,(C, R, B, \Omega) \boxminus \widehat{S_1}\,(C_1, R_1, B_1, \Omega_1)\\ &\quad \triangleq \widehat{S_2}\,(C \setminus C'_1, R \setminus (R'_1 \cup \Delta R_{12}), B \setminus (B_1 \cup \Delta B_{12}), \Omega \setminus \Omega_1)\\ &\quad = \widehat{S_2}\,(C_2, R_2, B_2, \Omega_2) \qquad (10.55)\end{aligned}$$

where $C'_1 \subseteq C_1 \wedge C'_1 \cap C_2 = \varnothing$ and $R'_1 \subseteq R_1 \wedge R'_1 \cap R_2 = \varnothing$.

According to Definition 10.57, a difference of a subsystem from a system *S* will result in the removal of not only the given subsystem but also all interrelations and incremental behaviors between the subsystem and other subsystem in *S*.

It is noteworthy that if there is an overlap between two subsystems in $\widehat{S}$, the operation of system difference will only remove C'_1 and R'_1, which are only the disjoint subsets of C_1 and R_1, respectively.

More generally, Definition 10.57 can be extended to open systems as follows.

Definition 10.58 The *difference* between an open system *S* and a subsystem S_1, denoted by ⊟, results in an open subsystem S_2 that is formed by the differences of sets of components, input relations, output relations, and constraints, difference of sets of internal relations minus both $R^{c'}{}_1$ and $\Delta R^c{}_{12}$, and difference of sets of behaviors minus both B_1 and ΔB_{12}, i.e.:

$$\begin{aligned}&S(C, R^c, R^i, R^o, B, \Omega, \Theta) \boxminus S_1(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1)\\ &\quad \triangleq S_2(C \setminus C'_1, R^c \setminus (R^{c'}{}_1 \cup \Delta R^c{}_{12}), R^i \setminus R^i{}_1, R^o \setminus R^o{}_1,\\ &\qquad\quad B \setminus (B_1 \cup \Delta B_{12}), \Omega \setminus \Omega_1, \Theta \setminus \Theta'_1)\\ &\quad = S_2(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \qquad (10.56)\end{aligned}$$

where $C'_1 \subseteq C \wedge C'_1 \cap C_2 = \varnothing$, $R^{c'}{}_1 \subseteq R^c{}_1 \wedge R^{c'}{}_1 \cap R^c{}_2 = \varnothing$, and $\Theta'_1 \subseteq \Theta_1 \wedge \Theta'_1 \cap \Theta_2 = \varnothing$.

As shown in Eq. 10.56, the operation of system difference will only remove the disjoint subsets, i.e., C'_1, $R^{c'}_1$, and/or Θ'_1 from S, respectively.

An open system difference $S \boxminus S_1 = S_2$ is illustrated in Fig. 10.9.

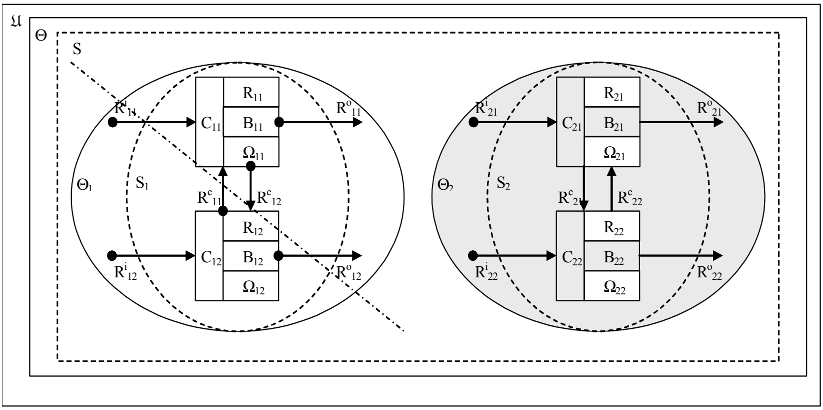


Figure 10.9 The difference of two open systems ($S \boxminus S_1 = S_2$)

Example 10.4 According to Definition 10.58, the difference of the two open systems $S(AlarmClock)$ and $S_2(Alarm)$ as given in Examples 10.1 and 10.2 results in a new subsystem $S_1(Clock)$ as follows:

$$\begin{aligned}
S_1(C_1, R^c{}_1, R^i{}_1, R^o{}_1, B_1, \Omega_1, \Theta_1) &= S(C, R^c, R^i, R^o, B, \Omega, \Theta) \boxminus \\
&\qquad S_2(C_2, R^c{}_2, R^i{}_2, R^o{}_2, B_2, \Omega_2, \Theta_2) \\
&= S_1(C \setminus C'_2, R^c \setminus (R^{c'}{}_2 \cup \Delta R^c{}_{12}), R^i \setminus R^i{}_2, R^o \setminus R^o{}_2, \\
&\qquad B \setminus (B_2 \cup \Delta B_{12}), \Omega \setminus \Omega_2, \Theta \setminus \Theta'_2)
\end{aligned}$$

where

- The set of components:
  $C_1 = C \setminus C'_2$
  $= \{Processor, Keypad, LEDs, ClockPulse, Bell\} \setminus \{Bell\}$
  $= \{Processor, Keypad, LEDs, ClockPulse\}$

- The set of internal relations:
  $R^c{}_1 = R^c \setminus (R^{c'}{}_2 \cup \Delta R^c{}_{12}),$
  $= \{Input(Keypad, Processor),$
  $Tick(ClockPulse, Processor),$
  $AlarmCheck(Time, Alarm),$
  $AlarmRelease(Keypad, Processor),$
  $Output(Processor, LEDs),$
  $Ring(Processor, Bell),$
  $Select(Clock, Alarm)$
  $\} \setminus$

$$\begin{aligned}
&\{AlarmCheck(Time,\ Alarm),\\
&\ \ AlarmRelease(Keypad,\ Processor),\\
&\ \ Ring(Processor,\ Bell)\\
&\} \cup\\
&\{Select(Clock,\ Alarm)\}\\
= &\{Input(Keypad,\ Processor),\\
&\ \ Tick(ClockPulse,\ Processor),\\
&\ \ Output(Processor,\ LEDs)\\
&\}
\end{aligned}$$

- The set of input relations:
  $$\begin{aligned} R^i{}_1 &= R^i \setminus R^i{}_2 \\ &= \{SetTime(User,\ Keypad)\} \end{aligned}$$

- The set of output relations:
  $$\begin{aligned} R^o{}_1 &= R^o \setminus R^o{}_2 \\ &= \{ShowTime(LEDs,\ User)\} \end{aligned}$$

- The set of behaviors:
  $$\begin{aligned} B_1 &= B \setminus \{B_2 \cup \Delta B_{12}\} \\ &= \{SetTime,\ ShownTime,\ tick,\ SetAlarm,\ ShownAlarm,\\ &\qquad CheckAlarm,\ Ring,\ AlarmRelease,\ SelectClock,\\ &\qquad SelectAlarm\\ &\quad\} \setminus\\ &\quad \{SetAlarm,\ ShownAlarm,\ CheckAlarm,\ Ring,\ AlarmRelease,\\ &\qquad SelectClock,\ SelectAlarm\\ &\quad\}\\ &= \{SetTime,\ ShownTime,\ tick\} \end{aligned}$$

- The set of constraints:
  $$\begin{aligned} \Omega_1 &= \Omega \setminus \Omega_2 \\ &= \{Time = \text{hh} \times \text{mm} \times \text{ss}\} \end{aligned}$$

- The environment:
  $$\begin{aligned} \Theta_1 &= \Theta \setminus \Theta'_2 \\ &= \{\text{User}\} \end{aligned}$$

Note that within the given system S, since there is an overlap between the two subsystems S_1 and S_2, the difference operation may only remove the disjoint subset C'_2, $R^{c'}{}_2$, and Θ'_2 from S.

10.4.2.3 System Composition

System composition is the most complicated system operation that integrates two or more systems into a super system with a hierarchical architecture. It is noteworthy that only open systems may be composed, or a closed system should be transformed into an open system before it can be composed.

There are three basic forms of system compositions known as: *parallel* ($\|$), *serial* ($\rightarrow$), and *nested* ($\rightarrowtail$) compositions as shown in Fig. 10.10. Complex system compositions can be represented by a combination of these three basic forms. The syntaxes and semantics of these three system composition rules have been defined in RTPA in Section 4.6.5 and Section 6.6.2, respectively.

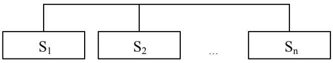
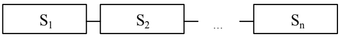
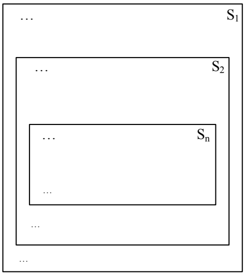
No	Form of composition	Syntax	Example
1	Parallel	$S_1 \| S_2$	$S \triangleq S_1 \| S_2 \| \ldots \| S_n$ 
2	Serial	$S_1 \rightarrow S_2$	$S \triangleq S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$ 
3	Nested	$S_1 \rightarrowtail S_2$	$S \triangleq S_1 \rightarrowtail S_2 \rightarrowtail \ldots \rightarrowtail S_n$ 

Figure 10.10 Basic forms of system compositions

Definition 10.59 The *composition* of two open systems S_1 and S_2, denoted by $\uplus$, is an integration of both systems into a super system S at a given level of the system hierarchy by one of the compositional relations $R_c = \{\|, \rightarrow, \rightarrowtail\}$, i.e.:

$$S(C, R, B, \Omega, \Theta) \triangleq S_1(C_1, R_1, B_1, \Omega_1, \Theta_1) \uplus S_2(C_2, R_2, B_2, \Omega_2, \Theta_2) \qquad (10.57)$$

where $\uplus \in R_c$.

Eq. 10.57 can be extended to *n*-nary compositions as given below:

$$S(C, R, B, \Omega, \Theta) \triangleq S_1 \uplus_{12} S_2 \uplus_{23} \ldots \uplus_{n-1,n} S_n \tag{10.58}$$

where $\uplus_{ij} \in R_c$.

According to Definition 10.59, a system can be integrated from the bottom up by a series of compositions level by level in a system hierarchy.

Example 10.5 A composed system $S(C, R, B, \Omega, \Theta)$ as given in Fig. 10.11 can be formally described below.

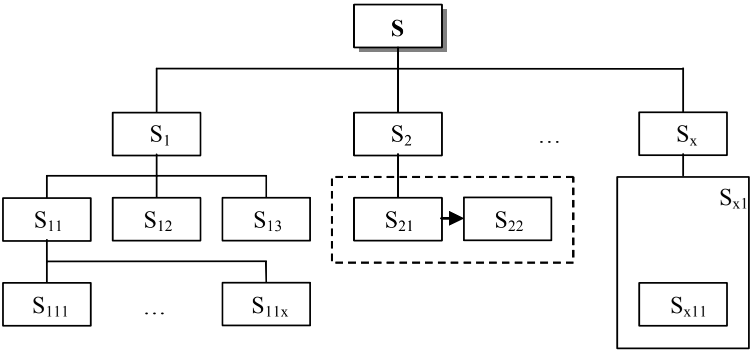


Figure 10.11 The hierarchical organization chart of system compositions

$$\begin{aligned} S(C, R, B, \Omega, \Theta) \triangleq\ & S_1 \\ & \| S_2 \\ & \| \ldots \\ & \| S_x \end{aligned}$$

in which the subsystems of *S* can be refined as follows:

$$\begin{aligned} S_1 \triangleq\ & S_{11} \\ & \| S_{12} \\ & \| S_{13} \\ =\ & (\ S_{111} \\ & \quad \| \ldots \\ & \quad \| S_{11x} \\ & \ ) \\ & \| S_{12} \\ & \| S_{13} \end{aligned}$$

$$S_2 \triangleq S_{21} \rightarrow S_{22}$$

$$S_x \triangleq S_{x1} \rightarrowtail S_{x11}$$

where ||, →, and $\rightarrowtail$ denote the parallel, serial, and embedded (function call) relations, as defined in Section 4.6.5 in RTPA, respectively.

In the hierarchical organization chart of systems as shown in Fig. 10.11, the labeling convention is defined below.

Definition 10.60 The *labeling convention of nodes* in the system organization chart is that each node is labeled by *d* digits where *d* is the depth of the node in the hierarchy. The configuration of the *d* digits is as follows: a) The last digit is always the series number of the nodes at the same level and belongs to the same parent; and b) The remainder of the proceeding digit(s) are the identification label of the node's parent.

Note that *x* represents a digit that is flexible and will be instantiated for a specific case. Also, an *x* that appears in different places may be different.

For simplifying system architectures, system compositions should obey the following rules.

Lemma 10.11 The following *architectural rules of a normalized system* should be maintained in system compositions:

Rule 1. Direct relations between subsystems at different levels of the hierarchy are not allowed.

Rule 2. Direct relations between components of different subsystems are not allowed.

Rule 3. Communications across systems and subsystems should be through commonly inherited higher-level system(s) in the system hierarchy.

For example, subsystems S_{111} and S_{13} should communicate through commonly shared system S_1 rather than directly link by themselves; subsystems S_{13} and S_{22} should communicate through commonly inherited system *S*. These rules ensure that coordination between functions of different system/subsystems can be carried out via standard interfacing mechanisms, and a strict system hierarchical can be maintained.

10.4.2.4 System Decomposition

System decomposition is an inverse operation of system composition that breaks up a system into two or more subsystems. It is noteworthy both open and closed systems can be decomposed, and all of them result in open subsystems.

Definition 10.61 The *decomposition of an open systems S*, denoted by $\pitchfork$, is to break up S into two subsystems at the same level of the system hierarchy by one of the compositional relations $R_c = \{||, \rightarrow, \rightarrowtail\}$, i.e.:

$$S(C, R, B, \Omega, \Theta) \triangleq \\ S_1(C_1, R_1, B_1, \Omega_1, \Theta_1) \pitchfork S_2(C_2, R_2, B_2, \Omega_2, \Theta_2) \qquad (10.59)$$

where $\pitchfork \in R_c$.

Eq. 10.59 can be extended to n-nary decompositions as given below:

$$S(C, R, B, \Omega, \Theta) \triangleq S_1 \pitchfork_{12} S_2 \pitchfork_{23} \ldots \pitchfork_{n-1,n} S_n \qquad (10.60)$$

where $\pitchfork_{ij} \in R_c$.

Similarly, the decomposition of a closed system can be defined below.

Definition 10.62 The *decomposition of a closed system* $\widehat{S}$, denoted by $\pitchfork$, is to break up $\widehat{S}$ into two subsystems at the same level of the system hierarchy by one of the compositional relations $R_c = \{||, \rightarrow, \rightarrowtail\}$, i.e.:

$$\widehat{S}(C, R, B, \Omega) \triangleq \\ \widehat{S_1}(C_1, R_1, B_1, \Omega_1) \pitchfork \widehat{S_2}(C_2, R_2, B_2, \Omega_2) \qquad (10.61)$$

where $\pitchfork \in R_c$.

Eq. 10.61 can be extended to n-nary decompositions of closed systems as given below:

$$\widehat{S}(C, R, B, \Omega) \triangleq \widehat{S_1} \pitchfork_{12} \widehat{S_2} \pitchfork_{23} \ldots \pitchfork_{n-1,n} \widehat{S_n} \qquad (10.62)$$

where $\pitchfork_{ij} \in R_c$.

According to Definitions 10.61 and 10.62, either an open or a closed system can be resolved from the top down by a series of decompositions level by level in the system hierarchy.

System decomposition can be illustrated by the same diagram as shown in Fig. 10.11. The only difference between system composition and decomposition is that the former is a bottom-up operation, while the latter is a top-down operation.

10.5 Principles of System Science

The theories of system science have evolved from classic to contemporary with I. Prigogine's *dissipative structure theory* [Prigogine and Stengers, 1984/97], H. Haken's *synergetics* [Haken, 1977/83], and M. Eigen's *hypercycle theory* [Eigen and Schuster, 1979]. Then, the field has shifted on proposals of systematology [Klir, 2001], complex systems theory [Ashby, 1958; Simon, 1965; Zadeh, 1973; Gaines, 1977/76], fuzzy theories [Zadeh, 1965/82; Pedrycz, 1981; Gaines, 1983; Negoita, 1989], and chaos theories [Prigogine and Stengers, 1984/97; Ford, 1986; Skarda and Freeman, 1987].

The *abstract system theory* supporting by system algebra is developed in 2006 [Wang, 2006d] is the latest attempt to provide a formal and rigorous treatment of abstract systems, their properties, and principles. This section describes fundamental principles of system science on the basis of abstract system theories and system algebra. A comprehensive set of system phenomena and principles, such as system fusions, system functions and behaviors, work done by systems, the maximum output of systems, system equilibrium and organization, system synchronization and coordination, and system dissimilation, are formally explained by the abstract system theories.

10.5.1 SYSTEM FUSIONS

Systems are needed because of the special and self-productive properties known as the fusion effect, which is not possessed by any of its parts or components before compositions.

Definition 10.63 The *fusion effect* of a system is a self-productive property of systems that only appears when the system functions coherently as a whole.

The fusion effect can be quantitatively analyzed using Law 34 of system gain of functionality as stated in Theorem 10.4. Based on Law 34, the following corollaries on system fusion during system compositions are derived.

Lemma 10.12 The *system fusion principle* states that the *fusion effect* of systems is generated by either increments of quantity in *C* or increments of diversity in *R*.

Corollary 10.10 There exists a threshold that triggers the fusion effect of systems known as the *critical mass* Q_{cm}, which is the minimum quantity for obtaining or implementing the system fusion effect over the incremental of quantity.

The critical mass and the curve of system fusion effect can be illustrated as shown in Fig. 10.12. An important phenomenon in system fusion is the mutation of system functions triggered by the critical mass Q_{cm}.

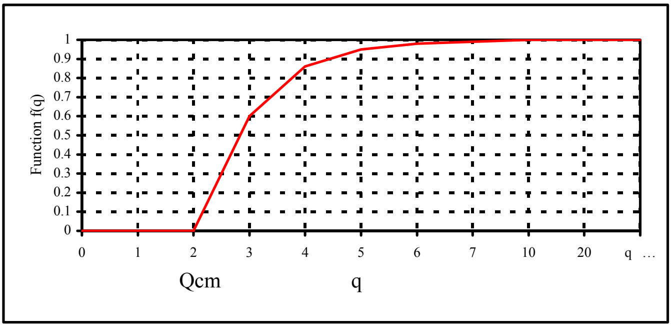


Figure 10.12 System fusion and the critical mass

The 35th Law of Software Engineering

Theorem 10.5 *System mutation* states that the gradual increment of quantities, e.g., ΔC or ΔR, in a system beyond the point of the critical mass Q_{cm} triggers the abrupt generation of functionality (quality) F_{cm} of the system.

Definition 10.64 The *behavior of system mutation* can be formally modeled as a system function $f(q)$, i.e.:

$$\begin{cases} f(q) = 0, \ q < Q_{cm} \\ f(q) = f_{cm}(q), \ q \geq Q_{cm} \end{cases} \tag{10.63}$$

where $f_{cm}(q)$ is the active system behavior that is system- or context-specific.

Example 10.6 The following examples show, respectively, how quantitative or diverse increments generate the fusion effects in systems.

(a) A *voltage evaluator* as a circuit system: The output is triggered by the gradual increment of its input V_{i}, i.e.:

$$S(voltage\_\ evaluator) = \begin{cases} 0, \ V_i < 2.5\text{v} \\ 1, \ V_i \geq 2.5\text{v} \end{cases}$$

(b) A *water molecular* as a chemical system: The resulting compound gains a set of new chemical and physical properties that are not possessed by its individual component molecules, i.e.:

$$S(water\_molecular) = 2\text{H} + \text{O} \Rightarrow \text{H}_2\text{O}$$

10.5.2 SYSTEM FUNCTIONS AND BEHAVIORS

There was an attempt in system theories to describe the behaviors of all systems or a category of systems by a single mathematical model. Because of the extreme complexity and wide variety of systems, this aim seems impossible to be achieved. However, as described in Sections 10.3 and 10.4, the generic mathematical models for the architectures of abstract systems do exist.

Although homogeneously structured systems may implement different behaviors, heterogeneously structured systems may implement identical behaviors. To this extent, computers and software engineering are a generic system engineering platform to implement a wide range of system behaviors by homogeneous and/or heterogeneous system architectures.

The behaviors of systems vary greatly at the application level. However, there are only a finite set of meta behaviors shared by all the applications at the fundamental level in both physical and intelligent systems. These meta behaviors can be modeled by a set of 17 meta processes in RTPA. The 17 meta processes can be composed by a set of 17 process relations to built the architectures and behaviors of larger components and complex systems. In other words, computing systems or human behaviors can be described and implemented by a set of processes based on compositions of the meta processes and their algebraic relations [Wang, 2002a/03c].

On the basis of Theorem 4.6, the set of 17 meta processes $\mathfrak{P}$ in computing have been defined in Section 4.6.4 as follows:

$$\mathfrak{P} = \{:=, \text{�}, \Rightarrow, \Leftarrow, \not\Leftarrow, \gtrdot, \lessdot, |\gtrdot, |\lessdot, \underline{\underline{@}}, \triangleq, \uparrow, \downarrow, !, \oslash, \boxtimes, \S\} \tag{10.64}$$

According to Theorem 4.7, the set of 17 algebraic process operations $\mathfrak{R}$ in computing have been defined in Section 4.6.5 as follows:

$$\mathfrak{R} =$$

$$\{\rightarrow, \curvearrowright, |, |\ldots|, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \|, \oiint, |||, \text{»}, \text{↯}, \hookrightarrow_t, \hookrightarrow_e, \hookrightarrow_i\} \tag{10.65}$$

The syntaxes and semantics of each of the above meta processes and algebraic process operations may be referred to Sections 4.6, 5.4, and 6.6, respectively. Formal descriptions of system behaviors in RTPA may be referred to Sections 4.8, 5.5, and Appendix K. In addition, formal description of human behaviors in RTPA may be referred to the examples in Fig. 9.16 and Fig. 11.10.

10.5.3 WORK DONE BY SYSTEMS

This subsection introduces the concept of abstract work done by a system that is an extension and generalization of the concept of work in kinematics, electricity, and thermodynamics.

Definition 10.65 The *abstract work* done by a system *S*, *W*(*S*), is its output of utility *U* in term of the implemented number of functions *F*, i.e.:

$$W(S) = U \quad [\mathrm{F}] \tag{10.66}$$

where the functions of *U* can be perceived as energy spent in Joule in physical systems, information generated or processed in bit in intelligent systems, or tasks conducted in person-hour in human-based systems.

Definition 10.66 The *power* of a system *S*, *P*(*S*), is the work done by the system per unit time, i.e.:

$$\begin{aligned} P(S) &= \frac{W(S)}{t} \\ &= \frac{U}{t} \ [\mathrm{F/hr}] \end{aligned} \tag{10.67}$$

Definition 10.67 The *efficiency* of a system η is the ratio between the average output work $\overline{W}_o$ and the average input work $\overline{W}_i$ of the system, i.e.:

$$\eta = \frac{\overline{W}_o}{\overline{W}_i} \bullet 100\% \tag{10.68}$$

Definition 10.68 The *overhead* of a system ϖ is the ratio of average internal loss of work in the system, i.e.:

$$\begin{aligned} \varpi &= 100\% - \eta \\ &= (1 - \frac{\overline{W}_o}{\overline{W}_i}) \bullet 100\% \end{aligned} \tag{10.69}$$

where η and ϖ are complementary, i.e.: $\eta + \varpi = 100\%$.

Corollary 10.11 The *ideal system utility* states that no system *S* may reach the efficiency $\eta(S)$ that equals to or greater than 100%, i.e.:

$$\nexists S \Rightarrow \eta(S) \geq 100\% \tag{10.70}$$

The 36th Law of Software Engineering

Theorem 10.6 The *system gain of work* states that work done by a system is always greater than any of its components, but must not greater than the sum of those of its components, i.e.:

$$\begin{cases} W(S) \leq \sum_{i=1}^{n} W(C_i), & \eta \leq 100\% \\ W(S) > \max(W(C_i)), & C_i \in E_S \end{cases} \tag{10.71}$$

Eq. 10.71 indicates that, although a system's capability to carry out a work is more powerful than any of its components, the total work done by the system cannot exceed the sum of all its components because the existence of the overhead $\varpi > 0$, or no system or its components may reach an efficiency $\eta \geq 100\%$.

There was a myth on an ideal system in conventional systems theory that supposes the work done by the ideal system *W*(*S*) may be greater than the sum of all its components $W(e_i)$, i.e., $W(S) \geq \sum_{i=1}^{n} W(C_i)$. According to Theorem 10.6 and Corollary 10.11, this ideal system utility is impossible to achieve.

10.5.4 THE MAXIMUM OUTPUT OF SYSTEMS

The output work of systems is dependent on the architectural types of the systems. The basic structures of system compositions are serial, parallel, and hybrid as analyzed below.

Definition 10.69 A *serial system* S_s is a specially structured system that can be described as a pair, i.e.:

$$S_s = (C_s, R_s) \tag{10.72}$$

where C_s is a set of components, i.e., $C_s = (C_1, C_2, \ldots, C_n)$, and R_s is a serial relation between all components, i.e., $R_s = C_1 \wedge C_2 \wedge \ldots \wedge C_n$.

The 34th Principle of Software Engineering

Theorem 10.7 The *bottleneck principle of systems* states that the output work of a serial system $W(S_s)$ is determined by the least powerful component of the system, i.e.:

$$W(S_s) = min\ (\mathrm{W}(C_i\ ) \mid C_i \in C_s \wedge 1 \leq i \leq n)) \tag{10.73}$$

The bottleneck principle can be described by the *bucket effect* of serial systems, which states that the capacity of a bucket is determined by the shortest piece that the bucket is made of. This is a vivid example to explain the behavior of serial systems.

Definition 10.70 A *parallel system* S_p is a specially structured system that can be described as a pair, i.e.:

$$S_p = (C_p, R_p) \tag{10.74}$$

where C_p is a set of components, i.e., $C_p = \{C_1, C_2, \ldots, C_n\}$, and R_p is a parallel relation between all components, i.e., $R_p = C_1 \vee C_2 \vee \ldots \vee C_n$.

The 35th Principle of Software Engineering

Theorem 10.8 The *linear sum principle of systems* states that the output work of a parallel system $W(S_p)$ is a sum of the work done by all its components less the overhead ϖ of the system, i.e.:

$$W(S_p) = \sum_{i=1}^{n} W(C_i) - \varpi, \quad C_i \in C_p,\ \varpi > 0 \qquad (10.75)$$

A *complex system* with hybrid architectures of serial and parallel components can be analyzed separately by the sum of a number of parallel subsystems of serial components, or a number of serial subsystems of parallel components.

10.5.5 SYSTEM EQUILIBRIUM AND ORGANIZATION

A set of components may form a system because of their coherent organization towards a common goal of the system. Self-organization is a universal property of systems with the characteristic of equilibrium.

In *Principles of the Self-Organizing System* (1962), Ashby wrote:

> "We start with the fact that systems in general go to equilibrium. Now most of a system's states are non-equilibrial … . So in going from any state to one of the equilibria, the system is going from a larger number of states to a smaller. In this way, it is performing a selection, in the purely objective sense that it rejects some states, by leaving them, and retains some other state, by sticking to it. "

10.5.5.1 The Generic IPO Model of Systems

The functional architecture of any open system can be described by its input, output, and the internal behaviors in terms of processes of the system, *IPO*, as shown in Fig. 10.13.

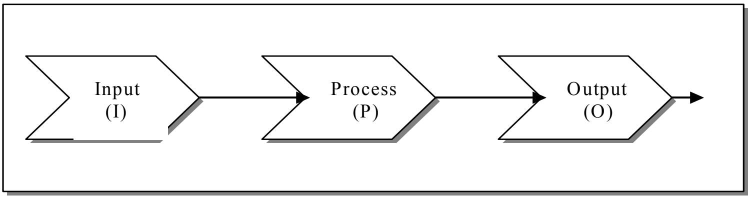


Figure 10.13 The generic IPO system model

The architecture of a system with feedback, IPO_f, is shown in Fig. 10.14. The feedback for a system can be positive or negative. The former, IPO_{f+}, is a self-stimulated system; while the latter, IPO_{f-}, is a self-regulated system for autonomously maintaining system equilibrium and self-organization.

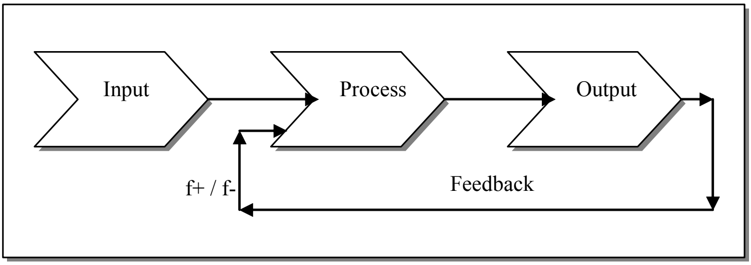


Figure 10.14 The positive/negative feedback systems $\text{IPO}_{\text{f+}}$ or $\text{IPO}_{\text{f-}}$

Definition 10.71 The *negative feedback* of a system is a feedback that is proportional to the output of the system and its effect is to reduce or regulate the aggregative tendency of the system.

Feedback is a universal phenomenon that exists not only in physical systems, but also in advanced systems such as biological, physiological, economical, and social systems. The observation of negative and positive feedback among neurons in the brain via synapses may be referred to [Smith, 1993; Kotulak, 1997; Pinel, 1997; Rosenzmeig et al., 1999]. The effect of negative feedback in economics will be discussed in Section 12.2.2 on economic equilibriums.

10.5.5.2 Laws of System Equilibrium and Organization

It is empirically observed that system equilibriums and self-organization exist when the negative feedback of a system is proportional to its aggregative effects in the system.

Definition 10.72 The *equilibrium of a system* is a stable state where the effects of all components in terms of their abstract work form a zero-sum, i.e.:

$$W(S) = \sum_{i=1}^{n} W(C_i) = 0 \tag{10.76}$$

where $W(S)$ is the total work done by the system, and such a system is called a *zero-sum system.*

Example 10.7 The following phenomena are system equilibriums in different disciplines:

a) *Newton's 1st law in* kinematics: The sum of all work done by forces *F* in a circle of movement *d* is zero, i.e.:

$$\sum_{i=1}^{n} F_i d_i = 0 \tag{10.77}$$

b) *Energy conservation*: The sum of all forms of energy *E* in a system is zero, i.e.:

$$\sum_{i=1}^{n} E_i = 0 \tag{10.78}$$

c) *Kirchhoff's rule in electricity*: The sum of all potentials *P* in a closed circuit system is zero, i.e.:

$$\sum_{i=1}^{n} P_i = 0 \tag{10.79}$$

d) *Economic equilibrium*: The effect of all demands *D* and supplies *S* on the price *P* in a market is a zero-sum, i.e.:

$$\sum_{i=1}^{n} (P_i(D) + P_i(S)) = 0 \tag{10.80}$$

The 37th Law of Software Engineering

Theorem 10.9 *Conservative work of equilibrium systems* states that the sum of all types of work is always zero in an equilibrium system, i.e.:

$$\sum_{i=1}^{n} W(C_i) = 0 \tag{10.81}$$

where $W(C_i)$ is the abstract work of a system component C_i.

Definition 10.73 *System organization* is a process to configure and manipulate the system approaching to a stable and ordered state with an internal equilibrium.

System organizations can be classified as self-organization or hetero-organization. The mechanisms of system self-organization are analyzed below.

Definition 10.74 Let $f(x)$ be a continuous and deferential function of a system defined in a domain [a, b]. Then, the *minimum* of $f(x)$, $f_{min}(x)$, satisfies the following condition:

$$f(x) - f_{min}(x) > 0,\ \ \mathrm{x}, x_{min} \in [\mathrm{a, b}],\ \mathrm{x} \neq x_{min} \qquad (10.82)$$

There may be multiple minimums for a given function $f(x)$ in different subdomains of $[a, b]$, such as $f_{min}(x_1 \mid x_1 \in (a_1, b_1))$, $f_{min}(x_2 \mid x_2 \in (a_2, b_2))$, and $f_{min}(x_k \mid x_k \in (a_k, b_k))$. Among those, the minimum of minimums is called the *global* minimum, $f_{min}(x_g)$, i.e.:

$$f_{\min}(x_g \mid x_g = \min(x_i), 1 \le i \le k) \qquad (10.83)$$

The remainders are called *local* minimums.

The 38th Law of Software Engineering

Theorem 10.10 The *condition of self-organization* states that the *necessary* and *sufficient* condition of self-organization is the existence of at least one minimum on the state curve of a system $f(x)$, which satisfies the following requirements:

$$\begin{cases} f'(x_{min} \mid x_{min} \in \ (a,b)) = 0 \\ f''(x_{min} \mid x_{min} \in \ (a,b)) \neq 0 \end{cases} \qquad (10.84)$$

Or equivalently

$$\begin{cases} f'(x_{min} \mid x_{min} \in \ (a,b)) = 0 \\ f''(x \mid x < x_{min} \in \ (a,b)) < 0 \\ f''(x \mid x > x_{min} \in \ (a,b)) > 0 \end{cases} \qquad (10.85)$$

where $f'(x)$ and $f''(x)$ are the first and second order derivatives of $f(x)$ on (a, b).

Since negative feedback is the only means to regulate the states of a system, the following conclusions can be derived.

Corollary 10.12 The *functional condition* of self-organization system is the existence of the negative feedback mechanism that is proportional to the incremental or aggressive effects of the system.

10.5.6 SYSTEM SYNCHRONIZATION AND COORDINATION

A system reaches the maximum output when it is synchronized over time, unified on a common goal, or coordinated among individual efforts. This assertion can be illustrated in Fig. 10.15, where $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ denote the vectors of efforts or work done by components S_1 and S_2 in the system S.

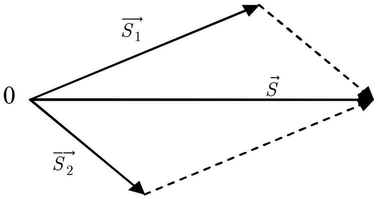


Figure 10.15 System synchronization and coordination

The 39th Law of Software Engineering

Theorem 10.11 *System synchronization* states that a system reaches its maximum utility $\vec{S}_{\max}$ when all components' efforts $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ are synchronized, i.e.:

$$\begin{cases} \vec{S} = \overrightarrow{S_1} + \overrightarrow{S_2} \\ \vec{S}_{\max} = \ | \overrightarrow{S_1} | + | \overrightarrow{S_2} | \end{cases} \tag{10.86}$$

The synchronization principle described in the 39th Law proves from another angle that the work of a system may not exceed the sum of its components as revealed in Theorem 10.6.

Corollary 10.13 A dynamic system tends to synchronize on a certain state where it is *stable* or *dynamically equilibrial* that satisfies one of the following conditions:

a) The *rational* condition: The apparent best equilibrium condition; or
b) The *arbitrary* condition: The first meet or most conventional equilibrium condition.

Science and engineering methodologies are rational because they usually adopt the Category (a) conditions in Corollary 10.13 when the constraints and all possible solutions are known. However, social and psychological systems often adopt a Category (b) condition when there is no rational or apparent optimal solution available. The latter is also in line with the basic axiom of science, *minimizing energy consumption*, in searching rational or arbitrary solutions.

10.5.7 SYSTEM DISSIMILATION

Dissimilation is a universal property of any system such as physical, economic, living, or social systems. According to system topology, there are maintainable and nonmaintainable systems. The properties of dissimilation of both types of systems are analyzed in the following subsections, respectively.

10.5.7.1 Dissimilation of Nonmaintainable Systems

Definition 10.75 *Dissimilation* is the tendency that a system undergoes an apparent or hidden destructive change against its original purposes or designed functions.

System dissimilation can be analyzed on the basis of how systems maintain their functional availability and against the loss of it.

Definition 10.76 The *availability of a system* $\alpha(t)$ can be the designed utility, function, efficiency, or reliability of the system.

Definition 10.77 The *dissimilation of a nonmaintainable system* D_{nm} is determined by its degradation of availability over time during its *lifecycle T*, i.e.:

$$D_{nm} = k\,(1 - e^{t-T}),\ 0 \leq t \leq T \qquad (10.87)$$

where *k* is a positive constant called the initial availability of the system.

Definition 10.78 The *rate of dissimilation* δ of a nonmaintainable system can be derived as follows:

$$\begin{aligned}\delta &= \frac{dD_{nm}(t)}{dt} \\ &= \frac{dk(1 - e^{t-T})}{dt} \\ &= -ke^{t-T},\ 0 < t \le T\end{aligned} \tag{10.88}$$

The trend of dissimilation of a nonmaintainable system D_{nm} and its rate δ are shown in Fig. 10.16, where the curves are normalized with k = 1. The unit of δ uses a different scale from that of D_{nm} in order to better plot the lower curve.

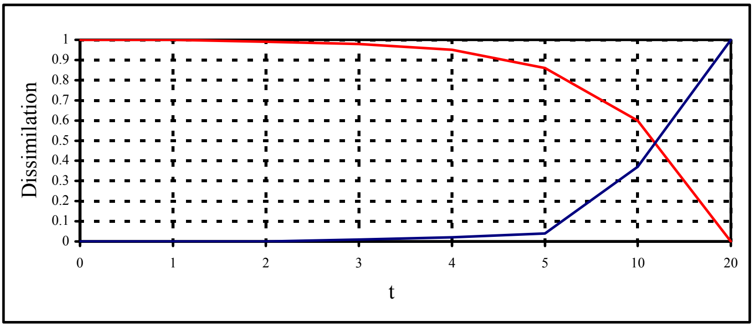


Figure 10.16 System dissimilation (nonmaintainable system, *T* = 20)

It can be seen in Fig. 10.16 that a system is exponentially dissimilating during its lifecycle, and the rate of dissimilation reaches the maximum in the last period of its lifecycle.

The 40th Law of Software Engineering

Theorem 10.12 *System dissimilation* states that any system tends to undergo a continuous degradation that leads to the eventual loss of its designed utility and against its initial purposes to form the system.

Theorem 10.12 indicates that any concrete system has a certain lifecycle, in which dissimilation is being undergone since it has been put into operation.

Corollary 10.14 The *most critical period of system dissimilation* is its exiting period. During this period the rate of dissimilation of the system will be exponentially increased.

When considering the development or building period as part of the lifecycle of a system, the dissimilation of a nonmaintainable system during this period is negative, because of the continuous effort of development.

Definition 10.79 The *dissimilation of a nonmaintainable system* during the development period D'_{nm} is determined as follows:

$$D'_{nm} = k\,(1 - e^{-t-T'}),\ \ T' \le t \le 0 \qquad (10.89)$$

where T' may be different from T.

Therefore, in the entire lifecycle of a nonmaintainable system, $T' + T$, the trend of dissimilation is shown in Fig. 10.17, where $t = 0$ is the time that the system is put into operation.

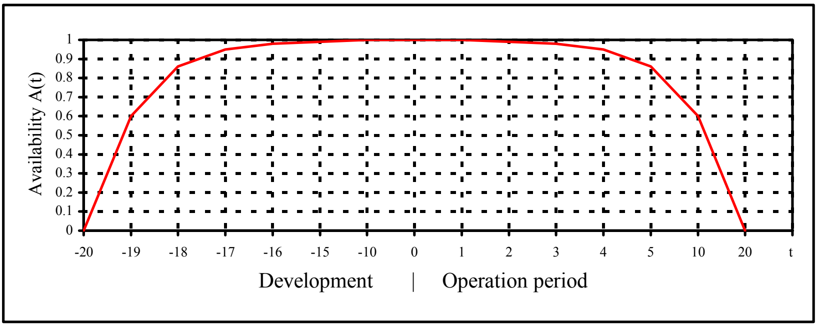


Figure 10.17 The entire lifecycle of system dissimilation (nonmaintainable system, $T = T' = 20$)

10.5.7.2 Dissimilation of Maintainable Systems

The dissimilation property of maintainable systems can be derived based on those of nonmaintainable systems, if the maintenance effect is treated as a recovery of the original availability of the system. Therefore, the dissimilation during the whole lifecycle of a maintainable system is given in Fig. 10.18.

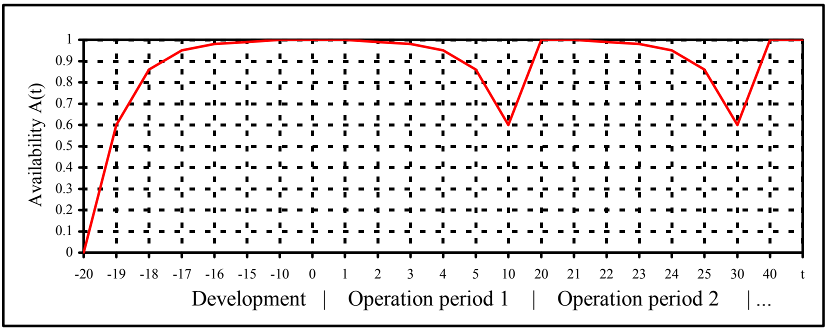


Figure 10.18 The entire lifecycle of system dissimilation (maintainable system, $T = 20$)

Definition 10.80 The *dissimilation of a maintainable system* D_m is determined as follows:

$$D_m = \begin{cases} k\,(1 - e^{-t-T'}), & T' \le t < 0 \\ k\,(1 - e^{t-T}), & nT \le t < (n+1)T, n \ge 0 \end{cases} \quad (10.90)$$

where T is the *maintenance cycle*, and n is the *number of operation period*.

System dissimilation may be considered as an inversed process of system fusion as described in Section 10.5.1. The property of system dissimilation can be used to explain a wide range of phenomena in systems, such as system availability, efficiency, reliability, and the trends of systems during the entire lifecycle. For example, the discovery of the phenomena of *software maintenance crisis* in software development organizations [Wang, 2005d], which will be described in Sections 12.6.5 and 14.3.3, is a direct application of the principle of system dissimilation.

10.6 Software System Engineering

Software systems are a category of the most complicated systems in the abstract world interacting with the concrete-world. Therefore, software engineering is an important discipline that may be used to test the theories of system science as developed in the previous sections.

This section describes the abstract model of computing systems and the hierarchical model of software systems, as well as corresponding work products in software engineering. The ISO/IEC 15288 system engineering model for software engineering is introduced. Then, a set of common and frequently observed phenomena in software engineering is explained as typical system engineering issues. Software system complexity as well as cohesion and coupling will be formally modeled in Section 10.7.

10.6.1 THE ABSTRACT MODEL OF COMPUTING SYSTEMS

According to Definition 5.65, the abstract model of a generic computing system, GCS, has been modeled in Section 5.6.1 as the common architecture of operating systems. The GCS *system*, §, is an abstract logical model of the executing platform denoted by a set of parallel computing resources as given in Eq. 5.87.

The abstract computing system as defined in Eq. 5.87 can be illustrated in Fig. 10.19, where the GCS § controls all the computing resources of an abstract target machine. The system is logically abstracted as a set of processes and underlying resources, such as memory, ports, and the system clock. A process is dispatched and controlled by the system §, which is triggered by various external, timing, or interrupt events [Wang, 2005*l*].

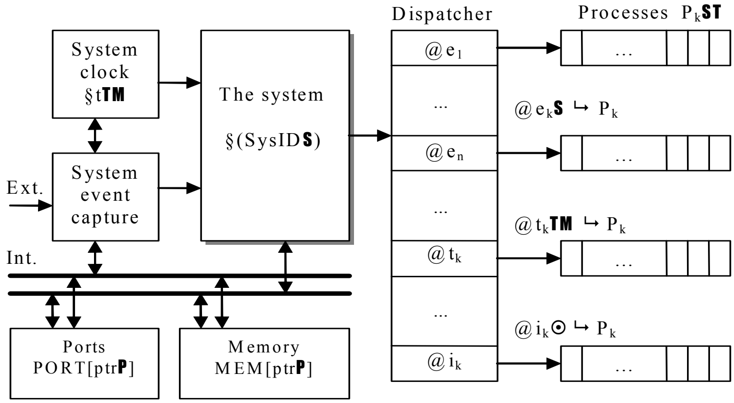


Figure 10.19 The abstract model of the Generic Computing System (GCS)

10.6.2 THE HIERARCHICAL MODEL OF SOFTWARE SYSTEMS

According to Theorem 10.2, the hierarchical architecture is one of the universal system principles that software system architectures obey as well. Actually, abstract systems need much more explicit architectural design and description than that of concrete systems, because the former are inexplicit and nonself-expressive.

10.6.2.1 The Hierarchical Structure of Software Systems

A hierarchical structure of software systems [Wang, 2005*l*] can be described in Fig. 10.20. The generic layered model shows that a software system can be decomposed from top down at seven levels known as those of the system, subsystem, component (class, or pattern), function (method), statement, data model (structure), and target code. The layered decomposition of software systems can be perceived as a stepwise refinement process that transfers a system into target code.

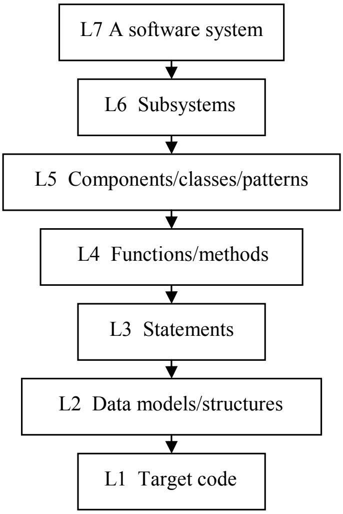


Figure 10.20 The layered model of software systems

10.6.2.2 The Hierarchical Structure of Software Engineering Processes and Work Products

As that of software system architectures, software engineering processes and their work products can also be described by a hierarchical system model. The layered system model of software engineering processes and corresponding work products and documentation [Wang, 2005*l*] are shown in Fig. 10.21. The clarification of the work products or results of each layered process are helpful to establish job expectations, quality standards, and process transition criteria in software engineering organization.

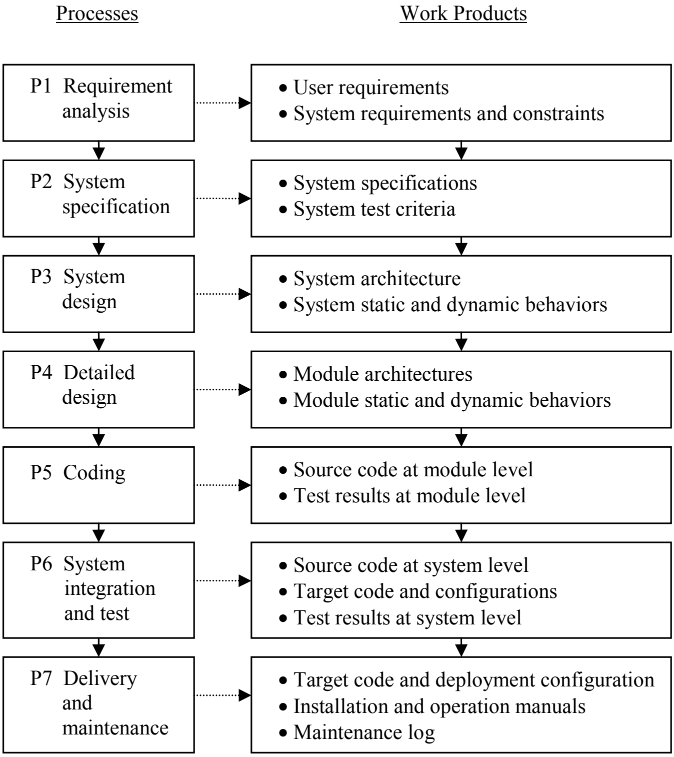

Processes
Work Products
P1 Requirement analysis
• User requirements
• System requirements and constraints
P2 System specification
• System specifications
• System test criteria
P3 System design
• System architecture
• System static and dynamic behaviors
P4 Detailed design
• Module architectures
• Module static and dynamic behaviors
P5 Coding
• Source code at module level
• Test results at module level
P6 System integration and test
• Source code at system level
• Target code and configurations
• Test results at system level
P7 Delivery and maintenance
• Target code and deployment configuration
• Installation and operation manuals
• Maintenance log

Figure 10.21 Work products of software engineering processes

10.6.3 THE ISO/IEC 15288 SYSTEM ENGINEERING MODEL FOR SOFTWARE ENGINEERING

It has been seen in the previous subsections that software systems are highly complex abstract systems that need to adopt system theories in their entire lifecycle such as design, modeling, implementation, and maintenance. This section presents a paradigm of software system models, the ISO/IEC 15288 System Engineering Model of Software Engineering [ISO/IEC, 1999].

ISO 15288 describes a software development organization by a four-level hierarchical process model known as the processes of enterprise, project, technology, and agreement. Within each process, there are a number of defined activities as shown in each column of Table 10.3.

Table 10.3
The ISO/IEC 15288 System Engineering Model of Software Engineering

No	Enterprise processes	Project processes	Technical processes	Agreement processes
1	Enterprise environment management	Project planning	Stakeholder requirements definition	Acquisition
2	Investment management	Project assessment	Requirements analysis	Supply
3	System lifecycle management	Project control	Architectural design	
4	Resource management	Decision making	Implementation	
5	Quality management	Risk management	Integration	
6		Configuration management	Verification	
7		Information management	Transition	
8			Validation	
9			Operation	
10			Maintenance	
11			Disposal	

In ISO/IEC 15288, the behavioral processes of a software development organization can be classified as *intra-* and *inter-organizational processes*. The relationship between these two types of organizational processes is illustrated in Fig. 10.22 [ISO/IEC, 1999].

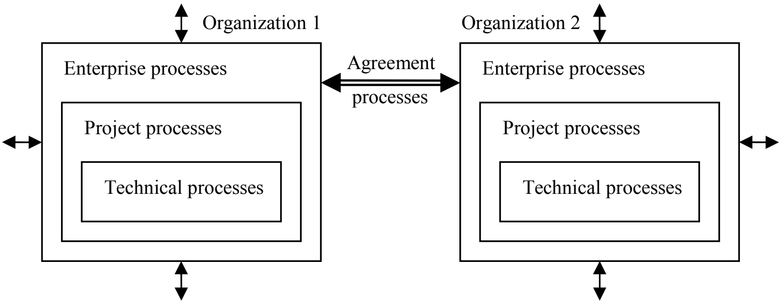


Figure 10.22 The system model of intra- and inter-organization processes

ISO/IEC 15288 provides a generic process model that treats a software development organization and its software engineering processes as a system. ISO/IEC 15288 provides an extended process system for software engineering, which encompasses not only conventional software processes, but also system processes integrating an extended scope of stakeholders involved in software engineering.

10.6.4 SOFTWARE ENGINEERING PHENOMENA AS SYSTEM ENGINEERING PROBLEMS

Systems theories and philosophies are considered as a cure to many unwise practices in software engineering [Wang, 2005*l*]. Some typical phenomena in software engineering practice that are typical system engineering problems are presented below.

a) *New is beautiful:* This is a software engineering phenomenon that practitioners chase anything claimed *new* and *fancy* in practice, such as programming languages, models, tools, and environments. This shows no confidence in the existing methodologies and tools, no patience to get familiar with the existing knowledge, and no awareness of necessary domain knowledge built-up requirements. From the system science point of view, this practice ignores engineering experience accumulation needs and wastes resources on various languages where there is no theoretical difference in their descriptive power for programming.

b) *Fundamental research left behind industrial practices:* Scientists in software engineering are busy explaining a great many complicated phenomena in a numerals different techniques. A few researchers concentrate on the fundamental theories and the order of knowledge for software engineering. This allows software engineering probably to be the first engineering discipline that is led by practitioners rather than scientists.

c) *Overlooked coordinative work organization as the key software engineering technology:* The whole spectrum of the software engineering system encompasses technology, organization, and management. According to Theorem 8.7, it is recognized that the coordinative work organization theory is the top-level technology for software engineering. Without it, pure focuses on technical issues in software engineering such as programming languages and testing tools would lead to a significant loss of effort, resources, and time into the black hole of inappropriate engineering project organization. Therefore, emphases must be put on the systematical view towards software engineering constraints and solutions, before any detailed technical decisions are made.

d) *Product lifespan is too short:* Nobody prepared for a durable design and implementation that may last for a few decades, if not for a few hundred years. It was reported that the current owner of a historical building in Shanghai received a letter for its 100th anniversary survey from a UK construction company in 1998. The reaction of the owner was astonished: 'Somebody had still remembered us!' Even though the owners of the building had changed so many times, the architects still kept the record of the entity and their responsibility. Is this a shame on the software engineering community? Can one find all design and testing documents for a system conducted ten years ago? Is it ethical when a vendor tries to stop supporting existing systems, environments, or languages that numerous legacy systems rely on?

e) *Local maximum is often adopted:* According to systems theory, optimization of a component of a system may not improve the performance of the whole system. Instead, it is even harmful for the system. For instance, a fax control system consists of the fax machine, transmission lines, switching systems, signaling systems, code error monitoring, and the flow control software. The system's performance would get worse when the transmission speed is increased. Instead, the fax system may reach a better performance by decreasing the speed of the modem. Therefore, global tradeoffs in a software system should always be maintained.

f) *Pentium inside?* Most PCs nowadays have a label on it – Pentium inside! This is simply because one cannot not feel too much difference in speed and performance whatever if there is a Pentium processor inside or not! Although the CPU speed in a PC has improved 10^4-10^6 times in the last decades, the speed of external buses, peripheral devices, and communication modems have not been significantly matched up. As a result, the processing speed of PCs as a system has not been enhanced significantly. Sometime, it has even been getting worse when using the scroll bar or marking multiple pages, it may be too fast to be able to get the pages stopped on the place that one wishes.

g) *Views on software systems – pessimism vs. optimism:* Almost all doctors are astonished that human beings are relatively much more robust than biological and physiological principles may suggest. Fortunately, so are software systems. In principle, a software system may go wrong by only a single bit error, and for some of them the programmers have no control because they are dependent on run-time situations, such as dynamic memory allocations and external interferences. However, there are still more and more software systems, especially mission-critical ones, running correctly day and night via systematic exception detection, handling, and fault-tolerant techniques.

h) *The software maintenance crisis:* Software maintenance crisis is an inherited type of software crisis that happens when the demand for legacy software maintenance largely exceeded the capability that the software industry can provide, or when the costs of legacy software maintenance predominantly override the investment in new software development. Detailed description and solutions on software maintenance crisis may be referred to Section 14.3.3. [Wang, 2005d].

i) *Synchronization by process-based software engineering:* Software engineering may adopt the principle of system synchronization in its organizational infrastructure, known as process-based software engineering. Detailed techniques and explanations will be proved in Section 14.2.2.

j) *Measuring the tendency of programmers:* The productivity of software engineering is conventionally measured based on the symbolic size of programs as defined below.

Definition 10.81 The *productivity* ρ of software development is determined by the symbolic size of programs S_s developed per person p per month t, i.e.:

$$\rho = \frac{S_s}{p \bullet t} \text{ [LOC/PM]} \tag{10.91}$$

where the unit of S_s is LOC, and the unit of $p \bullet t$ is person-month (PM).

Since the above simple absolute measure of ρ assumes a philosophy of the higher the better, programmers are encouraged to develop larger physical sized program or longer code for a certain required function. This is a typical practice that results from an imbalanced measurement system in software engineering. Just like Tom Clancy wrote about the ancient Roman bridges:

> "The Roman bridges of antiquity were very inefficient structures. By modern standards, they used too much stone, and as a result, far too much labor to build. Over the years we have learned to build bridges more efficiently, using fewer materials and less labor to perform the same task."

Therefore, a countermeasure known as coding efficiency as defined below needs to be adopted supplemented to the symbolic size measure in software engineering.

Definition 10.82 *Coding efficiency e* in software implementation is measured by the ratio of functional size S_f and symbolic size S_s, i.e.:

$$e = \frac{S_f}{S_s} \text{ [FO/LOC]} \tag{10.92}$$

where the functional size of software S_f is equivalent to the cognitive complexity of software C_c in the unit of function-object (FO), which will be presented in Section 10.7.3.

When the measure of implementation efficiency is introduced into the software engineering processes, the tendency of over-sized software, so called *fat-ware*, in software engineering can be effectively rectified [Wang and King, 2000a; Wang, 2003f].

The discussions in this section show that system theories and philosophies are an ideal cure to many unwise practices in software engineering that were not conform with fundamental system principles. Therefore, system science and engineering methodologies [Hall, 1967; Bate et al., 1993; Harauz, 1997; Wang, 2006d/2007d] will play an increasingly significant role in software engineering.

10.7 The Complexity Theory of Software Systems

Applying the system complexity theories developed in Section 10.3.3, the symbolic and functional complexities of software, as well as cohesions and couplings of software systems can be formally analyzed using a set of system level measurements in software engineering.

Although computational complexities, particularly *algorithm complexities*, are one of the focuses in computer science, software engineering is particularly interested in the *functional complexity* of large scale and real-world software systems.

The computational complexity of algorithms puts emphases on the computability and efficiency of typical algorithms of massive data processing and high throughput systems, in which computing efficiency is dependent on and dominated by the problem sizes in terms of their number of inputs such as those in sorting and searching. However, there are more generic computational problems and software systems that are not dominated by this kind of input sizes rather than by internal architectural and operational complexities such as problem solving and process dispatching. This shows the differences of focuses or problem models in the complexity theories of software engineering and conventional computing.

The 36th Principle of Software Engineering

Theorem 10.13 The *orientation of software engineering complexity theories* states that the complexity theories of computation and software engineering are different. The former is focused on the problems of *high throughput complexity* that are computing *time efficiency* centered; while the latter puts emphases on the problems of *functional complexity* that are human *cognition time* and *workload* oriented.

In software engineering, a problem with very high computational complexity may be quite simple for human comprehension, and vice versa. According to cognitive informatics, human beings may comprehend a large cycle of iteration, which is the major issue of computational complexity, by looking at only the beginning and termination conditions, and one or a few arbitrary internal loops with inductive inferences. However, humans are not good at dealing with functional complexities such as a long chain of interrelated operations, very abstract data objects, and their consistency. Therefore, the system complexity of large-scale software is the focus of software engineering.

10.7.1 COMPUTATIONAL COMPLEXITY

Computational complexity theory is a well established area in computing [Hartmanis and Stearns, 1965; Hartmanis, 1994; Lewis and Papadimitriou, 1998] that studies: a) The taxonomy of problems in computing and their solvabilities; and b) Complexities and efficiencies of algorithms for a given problem. Computational complexity centered by the algorithm complexity can be modeled by its *time* or *space* complexity, particularly the former, proportional to the sizes of problems.

10.7.1.1 Taxonomy of Computational Problems

Computational complexity theories study the solvability in computing. The *solvable problems* are those that can be computed by *polynomial-time* consumption. The *nonsolvable problems* are those that cannot be solved in any practical sense by computers due to excessive time requirements.

The taxonomy of problems in computation can be classified into the following classes. The class of solvable problems that are *polynomial-time* computational by *deterministic* Turing machines are called the *class P* problems. The class of problems that are polynomial-time computational by *nondeterministic* Turing machines are called the *class NP* problems. The class of problems that their answers are complementary to the NP problems are called the *NP complementary* (coNP) problems. The subclass of NP problems, which serves as a meta problem where other NP problems may be reduced to them in polynomial time, is called the *NP-complete* (NPc) problems. There is a special class of problems that can be reduced to known NP problems in polynomial time, which are usually referred to as the *NP-hard* (NPh) problems [Hartmanis and Stearns 1965; McDermid, 1991; Lewis and Papadimitriou, 1998].

The relationship among various classes of problems in computation can be illustrated as shown in Fig. 10.23. It is noteworthy that there are certain problems that are unsolvable or with unknown solvability in computing, but they might be solvable by human brains.

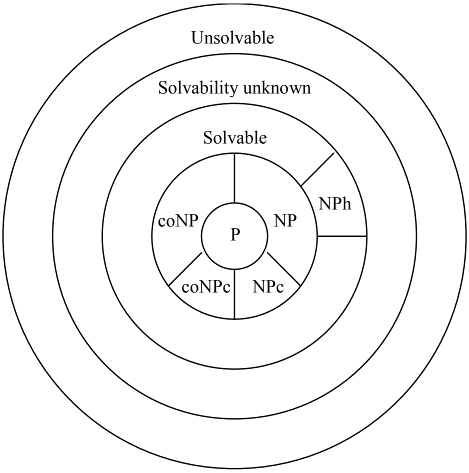


Figure 10.23 Taxonomy of problems in computing

10.7.1.2 Time Complexity of Algorithms

The time complexity of an algorithm for a given problem is measured in software engineering as an estimation of its computational complexity. The time complexity of an algorithm can be estimated by analyzing the number of dominant operations in the algorithm, where each of the dominant operations is assumed to take an identical unit of time in operation.

Definition 10.83 The *dominant operations* in an algorithm are those statements within iterative structures that are proportional to the size of the problem or the number of inputs n of the algorithm.

Definition 10.84 For a given function $f(x)$, its *asymptotic function* $f_a(x)$ is a function that satisfies:

$$|f(x)| \leq k|f_a(x)|, x > b \tag{10.93}$$

where k and b are a positive constant.

Definition 10.85 If a function $f(x)$ has an asymptotic function $f_a(x)$, the function $f(x)$ is said to be of *order of* $f_a(x)$, denoted by:

$$f(x) = O(f_a(x)) \tag{10.94}$$

where O is known as the *big O* notation.

Definition 10.86 For a given size of a problem n, the *time complexity* $C_t(n)$ of an algorithm for solving the problem is a function of the *maximum* required number of *dominant operations* $O(f_a(n))$, i.e.:

$$C_t(n) = O(f_a(n)) \tag{10.95}$$

where $f_a(n)$ is called the *asymptotic function* of $C_t(n)$.

It is noteworthy in Definition 10.86 that the maximum number of dominant operations $C_t(n)$ indicates the *worst case* scenario. An *average case* complexity is a mathematical expectation of $C_t(n)$.

Example 10.8 According to Definition 10.86, the time complexity $C_t(n)$ of the following functions $f_1(n)$ through $f_4(n)$ can be estimated as follows:

- $f_1(n) = 5n^3 + 2n^2 - 6 \quad \Rightarrow C_{t1}(n) = \mathrm{O}(f_{a1}(n)) = \mathrm{O}(n^3)$
- $f_2(n) = 3n \quad \Rightarrow C_{t2}(n) = \mathrm{O}(f_{a2}(n)) = \mathrm{O}(n)$
- $f_3(n) = 4\log_2 n + 10 \quad \Rightarrow C_{t3}(n) = \mathrm{O}(f_{a3}(n)) = \mathrm{O}(\log_2 n)$
- $f_4(n) = 2 + 8 \quad \Rightarrow C_{t4}(n) = \mathrm{O}(f_{a4}(n)) = \mathrm{O}(\varepsilon)$

where ε is a positive constant.

Typical asymptotic functions of algorithm and programs are shown in Fig. 10.23, where the computational loads in term of processing time of different functions may grow polynomially or exponentially as the size of problem n, usually the number of input items, increasing. If an algorithm can be reduced to a type of function with polynomial complexity, it is always a computable problem; otherwise, it would be a very hard problem, particularly in the worst case when n is large.

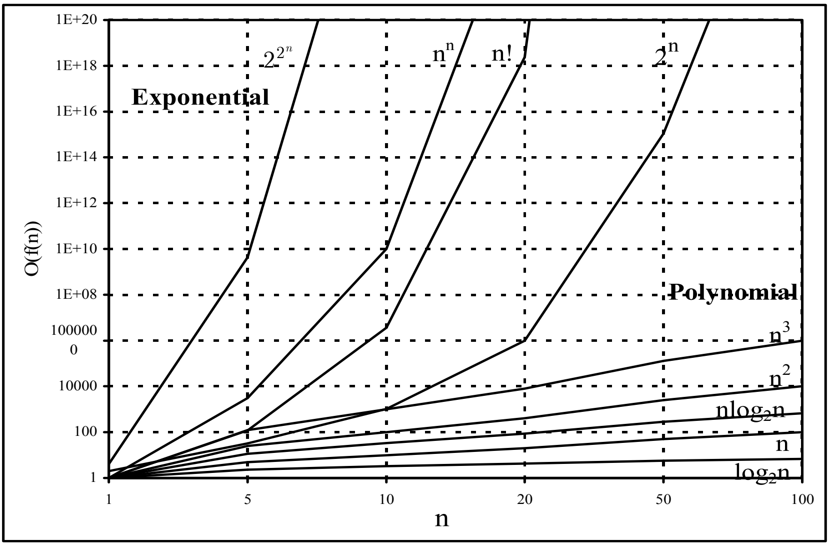


Figure 10.24 Typical asymptotic functions of software time complexities

10.7.1.3 Space Complexity of Algorithms

Definition 10.87 The *space complexity* of an algorithm for a given problem is the maximum required space for both working memory w and target code memory o, i.e.:

$$\begin{aligned} C_m(n) &= O(f(w+o)) \\ &\approx O(f(w)) \end{aligned} \qquad (10.96)$$

where w refers to the memory for data objects under processing such as input/output and intermediate variables, and o refers to the memory for executable code.

Because the target code memory is static and determinable, algorithm space complexity is focused on the dynamic working memory complexity.

10.7.2 SYMBOLIC AND CONTROL FLOW COMPLEXITIES

Software system complexities may also be measured by two popular and simple methods known as the *symbolic complexity* [Wang, 2001d/03f] and the *control flow complexity*. The former is also known as the *Lines of Code* (LOC) [Halstead, 1977; Albrecht and Gaffney, 1983], while the latter is also called the McCabe *cyclomatic complexity* [McCabe, 1976].

10.7.2.1 Symbolic Complexity of Software Systems

The most simple and direct forward complexity measure of software systems is the symbolic complexity that can be represented by the number of lines of statements in a programming language.

Definition 10.88 The *symbolic complexity* of a software system *S,* $C_s(S)$*,* is the linear length of its static statements measured in the unit of Lines of Code (LOC), i.e.:

$$C_s(S) = \sum_{k=1}^{n_c} C_s(k) \quad [\text{LOC}] \tag{10.97}$$

where C_s(k) represents the symbolic complexity of component *k* in *S*.

In the measure of symbolic complexity of software, variables, data objects declarations, and comments are not considered as a valid line of instructions, and an instruction separated in multiple lines is usually counted as a single line.

10.7.2.2 Control Flow Complexity of Software Systems

Another approach to measure the code complexity of software systems can be via its control structures based on the Control Flow Graph (CFG) as described in Section 5.4.1. When a program is abstracted by a CFG, well-defined graph theory may be used to analyze its complexity properties. In the remainder of this section, *Euler's theorem* is introduced and how it is used to model the complexity of CFGs is described. Then, the relationship between Euler's theorem and McCabe *cyclomatic complexity* is discussed.

Lemma 10.13 *Euler's theorem* states that the following formula holds for the numbers of nodes n, of edges e, and of regions r for any connected planar graph or map G:

$$n - e + r = 2 \tag{10.98}$$

The proof of Lemma 10.13 may refer to Lipschutz and Lipson (1997).

The McCabe *cyclomatic complexity* [McCabe, 1976] of a software system can be determined by applying Euler's theorem onto the CFGs of software systems.

Definition 10.89 The *cyclomatic complexity* of a software system S, $C_r(S)$, is determined by the number of regions contained in a given CFG G_S, $r(G_S)$, provided that G is connected, i.e.:

$$\begin{aligned} C_r(S) &= r(G_S) \\ &= e - n + 2 \end{aligned} \tag{10.99}$$

where, e is the number of edges in G_S representing branches and cycles, n number of nodes in G_S where a block of sequential code may be reduced to a single node.

It can be observed that Eq. 10.99 is derived property of Euler's theorem as provided in Lemma 10.13, which shows that the physical meaning of the McCabe cyclomatic complexity is the number of regions in a CFG of a software system.

Example 10.9 The cyclomatic complexity of the program *MaxFinder* as given in Example 5.14 can be determined by using Eq. 10.99 on the CFG as shown in Fig. 5.19 as follows:

$$\begin{aligned} C_r(S) &= e - n + 2 \\ &= 7 - 6 + 2 \\ &= 3 \end{aligned}$$

Observing Fig. 5.19, it may be found that the result $C_r(S) = r(G_S) = 3$ is the number of regions in the CFG, providing there is always a region by linking the first node to the last node in the CFG. This finding indicates that the calculation as required in Eq. 10.99 can be omitted; instead, a simple count of the number of regions in G_S is enough.

Further, it may also be seen in Fig. 5.19 that the result $C_r(S) = r(G_S) = 3$ is the number of BCS's in the program or its formal specification, providing a single sequential BCS is always taken into account by one. This finding reveals that the drawing of a CFG for a given program or component is not necessary in the cyclomatic analysis. In other words, *r(CFG)* equals to the numbers of BCS's in a program [Wang, 2003f].

Incorporating the above findings with Definition 10.89, the following corollary is obtained.

Corollary 10.15 The *cyclomatic complexity* of a connected software system $C_r(S)$ can be determined by any of the following three methods:

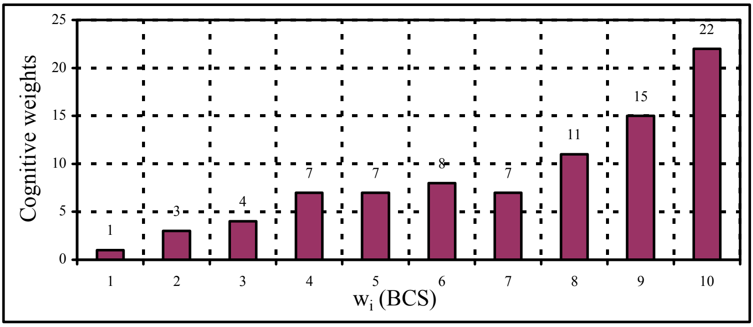
$$\begin{aligned} C_r(S) &= e - n + 2 && \text{// Method 1} \\ &= r(CFG) && \text{// Method 2} \\ &= \#(BCS) && \text{// Method 3} \end{aligned} \qquad (10.100)$$

10.7.3 THE COGNITIVE COMPLEXITIES OF SOFTWARE SYSTEMS

According to Theorem 10.13 on the orientations of complexity theories of computation and software engineering, the complexities of an abstract system can be classified as the *symbolic, relational, architectural, operational,* and *functional* complexities. This subsection describes the cognitive complexity of software systems that is a special type of software functional complexity defined as a product of the architectural and operational complexities of software systems.

10.7.3.1 The Operational Complexity of Software Systems

In Section 9.6.3 a set of calibrated cognitive weights of BCS's has been derived based on a series of psychological and cognitive experiments [Wang, 2005j]. The calibrated cognitive weights for the ten fundamental BCS's are illustrated in Fig. 10.25, where the relative cognitive weight of the sequential structure is assumed one, i.e., $w_1 = 1$.



Note: 1 – sequence, 2 – branch, 3 – switch, 4 – for-loop, 5 – repeat-loop, 6 – while-loop, 7 – functional call, 8 – recursion, 9 – parallel, 10 - interrupt

Figure 10.25 The relative cognitive weights of BCS's of software systems

According to the generic mathematical model of programs as modeled in Theorem 5.7, a software system can be rigorously described by a complex *process P* with a composed operation of *n* meta statements p_i and p_j, $1 \leq i < n$, $j = i+1$, based on certain composing relations r_{ij} known as the ten BCS's as shown in Fig. 10.25, i.e.:

$$\begin{aligned} P &= \mathop{R}_{i=1}^{n-1}(p_i \; r_{ij} \; p_j), j = i+1 \\ &= (...(((p_1) \; r_{12} \; p_2) \; r_{23} \; p_3) \; ... \; r_{n-1,n} \; p_n) \end{aligned} \qquad (10.101)$$

Therefore, the sum of the cognitive weights of all r_{ij} represents the operational complexity of a software system.

There are two structural patterns of BCS's in a software system: the *sequentially* and the *embedded* related BCS's. In the former, all the BCS's are in a linear layout in *S*, therefore the operational complexity of *S* is a sum of the cognitive weights of all linear BCS's. In the latter, some BCS's are embedded in others in *S*, hence the operational complexity of *S* is a product of the cognitive weights of inner BCS's and the weights of outer layer BCS's. In general, the two types of BCS architectural relations in *S* may be combined in various ways. Therefore, a general method for calculating the operational complexity of software can be derived as follows.

Definition 10.90 The *operational complexity* of a software system *S*, $C_{op}(S)$, is determined by the sum of the cognitive weights of its *n* linear blocks composed by individual BCS's, where each block may consist of *q* layers of embedded BCS's, and within each of the layer there are *m* linear BCS's, i.e.:

$$\begin{aligned} C_{op}(S) &= \sum_{k=1}^{n_c} C_{op}(C_k) \\ &= \sum_{k=1}^{n_c} (\prod_{j=1}^{q_k} \sum_{i=1}^{m_{k,j}} w(k,j,i)) \quad [\text{F}] \end{aligned} \tag{10.102}$$

If there is no embedded BCS in any of the n_c components in Eq. 10.102, i.e., $q = 1$, then Eq. 10.102 can be simplified as follows:

$$\begin{aligned} C_{op}(S) &= \sum_{k=1}^{n_c} C_{op}(C_k) \\ &= \sum_{k=1}^{n_c} \sum_{i=1}^{m_k} w(k,i) \quad [\text{F}] \end{aligned} \tag{10.103}$$

where *w*(*k, BCS*) is given in Fig. 10.25.

Definition 10.91 The *unit of operational complexity* of software systems is a single sequential operation called a function *F*, i.e.:

$$C_{op}(S) = 1 \; [\text{F}] \Leftrightarrow \#(\text{SeqOP}(S)) = 1 \tag{10.104}$$

With the cognitive weight of sequential process relation defined as one unit of operational function of software systems, complex process relations can be analyzed.

Example 10.10 The operational complexity of the algorithm of In-Between Sum, IBS_Algorithm**ST**, as given in Fig. 10.26, can be analyzed as follows:

$$\begin{aligned} C_{op}(S) &= \sum_{k=1}^{n_c} (\prod_{j=1}^{q} \sum_{i=1}^{m} w(k,i,j)) \\ &= \sum_{k=1}^{n_c} \sum_{i=1}^{m} w(k,i) \\ &= w_{BCS}(\text{SEQ}) + \{w(\text{ITE}) \bullet 2w(\text{SEQ}) + w(\text{ITE}) \bullet 2w(\text{SEQ})\} \\ &= 1 + (3 \bullet 2 + 3 \bullet 2) \\ &= 13 \quad [\text{F}] \end{aligned}$$

```
IBS_AlgorithmST ({I:: AN, BN}; {O:: ⓈIBSResultBL, IBSumN}) ≜
{
     MaxN := 65535
     → (  ? (0 < AN < maxN) ∧ (0 < BN < maxN) ∧ (AN < BN)
                → IBSumN := ((BN - 1) * BN) / 2) - (AN * (AN + 1) / 2)
                → ⓈIBSResultBL := T
          | ? ~
                → ⓈIBSResultBL := F
                → !(@'AN and/or BN out of range, or AN ≥ BN')
        )
}
```

Figure 10.26 The IBS algorithm (a) specified in RTPA

It is noteworthy that for a fully sequential software system where only *w*(sequence) = 1 [F] is involved, its operational complexity is reduced to the symbolic complexity.

Corollary 10.16 The symbolic complexity $C_s(S)$ is *a special case of the operational complexity* $C_{op}(S)$, where the cognitive weights of all kinds of BCS's, w_i(BCS), are simplified as always one, i.e.:

$$\begin{aligned} S_{op}(S) &= \sum_{k=1}^{n_C} \sum_{i=1}^{m_k} w(k,i) \\ &= C_s(S), w(k,i) \equiv 1 \\ &= C_s(S) \quad [\text{LOC}] \end{aligned} \tag{10.105}$$

Corollary 10.16 presents an important finding on the relationship between conventional symbolic complexity and the operational complexity of software. It indicates that the symbolic measure is oversimplified so that it cannot represent the real functional complexities and sizes of software systems. Case studies summarized in Table 10.4 show that algorithms or programs with similar symbolic complexities may possess widely different functional complexities in terms of the operational and cognitive complexities.

10.7.3.2 The Architectural Complexity of Software Systems

The architectural complexity of a software system is proportional to its number of global and local data objects such as inputs, outputs, data structures, and internal variables.

Definition 10.92 The *architectural complexity* of a software system *S, C_a(S),* is determined by the number of data objects at the system and component levels, i.e.:

$$\begin{aligned} C_a(S) &= \text{OBJ}(S)) \\ &= \sum_{k=1}^{n_{CLM}} \text{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \text{OBJ}(C_k) \ \ [\text{O}] \end{aligned} \tag{10.106}$$

where *OBJ* represents a function that counts the number of data objects in a given CLM (number of global variables) or components (number of local variables).

Definition 10.93 The *unit of architectural complexity* of software systems is a single data object, modeled either globally or locally, called an object *O*, i.e.:

$$C_a(S) = 1 \ [O] \Leftrightarrow \#(\text{OBJ}(S)) = 1 \tag{10.107}$$

There are special system input architectures known as the high throughput or pipeline system, in which a large even infinite number of similar inputs and/or outputs are operated. In this case, the architectural complexity of such pipeline systems will be defined as a relative equivalent constant rather than an absolute infinite as follows.

Definition 10.94 The *equivalent architectural complexity* of a high throughput or infinite pipeline system *S* with repetitive data objects, *C'_a(S),* is treated as a constant of three based on cognitive theory of inductive inferences.

In the above definition, *C'_a(S)* is determined on the basis of cognitive informatics where the inductive inference effort of a large or infinite series of similar patterns is equivalent to three, typically the first and the last items plus an arbitrary one in the middle. For instance, the equivalent number of the data object in the set {X[1]**N**, X[2]**N**, …, X[*n*]**N**} is counted as three rather than *n*.

Example 10.11 The architectural complexity of the *MaxFinder* component as given in Example 5.14 can be determined as follows:

$$\begin{aligned} C_a(MaxFider) &= \text{OBJ}(MaxFider) \\ &= \#(\text{inputs}) + \#(\text{outputs}) + \#(\text{local variables}) \\ &= 3{+}1{+}1 \\ &= 5 \quad [\text{O}] \end{aligned}$$

Example 10.12 The CLM SysClock**ST** given below encompasses 7 objects, therefore its architectural complexity is C_a(SysClock**ST**) = 7 [O].

```
SysClockST ≙  SysClockS ::
              ( <§t : N |  0 ≤ §tN  ≤ 1M>,
                <CurrentTime : hh:mm:ss:ms  | 00:00:00:000  ≤
                                  CurrentTime hh:mm:ss:ms ≤ 23:59:59:999>,
                <Timer : SS  | 0 ≤ TimerSS ≤ 3600>,
                <MainClockPort : B | MainClockPortB = FFF0H >,
                <ClockInterval : N | TimeIntervalN = 1ms>,
                <InterruptCounter : N |  0 ≤ InterruptCounterN  ≤ 999>
              )
```

10.7.3.3 The Cognitive Complexity of Software Systems

How the functional sizes of software systems may be modeled and measured is an age-old problem in software engineering. The concepts of *function point* [Albrecht and Gaffney, 1983] and MaCabe's cyclomatic complexity [McCabe, 1976] are proposed for measuring the functional complexity of software. However, in the former, it is not well defined what the physical meaning of a unit function point is. In the latter, only the internal loop architectures of a system are considered; the throughput of the system in terms of data objects and other internal architectures such as sequences, branches, and embedded constructs are excluded.

This subsection introduces the cognitive complexity of software systems as a fundamental measure of the functional sizes of software. It is empirically observed that the functional size of a software system is not only determined by its operational complexity, but also determined by its architectural complexity. That is, software functional size is proportional to its cognitive complexity, which is a product of its operational and architectural complexities.

According to Definition 6.58, the *semantic function* of a program $\wp$, $f_\theta(\wp)$, is a finite set of values V determined by a Cartesian product on a finite set of variables S and a finite set of executing steps T, i.e.:

$$
\begin{aligned}
f_\theta(\wp) &= f\colon T \times S \rightarrow V \\
&= \left(\begin{array}{lcccc}
 & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\
\mathbf{t_0} & \perp & \perp & \cdots & \perp \\
\mathbf{t_1} & v_{11} & v_{12} & & v_{1m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\mathbf{t_n} & v_{n1} & v_{n1} & \cdots & v_{nm}
\end{array}\right)
\end{aligned}
\qquad (10.108)
$$

where $T = \{t_0, t_1, ..., t_n\}$, $S = \{s_1, s_2, ..., s_m\}$, and V is a set of values $v(t_i, s_j)$, $0 \le i \le n$, and $1 \le j \le m$.

Therefore, the semantic space of a program can be illustrated by a two dimensional plane as shown in Fig. 10.26.

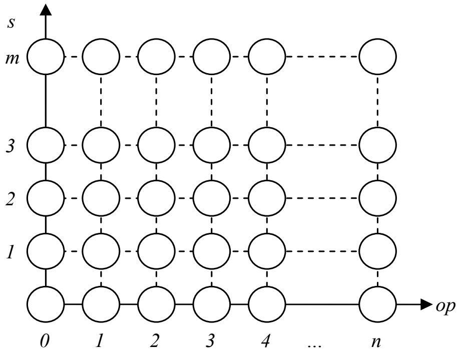


Figure 10.26 The semantic space of software systems

Observing Fig. 10.26 and Eq. 10.108, it can be seen that the complexity of a software system, or its semantic space, is determined not only by the number of operations, but also by the number of data objects. This leads to the formal description of the cognitive complexity of software systems.

The 41st Law of Software Engineering

Theorem 10.14 The *cognitive complexity of software* states that the *cognitive complexity* of a software system *S*, $C_c(S)$, is a product of its operational complexity $C_{op}(S)$ and its architectural complexity $C_a(S)$, i.e.:

$$\begin{aligned} S_f(S) &= C_{op}(S) \bullet C_a(S) \\ &= \{\sum_{k=1}^{n_C}\sum_{i=1}^{m_k} w(k,i)\} \bullet \qquad (10.109) \\ &\quad \{\sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k)\} \quad [\mathrm{FO}] \end{aligned}$$

Theorem 10.14, the 41st Law of software engineering, indicates that the more the architectural data objects and the higher the operational complicity onto these objects, the higher the cognitive complexity and the larger the functional size of the system.

Definition 10.95 The *unit of cognitive complexity* of software systems is a single sequential operation onto a single data object called a *function-object FO*, i.e.:

$$\begin{aligned} C_f &= C_{op} \bullet C_a \\ &= 1[\mathrm{F}] \bullet 1[\mathrm{O}] \\ &= 1 \quad [\mathrm{FO}] \end{aligned} \tag{10.110}$$

According to Theorem 10.14, the physical meaning of software cognitive complexity is how many function-object [FO] are equivalent for a given software system.

10.7.4 SOFTWARE SYSTEM COMPLEXITY ANALYSIS

In this subsection, the cognitive complexity of software systems with its operational and architectural complexities are compared with conventional system complexity measures such as time, cyclomatic, and symbolic (LOC) complexities. A set of case studies is carried out in order to examine the measurability and accuracy of various complexity and size measures for software systems. This subsection demonstrates that the cognitive complexity is the most sensitive measure for denoting the real complexities and functional sizes of software systems.

10.7.4.1 Comparative Case Studies on the Complexity Models of Software Systems

For self containment, all the RTPA specifications of the four cases are presented in Examples 10.13 through 10.15, except that the IBS_Algorithm (a) has been given in Fig. 10.26.

Example 10.13 The formal specification of the In-Between Sum (IBS) algorithm in RTPA, IBS_Algorithm**ST**, can be given in two approaches as specified in Figs. 10.26 (Algorithm (a)) and 10.28 (Algorithm (b)), respectively. Obviously, Algorithm (a) is more efficiently designed.

```
IBS_AlgorithmST ({I:: AN, BN}; {O:: ⓈIBSResultBL, IBSumN}) ≙
{ // Specification (b)
    MaxN := 65535
     → ( ? (0 < AN < maxN) ∧ (0 < BN < maxN) ∧ (AN < BN)
→ IBSumN := 0
                → IBSumN := $\mathop{R}\limits_{i\mathbf{N}=A\mathbf{N}+1}^{B\mathbf{N}-1} (IBSum\mathbf{N} + i\mathbf{N})$
                → ⓈIBSResultBL := T
          | ? ~
                → ⓈIBSResultBL := F
→ !(@'AN and/or BN out of range, or AN ≥ BN')
          )
}
```

Figure 10.28 The IBS algorithm (b) specified in RTPA

Example 10.14 The algorithm, *MaxFinder***ST**, is formally described in RTPA as shown in Fig. 10.29. Its function is to find the maximum number *max***N** from a set of *n* inputted integers {X[1]**N**, X[2]**N**, …, X[n]**N**}.

```
MaxFinderST ({I:: X[0]N, X[1]N, …, X[n-1]N }; {O:: maxN }) ≙
{
   XmaxN := 0
   → $\overset{nN-1}{\underset{iN=0}{R}}$ (
           ? X[i N]N > XmaxN
              → XmaxN := X[i N]N
         )
   → maxN := XmaxN
}
```

Figure 10.29 The *MaxFinder* algorithm specified in RTPA

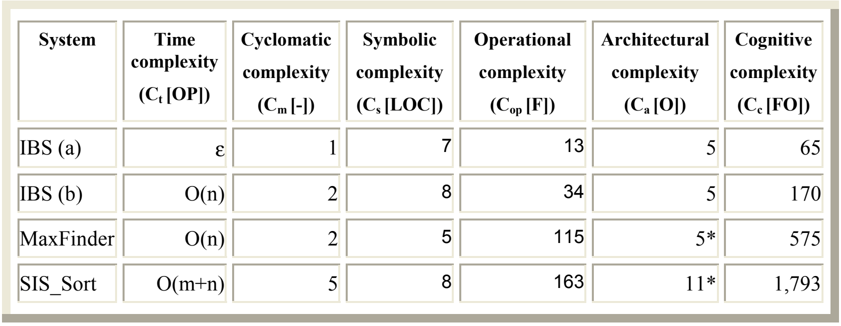
Example 10.15 The *Self-Index Sort algorithm* [Wang, 1996], SIS_Sort**ST**, can be formally described in RTPA as shown in Fig. 10.30.

```
SIS_SortST({I:: X[iN] Array }; {O:: X[$s_i$N] Array, ⓢSISResultBL) ≙
{ // <Input:: X[iN] : Array | 0 ≤ iN ≤ nN -1, 0 ≤ X[iN] N ≤ mN-1, mN > nN>
  // <Output:: X[$s_i$N] : Array | 0 ≤ $s_i$N ≤ nN -1,  $x_{s0}$≤ $x_{s1}$ ≤ , ..., ≤ $x_{si}$ ≤ ,..., ≤ $x_{sn-1}$>
  // <CLM:: SS[jN] : Array | 0 ≤ jN ≤ mN –1, 0 ≤ mN ≤ maxN>

  // Initialization
    $\overset{m-1}{\underset{j=0}{R}}$ SS[jN]N := 0

  // Self-index sorting
  → $\overset{n-1}{\underset{i=0}{R}}$ (↑ (SS[X[iN]N]N)

  // Compression
   → iN := 0

  → $\overset{m-1}{\underset{j=0}{R}}$ ( $\overset{ss[j]\leq 0}{\underset{ss[j]>0}{R}}$ ( X[iN] N := jN
                          → ↓(SS[jN]N)
                          → ↑(iN)
                          )
          )
 → ⓢSISResultBL := T
}
```

Figure 10.30 The SIS_Sort algorithm specified in RTPA

According to the definitions given in Sections 10.7.1 through 10.7.3, the typical complexities of software systems, known as the time complexity, cyclomatic complexity, symbolic complexity, and the cognitive complexity with the operational complexity and the architectural complexity, can be systematically analyzed as summarized in Table 10.4.

Table 10.4
Comparative Measurement of Software System Complexities



System	Time complexity (C_t [OP])	Cyclomatic complexity (C_m [-])	Symbolic complexity (C_s [LOC])	Operational complexity (C_{op} [F])	Architectural complexity (C_a [O])	Cognitive complexity (C_c [FO])
IBS (a)	ε	1	7	13	5	65
IBS (b)	O(n)	2	8	34	5	170
MaxFinder	O(n)	2	5	115	5*	575
SIS_Sort	O(m+n)	5	8	163	11*	1,793

* The equivalent objects as defined in Definition 10.94

Observing Table 10.4 it is noteworthy that the first three measurements, namely the time, cyclomatic, and symbolic complexities, cannot actually reflect the real complexity of software systems in design, representation, cognition, and/or comprehension in software engineering.

- Although the four example systems are with similar symbolic complexities, their operational and cognitive complexities are greatly different. This indicates that the symbolic complexity cannot be used to represent the operational or functional complexity of software systems.

- Symbolic complexity does not represent the throughput or the input size of problems.

- Time complexity does not work well for a system where is no loop and dominant operations, because theoretically in this case all statements in linear structures are treated as zero no matter how long they are. In addition, time complexity cannot distinguish the real complexities of systems with the same asymptotic function, such as in Case 2 (IBS (b)) and Case 3 (Maxfinder).

- The cognitive complexity is a more objective measure of software system complexities and sizes, because it represents the real semantic complexity by integrating both the operational and architectural complexities in a coherent measure. For example, the difference between IBS(a) and IBS(b) can be successfully captured by cognitive complexity. However, symbolic and cyclomatic complexities cannot identify the functional differences very well.

10.7.4.2 The Symbolic vs. Cognitive Sizes of Software Systems

On the basis of the complexity models developed so far, the sizes of software systems can be quantitatively analyzed by measures of the *symbolic, cognitive functional,* and *relational sizes* modeled by the corresponding complexity measures, respectively.

According to the generic system complexity theory discussed in Section 10.3.3, when a system is treated as a black box, the relational complexity of the system can be estimated by the maximum possible pairwise relations between all components in the system.

Definition 10.96 The *relational complexity* of software system *S, $C_r(S)$,* is the maximum number of relations n_r among components, i.e.:

$$\begin{aligned} C_r(S) &= n_r \\ &= n_c(n_c - 1) \quad [\text{R}] \end{aligned} \tag{10.111}$$

where the unit of the relational complexity is the number of relations *R*.

It is noteworthy that $C_r(S)$ provides the maximum potential or the upper limit of internal relational complexity of a given software system.

The relationship among the symbolic, relational, and operational complexities of software systems is plotted in Fig, 10.31 in the logarithmic scale. As shown in Fig. 10.31, the symbolic complexity of software $C_s(S)$ is the lower bound of the functional complexity of software and it is linearly proportional to the number of statements *n,* i.e., $O(n)$. The relational complexity $C_r(S)$ is the upper bound of functional complexity of software in the order of $O(n^2)$. Therefore, the real cognitive functional complexity represented by the operational complexity $C_{op}(S)$ is bounded between the curves of the symbolic and relational complexities.

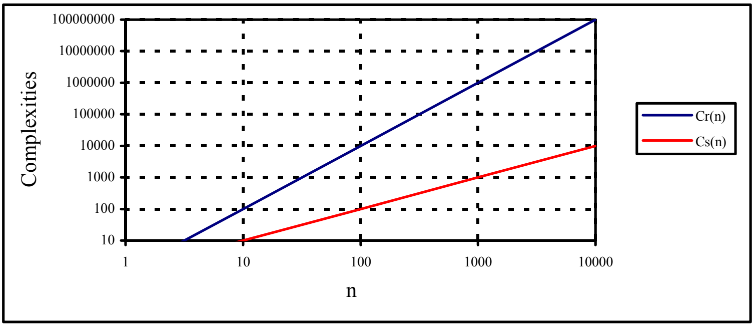


Figure 10.31 The relational and symbolic complexities as the upper/lower bounds of functional complexity of software systems

Fig. 10.31 indicates that the floor of the operational complexity $C_{op}(S)$ of software systems is determined by the symbolic complexity $C_s(S)$ when only sequential relation is considered between all adjacent statements in a given program, and all weights of the sequential relational operations in computing are simplified to one. The ceiling of the operational complexity $C_{op}(S)$ is determined by the relational complexity $C_r(S)$, when all potential relations among the components (statements) in computing are considered.

Corollary 10.17 The *operational complexity* $C_{op}(S)$ of a software system S is constrained by the lower bound of the symbolic complexity $C_s(S)$ and the upper bound of the relational complexity $C_r(S)$, i.e.:

$$O(n) \leq C_{op}(S) \leq O(n^2) \qquad (10.112)$$

where n is the number of statements in S.

According to Corollary 10.16 and Fig. 10.29 it can be seen that the real complexity and size of software systems are greatly underestimated when the conventional symbolic size measurement (LOC) is adopted, because it represents the minimum functional complexity of software. Therefore, the functional size of software systems measured by the cognitive complexity as described in Theorem 10.14, the 41st Law of software engineering, should be adopted to measure the actual sizes and complexities of software systems.

10.7.5 COHESION AND COUPLING COMPLEXITIES OF SOFTWARE SYSTEMS

The preceding subsections in Section 10.7 have focused on the measurements of software complexities and sizes within a software component. This subsection examining a larger scope where the focus is put on the relational complexity among components in software systems. Software system cohesion and coupling are introduced as a pair of higher-level relational complexities of software systems based on the system theory developed in Section 10.3.6. Then, properties and generic rules of software system cohesion and coupling are analyzed.

10.7.5.1 Cohesion of Software Systems

The relations between a given software system S and other systems can be categorized into internal relations ($R^c(S)$) and external relations ($R^i(S)$, $R^o(S)$). The former are relations between components belonging to S; the latter are those between components within and outside S.

Definition 10.97 The *cohesion of a software system S*, *CH(S),* is a ratio of the system's number of internal relations $\#R^c$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$, i.e.:

$$CH(S) = \frac{\#R^c}{\#R^c + \#R^i + \#R^o} \bullet 100\% \tag{10.113}$$

where $0\% \leq CH(S) \leq 100\%$.

It is expected that the higher the system cohesion, the better the architectural design. However, $CH(S) = 100\%$ is not a practical system because it indicates that S is a closed system.

If a system may be decomposed into multiple subsystems, each subsystem may be analyzed in the same way as defined in Eq. 10.113.

Corollary 10.18 Properties of software system cohesion are as follows:

- Nonnegative: $\forall S, CH(S) \geq 0$ (10.114a)
- Normalized domain: $\forall S, 0\% \leq CH(S) \leq 100\%$ (10.114b)
- Null if R^c is empty: $\exists S, R^c = \varnothing \Rightarrow CH(S) = 0\%$ (10.114c)
- Full if $R^i \cup R^o$ is empty: $\exists S, R^i \cup R^o = \varnothing \Rightarrow CH(S) = 100\%$ (10.114d)

10.7.5.2 Coupling of Software Systems

Definition 10.98 The *coupling of a software system S*, *CP(S),* is a ratio of the system's number of external relations $\#R^i + \#R^o$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$, i.e.:

$$CP(S) = \frac{\#R^i + \#R^o}{\#R^c + \#R^i + \#R^o} \bullet 100\% \qquad (10.115)$$

where $0\% \leq CP(S) \leq 100\%$ and a lower value of $CP(S)$ indicates a better architectural design.

Corollary 10.19 Properties of software system coupling are as follows:

- Nonnegative: $\forall S, CP(S) \geq 0$ (10.116a)
- Normalized domain: $\forall S, 0\% \leq CP(S) \leq 100\%$ (10.116b)
- Null if $R^i \cup R^o$ is empty: $\exists S, R^i \cup R^o = \varnothing \Rightarrow CP(S) = 0\%$ (10.116c)
- Full if R^c is empty: $\exists S, R^c = \varnothing \Rightarrow CP(S) = 100\%$ (10.116d)

The relationship between the cohesion and coupling of software systems is constrained by the same complement law as described in Section 10.3.6. Therefore, when either cohesion or coupling of a software system is known, the other can be determined.

10.7.5.3 Comparative Analysis of Software System Cohesions and Couplings

Reusing the examples and data shown in Table 10.4, the cohesions and couplings of the four given systems can be calculated as given in Table 10.5 using Eqs. 10.113 and 10.115.

Table 10.5
Measurement of Software System Cohesions and Couplings

System	Internal relations (#R^c)	External relations		Total relations (#R)	System cohesion (CH(S) [%])	System coupling (CP(S) [%])
		Input relations (#R^i)	Output relations (#R^o)			
IBS	7	2	2	11	63.6	36.4
ATM-PIN	23	1	3	27	85.2	14.8
MaxFinder	5	n	1	n+6	~0	~100
SIS_Sort	8	n	n+1	2n+9	~0	~100

In Table 10.5, there are a category of software systems that may be classified as a *pipeline system* such as a data processing system and a data sort system. The size of inputs and/or outputs *n* of such systems as those of the MaxFinder and SIS-Sort system is often flexible and indeterminable before run-time. In this case, the cohesion and coupling can be redefined as a limit below.

Definition 10.99 The *limits of cohesion and coupling* of software systems with indeterminable variable $\#R^i$ and $\#R^o$ are treated as functions $CH(S, n)$ and $CP(S, n)$ as n approaching ∞, i.e.:

$$\begin{aligned}
\lim_{n\to\infty} CH(S,n) &= \lim_{n\to\infty}\left(\frac{\#R^c}{\#R^c + \#R^i + \#R^o} \bullet 100\%\right) \\
&= \lim_{n\to\infty}\left(\frac{c}{c+cn}\right) \bullet 100\% \\
&= 0\%
\end{aligned} \tag{10.117}$$

$$\begin{aligned}
\lim_{n\to\infty} CP(S,n) &= \lim_{n\to\infty}\left(\frac{\#R^i + \#R^o}{\#R^c + \#R^i + \#R^o} \bullet 100\%\right) \\
&= \lim_{n\to\infty}\left(\frac{cn}{c+cn}\right) \bullet 100\% \\
&= 100\%
\end{aligned} \tag{10.118}$$

where *c* and *k* are a positive constant.

Eqs. 10.117 and 10.118 explain the nature of pipeline systems, where the cohesions of such systems are approaching 0, while their couplings are equivalent to 100%.

Software system coupling is usually too high to be efficiently handled in *ad hoc* system designs. Therefore, the normalized system decomposition rules and the system organization tree structures developed in Sections 10.3.4 and 10.3.5 should be adopted as guidelines in the architectural designs of software systems.

The 37th Principle of Software Engineering

Theorem 10.15 The *normalized software system architectures* states that components of different subsystems should not be coupled directly, rather than be invoked through their top layer components shared in the same subsystem.

Theorem 10.15 can be illustrated in Fig. 10.32 that shows valid and invalid couplings in structured programming. According to Theorem 10.15, if there is a need to couple two components belonging to different subsystems, the coupling of them should be done through their common parent nodes in the system hierarchical architecture. This forms the basic principle of software system modularization.

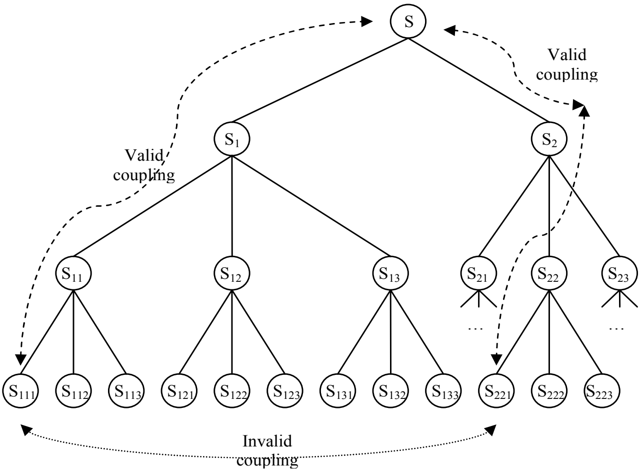


Figure 10.32 The normalized system architecture for component couplings

Corollary 10.20 In order to reduce system complexity and maintain a manageable cognitive handling ability, the *coupling among components* of a software system should be implemented through their common parent node (the super system) rather than by direct links between them.

10.8 Summary

In this chapter, an abstract system has been modeled as a collection of coherent and interactive entities that possesses stable functions and a clear boundary with external environment. The generic rules and theories of abstract systems and their applications in concrete systems have been developed. It has been identified that, to some extent, management science, economics, and sociology may be perceived as a special branch of system science that study objects and phenomena at different levels of human work and social organizations.

Because software engineering is one of the most complicated system engineering areas, it has naturally been identified as the ideal testbed for evaluating existing system theories and their enhancements. Treating software engineering and large-scale software project via system engineering has formed a promising trend in dealing with the problems, complexities, and human factors in software engineering.

This chapter has presented a systematic view towards software engineering. The theories of systems science, as well as underlying principles and modeling techniques of systems engineering, have been explored. A new mathematical structure, process algebra, has been developed to model and manipulate abstract and concrete systems, particularly software systems. Applications of system theories and system engineering methodologies in software engineering have been discussed. As a result, the **system science foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *System Science Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 10. Systems Science Foundations of SE

■ System philosophies

- The system metaphor for modeling complex entities
- Holism
- Systematic thinking

■ Abstract systems and system topology

- Mathematical models of abstract systems
  - The mathematical model of closed systems
  - The mathematical model of open systems
- Taxonomy of systems
  - Concrete and abstract systems
  - Physical and social systems
  - Finite and infinite systems
  - Closed and open systems

  - Static and dynamic systems
  - Linear and nonlinear systems
  - Continuous and discrete systems
  - Precise and fuzzy systems
  - Determinate and indeterminate systems
  - White-box and black-box systems
  - Intelligent and nonintelligent systems
  - Maintainable and nonmaintainable systems
- Magnitudes of systems
  - System sizes, magnitudes, and complexities
  - Taxonomy of system magnitudes
- Hierarchical architecture of systems
- The system organization tree
- Systems cohesion and coupling
  - The border of systems
  - System cohesions and coupling

■ System algebra

- Relational operations of systems
  - Algebraic relations of closed systems
  - Algebraic relations of open systems
  - Relationships between closed and open systems
- Algebraic operations of systems
  - System conjunction
  - System difference
  - System composition
  - System decomposition

■ Principles of system science

- System fusion and mutation
- System functions and behaviors
- Work done by systems
- The maximum output of systems
- System equilibrium and organization
  - The generic IPO model of systems
  - Laws of system equilibrium and organization
- System synchronization and coordination
- System dissimilation
  - Dissimilation of nonmaintainable systems
  - Dissimilation of maintainable systems

■ Software system engineering

- The abstract model of computing systems

- The hierarchical model of software systems
  - The hierarchical structure of software systems
  - The hierarchical structure of software engineering processes and work products
- The ISO/IEC 15288 system engineering model for SE
- SE phenomena as system engineering problems

■ The complexity theory of software systems

- Computational complexity
  - Taxonomy of computational problems
  - Time complexity of algorithms
  - Space complexity of algorithms
- Symbolic and Control flow complexities
  - Symbolic complexity of software systems
  - Control flow complexity of software systems
- Cognitive complexity of software systems
  - The operational complexity of software systems
  - The architectural complexity of software systems
  - The cognitive complexity of software systems
- Software cognitive complexity analysis
  - Comparative case studies on the complexity models of software systems
  - The symbolic vs. functional sizes of software systems
- Cohesion and coupling complexity of software systems
  - Cohesion of software systems
  - Coupling of software systems
  - Analysis of software system cohesion and coupling

SIGNIFICANT FINDINGS OF THIS CHAPTER

- An **abstract system** is an algebraic model of generic systems that encompasses a collection of coherent and interactive entities and possesses stable functions and a clear boundary with external environment.

- The principle of **generic constraints** states that any system is constrained by a set of common conditions, properties, and rules, which are obeyed by components inside the system, but not by those outside the system.

- **Continuous and discrete systems** are equivalent because any continuous system can be simulated by a discrete system on the basis of behavioral equivalence.

- The **holism complexity of systems** states that within the 7-level magnitudes of systems, known as the *empty, small, medium, large, giant, immense,* and *infinite* systems, almost all systems are too complicated to be cognitively understood or mentally handled as a whole, except small systems or those can be decomposed into small systems.

- The **generic topology of systems** tends to normalized into a hierarchical structure in the form of a **complete *n*-nary tree**.

- **Advantages of the normalized tree** architecture of systems are as follows:

    a) **Equilibrium**: Looking down from any node at a level of the system tree, except at the leave level, the structural property of fan-out or the number of coordinated components are the same and evenly distributed.

    b) **Evolvablility**: A normalized system does not change the existing structure for future growth needs.

    c) **Optimal predictability**: There is an optimal approach to create a unique system structure $T_c(n, N)$ determined by the attributes of the unified fan-out n and the number of leave nodes N at the bottom level.

- The **cohesion and coupling** of any open system are complementary, i.e., $CH(S) + CP(S) = 100\%$.

- The **equivalence between open and closed systems** states that an open system S is equivalent to a closed system $\widehat{S}$, and vice versa, when it is conjoined with its environment Θ_S or $\Theta_{\widehat{S}}$, respectively.

- The following **architectural rules of a normalized system** should be maintained in system compositions: a) Direct relations between subsystems at different levels of the hierarchy are not allowed; b) Direct relations between components of different subsystems are not allowed; and c) Communications across systems and subsystems should be through commonly inherited higher-level system(s) in the system hierarchy.

- The **system fusion principle** states that the *fusion effect* of systems is generated by either increments of quantity in the set of components C or increments of diversity in the set of relations R.

- There exists a threshold that triggers the fusion effect of systems known as the **critical mass** Q_{cm}, which is the minimum quantity for obtaining or implementing system fusion over the increment of quantity.

- The principle of **system gain of work** states that work done by a system is always greater than that of any of its components, but at most equals the sum of those of its components.

- The **equilibrium of a system** is a stable state where the effects of all components in term of their abstract work form a zero-sum.

- The **equilibrium system work** states that the sum of all types of work is always zero in an equilibrium system, i.e., $\sum_{i=1}^{n} W(C_i) = 0$, where $W(C_i)$ is the abstract work of a component C_i in the system.

- The necessary and sufficient **mathematical condition** of **self-organization** is the existence of at least **one minimum** on the state curve of a system $f(x)$.

- The **functional condition** of **self-organization** of a system is the existence of the **negative feedback mechanism** that is proportional to the incremental or aggressive effects of the system.

- A **dynamic system** tends to synchronize on a certain state where it is stable or dynamically equilibrium that satisfies one of the following conditions: a) The **rational** condition: The apparent best equilibrium condition; or b) The **arbitrary** condition: The first met or most conventional equilibrium condition.

- **System dissimilation** states that any system tends to undergo a continuous degradation that leads to the eventual loss of its designed utility and against its initial purposes to form the system.

- The **orientation of software engineering complexity theories** states that the orientations of complexity theories of computation and software engineering are different. That is, the former is focused on the problems of *high throughput complexity* that are computing **time efficiency centered**;

while the latter puts emphases on the problems of *functional complexity* that are human **cognition time and workload oriented**.

- In software engineering, a problem with very high **computational complexity** may be quite simple for human comprehension, and vice versa. According to **cognitive informatics**, human beings may comprehend a large cycle of iteration, which is the major issue of computational complexity, by looking at only the beginning and termination conditions, as well as one or a few arbitrary internal loops with inductive inferences. However, humans are not good at dealing with **functional complexities** such as huge numbers of interrelated operations and very abstract data objects. Therefore, the **system complexity of large-scale software** is the focus of software engineering.

- The **symbolic complexity** $C_s(S)$ is a special case of the **operational complexity** $C_{op}(S)$, where the cognitive weights of all kinds of BCS's, $w_i(BCS)$, are simply treated as one (Corollary 10.16).

  - This is an important finding on the relationship between conventional symbolic complexity and the operational complexity of software. It indicates that the symbolic measure is **oversimplified** so that it cannot represent the real functional complexities and sizes of software systems. In other words, algorithms or programs with similar symbolic complexities may possess widely different functional complexities in terms of the operational and cognitive complexities.

- The **cyclomatic complexity** of a connected software system $C_r(S)$ can be determined by any of the following three methods: a) $C_r(S) = e - n + 2$; b) $C_r(S) = r(CFG)$; or c) $C_r(S) = \#(BCS)$.

  - The **advantage of the third method** is that it does not require for transforming a given program into a CFG.

  - The **McCabe cyclomatic complexity** is a directly derived property of CFGs based **Euler's theorem** in graph theory, which is used for determining regions for a given connected planar graph with know topology.

- Comparative studies show that the **time, cyclomatic,** and **symbolic complexities** cannot actually reflect the real complexity of software systems in design, representation, cognition, and/or comprehension.

  - **Symbolic complexity** does not correlate to the operational or functional complexity of software systems. Symbolic complexity does not represent the throughput or the input size of problems.

- **Time complexity** does not work well for a system where there are no loops and dominant operations. Time complexity cannot distinguish the real complexities of systems with the same asymptotic function.

- The **cognitive complexity** is a more objective measure of software system complexities and sizes, because it represents the real semantic complexity by integrating both the operational and architectural complexities in a coherent measure.

• The **operational complexity** $C_{op}(S)$ of a software system S is constrained by the lower bound of the symbolic complexity $C_s(S)$ and the upper bound of the relational complexity $C_r(S)$, i.e., $O(n) \leq C_{op}(S) \leq O(n^2)$, where n is the number of statements in S.

- It indicates that the real complexities and sizes of software systems are used to be **greatly underestimated** when the conventional symbolic size measurement (LOC) is adopted, because it represents only the minimum functional complexity of software.

- The **cognitive complexity** results in more actual measurement of the **functional sizes** of software systems.

• The **normalized software system architectures** state that components of different subsystems should not be coupled directly, rather than be invoked through their top layer components in the same subsystem.

• In order to reduce system complexity and **maintain a manageable cognitive handling ability**, the coupling among components of a software system should be implemented through their common parent node (the super system) rather than by direct links between them.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

System philosophies

• The **system philosophy** is based on the observation that the nature is built by a small number of basic components and particles, and governed by a limited set of basic laws. Even all living things is configured by almost the same cells, chromosomes, and DNAs.

• The discipline of **system science** is an inquiry into the general principles and rules commonly shared by different kinds of systems. The system metaphor is one of the most widely used concepts and notions in almost all disciplines of science, engineering, and society.

System Abstraction

- The **generality principle** of system abstraction states that a system can be represented as a whole in a given level k of reasoning without knowing the details at levels below k.

- **Abstract systems** can be classified into two categories known as the closed and open systems. Most practical and useful systems in nature are open systems in which there are interactions between the system and its environment.

  - A **closed system** $\widehat{S}$ is a 4-tuple, i.e., $\widehat{S} = (C, R, B, \Omega)$, where C is a nonempty set of components of the system; R is a nonempty set of relations between pairs of the components in the system, $R \subseteq C \times C$; B is a set of behaviors (or functions); and Ω is a set of constraints on the memberships of components, the conditions of relations, and the scopes of behaviors.

  - An **open system** S is a 7-tuple, i.e., $S = (C, R, B, \Omega, \Theta) = (C, R^c, R^i, R^o, B, \Omega, \Theta)$, where Θ is the environment of S with a nonempty set of components C_Θ outside C; $R^c \subseteq C \times C$ is a set of internal relations; $R^i \subseteq C_\Theta \times C$ is a set of external input relations; and $R^o \subseteq C \times C_\Theta$ is a set of external output relations.

System topology

- Systems as complex entities may be classified into various categories according to key characteristics of their *components* (C), *relations* (R), *behaviors* (B), *constraints* (Ω), and/or *environments* (Θ).

  - A **concrete system** is a real and specific system with natural entities and certain functions.

  - An **abstract system** is a virtual or theoretical system that is modeled by mathematics or computing simulations.

- The **size of a system** S_s is the number of **components** encompassed in the system, i.e., $S_s = \#C = n_c$.

- The **magnitude of system** M_s is the number of asymmetric binary **relations** among the n_c components of the system including the reflexive relations, i.e., $M_s = \#R = n_c^2$.

- The taxonomy of the magnitudes of systems can be classified at seven levels, from bottom up, known as:
  - The **empty** system: ($S_s = 0$, $M_s = 0$)
  - **Small** system: ($1 \leq S_s \leq 10$, $1 \leq M_s \leq 100$)
  - **Medium** system: ($10 < S_s \leq 100$, $100 < M_s \leq 10^4$)
  - **Large** system: ($10^2 < S_s \leq 10^3$, $10^4 < M_s \leq 10^6$)
  - **Giant** system: ($10^3 < S_s \leq 10^4$, $10^6 < M_s \leq 10^8$)
  - **Immense** system: ($10^4 < S_s \leq 10^5$, $10^8 < M_s \leq 10^{10}$)
  - **The infinite** system: ($S_s = \infty$, $M_s = \infty$)

- A **System Organization Tree** (*SOT*) is an *n*-nary complete tree in which all leave nodes represent a *component* and the remainder, all nodes beyond the leaves, represent a *subsystem*.

- The **cohesion of a system *S*,** *CH*(*S*), is defined as a ratio between its number of internal relations $\#R^c(S)$ and the total relations of the system $\#R(S)$.

- The **coupling of a system *S*,** *CP*(*S*), is defined as a ratio between its number of external relations $\#R^i(S) + \#R^o(S)$ and the total relations of the system $\#R(S)$.

- The **relationship** between *cohesion* and *coupling* of any open system *S* are complementary, i.e., $CH(S) + CP(S) = 100\%$.

System Algebra

- **System algebra** is an abstract mathematical structure that provides an algebraic treatment of abstract systems as well as their relations and operation rules for forming complex systems.

- **System relationships** in system algebra can be *equivalent, independent, overlapped, related, being subsystem,* and *being super system.*
  - Two systems S_1 and S_2 are **equivalent,** denoted by =, if all sets of components, relations, behaviors, constraints, and environments are identical.
  - Two systems S_1 and S_2 are **independent,** denoted by $\not{R}$, if both their component sets and external relation sets are disjoint.

- Two systems S_1 and S_2 are **overlapped,** denoted by Π, if their component sets are overlapped.

- Two systems S_1 and S_2 are **related,** denoted by R, if either the sets of their input relations or output relations are overlapped.

- A **subsystem** S' is a system that is encompassed in another system S, denoted by $\sqsubseteq$.

- A ***super system*** S is a system that encompasses one or more subsystems S', denoted by $\sqsubseteq$.

- **System operations** in system algebra are *system conjunction, disjunction, difference, composition,* and *decomposition* as defined below.

- The **conjunction** of two open systems S_1 and S_2, denoted by $\sqcup$, results in a super system that is formed by **incremental conjunctions** of both sets of relations and behaviors, respectively, as well as simple conjunctions of sets of components, constraints, and environments.

- The **difference** between an open systems S and an open subsystem S_1, denoted by $\boxminus$, results in an open subsystem S_2 that is formed by the differences of sets of components, input relations, output relations, and constraints, difference of sets of internal relations minus both $R^c{}_1$ and $\Delta R^c{}_{12}$, as difference of sets of behaviors minus both B_1 and ΔB_{12}.

- The **composition** of two open systems S_1 and S_2, denoted by $\uplus$, is an integration of both systems into a super system S at a given level of the system hierarchy by one of the compositional relations $R_c = \{||, \rightarrow, \rightarrowtail\}$.

- The **decomposition** of an open systems S, denoted by $\pitchfork$, is to break up S into two subsystems at the same level of the system hierarchy by one of the compositional relations $R_c = \{||, \rightarrow, \rightarrowtail\}$.

System principles

- The **system gain of functionality** states that system conjunction or composition between two systems S_1 and S_2 creates **new relations** ΔR_{12} and/or **new behaviors** (functions) ΔB_{12} that are solely a property of the newly established super system S, which can be determined by the sizes of the two intersected component sets $\#C_1$ and $\#C_2$.

- The **fusion effect** of a system is a self-productive property of systems that only appears when the system functions collectively as a whole.

- **System mutation** states that the gradual increment of quantity of a system, i.e., ΔC or ΔR, in a system beyond the point of the critical mass Q_{cm} triggers the abrupt generation of functionality (quality) F_{cm} of the system.

- The **abstract work done by a system** S, $W(S)$, is its output of utility U in term of number of functions F implemented, i.e., $W(S) = U \ [\mathrm{F}]$.

    - The **bottleneck principle of systems** states that the output work of a *serial* system $W(S_s)$ is determined by the least powerful component of the system.

    - The **appreciation principle of systems** states that the output work of a *parallel* system $W(S_p)$ is a sum of the work done by all its components less the overhead of the system ϖ.

- **System organization** is a process to configure and manipulate the system into or approaching to a stable and ordered state with an internal equilibrium.

- **System synchronization** states that a system reaches its maximum utility $\vec{S}_{\max}$ when all components' efforts $\overrightarrow{S_1}$ and $\overrightarrow{S_2}$ are synchronized.

- **Dissimilation** is the tendency that a system undergoes in an apparent or hidden destructive change against its original purposes or designed functions. System dissimilation can be analyzed on the basis of how systems maintain their functional availability and against the loss of it, which can be the designed utility, function, efficiency, or reliability of the system.

    - The dissimilation property of **maintainable systems** can be derived based on those of nonmaintainable systems, if the maintenance effect is treated as a recovery of the original availability of the system.

Software System engineering

- The **Generic Computing System** (GCS) is an abstract logical model of the executing platform, which controls a set of processes and underlying resources, such as memory, ports, and the system clock. A process is dispatched and controlled by the system that is triggered by various external, timing, or interrupt events.

- The **hierarchical structure of software systems** shows that a software system can be decomposed from the top down at seven levels known as those of the *system, subsystem, component* (class, or pattern), *function* (method), *statement, data model* (structure), and *target code.* The layered decomposition of software systems can be perceived as a **stepwise refinement process** that transfers a system into target code.

- **Software engineering processes** and their **work products** can be described by the layered system model of software engineering processes and corresponding work products and documentation. The clarification of the work products and results of each layered process is helpful to establish job expectations, quality standards, and process transition criteria in software engineering organization.

- Many phenomena in software engineering practice can be identified as **typical system engineering problems**, such as: a) New is beautiful; b) Fundamental research left behind industrial practices; c) Overlooked coordinative work organization as the key software engineering technology; d) Product lifespan is too short; e) Local maximum is often adopted; f) Pentium inside? g) Views on software systems – pessimism vs. optimism; h) The software maintenance crisis; i) Synchronization by process-based software engineering; and j) Measuring the tendency of programmers.

The complexity theory of software systems

- Although **computational complexities**, particularly algorithm complexities, are one of the focuses in computer science, software engineering is particularly interested in the **functional complexity** of large scale and real-world systems.

- **Computational complexity theories** study the solvability in computing. The *solvable problems* are those that can be computed by *polynomial-time* consumption. The *nonsolvable problems* are those that cannot be solved in any practical sense by computers due to excessive time requirements.

  - The **time complexity** of an algorithm for a given problem is measured as an estimation of the number of dominant operations in the algorithm, where each of the dominant operations is assumed to take an identical unit of time in operation.

  - The **space complexity** of an algorithm for a given problem is the maximum required space for both working memory and target code memory. Because the target code memory is a constant and determinable, software space complexity is focused on the working memory complexity.

- The **symbolic complexity** of a software system *S*, $C_s(S)$, is the linear length of its static statements measured in the unit of lines of code (LOC).

- The **cyclomatic complexity** of a software system *S*, $C_r(S)$, is determined by the number of regions contained in the CFG, *r*(*CFG*), provided that *CFG* is connected according to Euler's theorem, i.e., $C_r(S) = r(CFG) = e - n + 2$, where, *e* is the number of edges in *CFG* representing branches and cycles, *n* number of nodes in *CFG* where each node is equivalent to a block of sequential code.

    - The cyclomatic complexity $C_r(S)$ can be determined by three **methods** such as: a) $C_r(S) = e - n + 2$; b) $C_r(S) = r(CFG)$; or c) $C_r(S) = \#(BCS)$. The advantage of the third method is that it does not require for transforming a given program into a CFG.

- The **cognitive complexity *of*** a software system *S*, $C_c(S)$, is a product of the **operational complexity** $C_{op}(S)$ and the **architectural complexity** $C_a(S)$. The **unit** of cognitive complexity is **function-objects** [FO].

    - The ***operational complexity*** of a software system *S*, $C_{op}(S)$, is determined by the sum of the cognitive weights of its *n* linear blocks composed by individual BCS's, where each block may consist of *q* layers of embedded BCS's, and within each of the layers there are *m* linear BCS's. The **unit** of operational complexity is **functions** [F].

    - The **architectural complexity** of a software system *S*, $C_a(S)$, is determined by the number of data objects at the system and component levels. The **unit** of architectural complexity is **objects** [O].

- The **cohesion of a software system** *S*, *CH(S)*, is a ratio of the system's number of internal relations $\#R^c$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$.

- The **coupling of a software system** *S*, *CP(S)*, is a ratio of the system's number of external relations $\#R^i + \#R^o$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$.

- In order to reduce system complexity and maintain a manageable cognitive handling ability, the **coupling among components** of a software system should be implemented through their common parent node (the super system) rather than by direct links between them.

Questions and Research Opportunities

10.1 The slogan of system science and philosophy is: “The whole is more than the sum of its parts.” Discuss on which condition(s) it is true and on which condition(s) it may be false.

10.2 Why do more and more researchers believe software engineering should adopt the system engineering approach? How may system science play an important role in software engineering?

10.3 According to system philosophy, explain why there is no number two in *sciences*, while there is no number one in *engineering*.

10.4 According to Definitions 10.5 and 10.3, discuss what the differences between an open system and a closed system are.

10.5 Why are almost all real-world systems open systems? Is the empty system an open or closed system? What is that of the universe system?

10.6 Thy to prove that the universal system $\mathfrak{U}(C_\mathrm{U}, R_{\mathrm{U},} B_\mathrm{U}, \Omega_\mathrm{U})$ is a closed system.

10.7 Try to prove that the empty system $\mathfrak{O}(C_\varnothing, R_{\varnothing,} B_\varnothing, \Omega_\varnothing)$ is a closed system.

10.8 What are the five basic characteristics or criteria based on that system taxonomy may be classified?

10.9 What are the differences between system sizes and system magnitudes?

10.10 What is the seven-level taxonomy of system scales based on system sizes and magnitudes? Why may systems easily grow very large and complicated?

10.11 Given a software system with 20 components, calculate the size, magnitude, and relative complexity of the system according to the *system magnitude model*. Then, determine the category of the system in the seven-level hierarchy of system magnitudes.

10.12 Redo Ex. 10.11 assuming a software system with 300 components and discuss what techniques may be adopted to deal with the complexity of such a system.

10.13 Why should the generic system topology be a hierarchical structure based on the complete *n*-nary tree? What are the advantages of the normalized system architecture?

10.14 An advanced property of the complete *n*-nary tree $T_c(n, N)$ is that it is uniquely determinable by only two attributes: the number of fan-out of the tree n and the total number of leaves N. Try to determine the architecture of two complete *n*-nary trees, $T_{c1}(n_1, N_1) = (3, 20)$ and $T_{c2}(n_2, N_2) = (4, 30)$, based on the given values.

10.15 According to Corollary 10.5, what are the underlying reasons that force systems to take hierarchical tree structures?

10.16 Apply the system organization theories to analyze the following issues:

a) Given a system S with 8 components (at the leave level), try to determine the structure of a complete ternary tree for system organization.

b) When 5 additional components are included into S, what changes need to be made in the ternary organization tree?

10.17 Draw a diagram to denote the growth of a system organization tree from $SOT(3, 5)$ to $SOT(3, 14)$.

10.18 Draw a diagram for the system organization tree $SOT(5, 30)$, and determine its structural attributes according to Corollary 10.6.

10.19 Given a normalized system organization tree $SOT(\overline{n}_{fo}, N) = SOT(4, 20)$, where N is the number of employees, and $\overline{n}_{fo}$ is the average fan-out of groups, how many managers are needed in this organization? What is the depth of the normalized organization tree?

10.20 According to Corollary 10.6, all properties of an n-nary *system organization tree* *SOT*($\overline{n}_{fo}$, *N*) are uniquely determined when the total number of leave nodes *N* and the average fan-out $\overline{n}_{fo}$ are given. For a system *SOT*($\overline{n}_{fo}$, *N*) = *SOT*(3, 18), try to determine the following properties:

a) The maximum number of fan-out of any node $\overline{n}_{fo}$.

b) The maximum number of nodes at a given level *k*, $n_{k.}$

c) The depth of the *SOT*, *d*.

d) The maximum number of nodes in the *SOT*, N_{SOT}.

e) The maximum number of *components* (on all leaves) in the *SOT*, N_e.

f) The maximum number of *subsystems* (nodes except all leaves) in the *SOT*, N_m.

10.21 What are the values of cohesion and coupling of a closed system?

10.22 What are the values of cohesion and coupling of an empty system?

10.23 What are the conditions that convert a closed system to an open system, and vice versa?

10.24 What is the mathematical model of *incremental union* $\Cup$, and how may it be used to explain the system gains during system conjunction and composition?

10.25 According to the law of system maximum gain, discuss why the conventional description of system gains, $W(S) \geq \sum_{i=1}^{n} W(C_i)$, is incorrect?

10.26 What would necessarily be sacrificed when systems gain new functionality from composition of multiple components?

10.27 What is the condition of system equilibrium? Try to provide an example for a particular equilibrium system.

10.28 What are the conditions of system self-organization? Try to provide an example for a particular self-organized system.

10.29 What is system dissimilation? What are the differences between dissimilations of maintainable and nonmaintainable systems?

10.30 What are your suggestions for implementing the system's maximum output in software engineering organization?

10.31 On the basis of system theories, try to identify a software engineering problem that is a system engineering issue rather than a technical issue.

10.32 Comparatively analyze the symbolic complexity and operational complexity of the formal model *Queue***ST** as given in Fig. 4.8.

10.33 According to the definition of software cognitive complexity, explain why the functional complexity of software is not a simple measure rather than a complex one as a product of software architectural and operational complexities.

10.34 Calculate the cognitive complexity of the formal model *Queue***ST** as given in Figs. 4.7 and 4.8.

10.35 Using the data presented in Table 10.4, explain why programs with widely different functional/cognitive complexities would not be distinguished by the measure of symbolic complexity.

10.36 Why is the symbolic complexity $C_s(S)$ treated as a special case of the operational complexity $\mathrm{C_{op}(S)}$? What is the general case of problems? What is the basic assumption that has been simplified in the symbolic complexity?

10.37 According to Corollary 10.17 and Fig. 10.31, explain why the real relational and functional complexities of software systems have been totally underestimated by the measure of the symbolic complexity.

10.38 Comparatively analyze the following methodologies for software complexity measurements, and discuss their advantages, disadvantages, and usages (application areas) in a table:

- Computational (time) complexity
- Symbolic complexity (in LOC)
- Cyclomatic complexity
- Cognitive complexity

10.39 Why may the cohesion and coupling complexities be treated as the system-level relational complexity measure for software systems, particularly component-based system?

10.40 According to Theorem 10.15, explain why direct coupling is prohibited in structured programming and normalized system architectures.

10.41 Read the following classic article in system science:

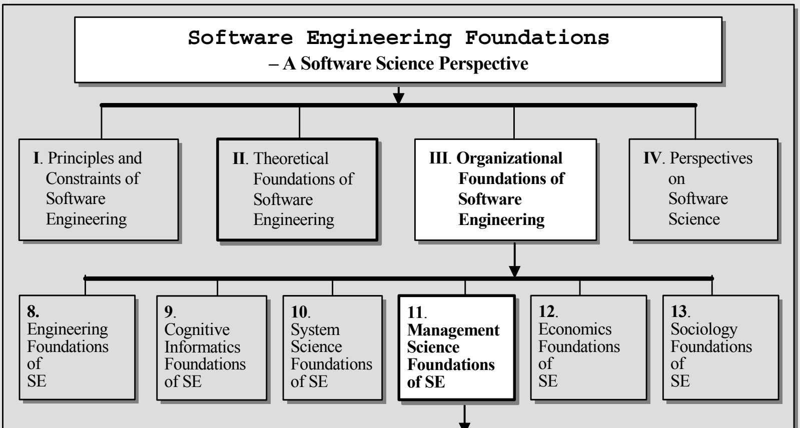
> George J. Klir (1988), System Profile: The emergence of System Science, System Research, 5(2), pp.145-156.

Discuss the following topics in a group:

- About the author.
- How did system science emerge?
- Why would system science and engineering be the next focus in software engineering?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 11

MANAGEMENT SCIENCE FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE

11. Management Science Foundations of SE

Knowledge Structure

❍ Principles of management science

- Classic management thought
- Architecture of management science
- Fundamental theory of management science

❍ Decision theories

- The mathematical model of decision making
- Decision making processes
- Static decision making strategies
- Game theory
- Decision grid theory

❍ Quality systems

- Quality principles
- Quality control and assurance
- Quality management systems

❍ Software engineering management

- Taxonomy of SE management
- The SE process reference model (SEPRM)

Learning Objectives

- To be aware of fundamental principles and architecture of management science.
- To understand theories of management science, particularly the gains of management and division of labor.
- To understand decision theories for engineering management and the structure of decision strategies and processes.
- To know game theories and the decision grid theory for dynamic and series decision making.
- To understand quality system theories and quality principles for software engineering.
- To be able to apply management theories to software engineering organization, management, and quality assurance.

"Doing the best you can with the resources at your disposal is an age-old problem."

J. Lawrence, Jr. and B. Pasternack (2002)

"Poor management can increase software costs more rapidly than any other factor."

Barry Boehm (1981)

11.1 Introduction

Management science studies how organizations may be operated efficiently, effectively, and profitably on certain constraints of resources and environments. Management is one of the important techniques and professions that emerged in the industrial revolutions in which it was found that management was needed when people worked together to achieve a result not possible by individuals acting alone. Management science as a discipline developed on the basis of a field of study known as *operations research* that proliferated during World War II.

Definition 11.1 *Management science* is the discipline that studies organizational behaviors, executive decision making, and resource optimization on given internal and external constraints.

The objects of study in management science are work, people, resources, and processes. The focal point of management science is productivity and quality. The basic principles of management science are organization, coordination, planning, forecasting, scheduling, and quality assurance. Therefore, formal and empirical theories of coordination, decision making, and quality are the major pursuits of management science.

A profound theory for management science is the formal work organization theory as developed in Chapter 8 represented by Theorems 8.4 through 8.11. Wang's coordinative work organization theory reveals the nature and laws behind human coordination in group work and the approaches for engineering project optimizations. Therefore, it plays a fundamental role in building the formalized theoretical framework of management science.

Historically, software engineering has focused on programming methodologies, programming languages, software development models, and tools. Areas now thought critical to software engineering – organizational and management infrastructures – have been largely ignored. Although the managerial foundations of software engineering had not been widely recognized in software engineering studies and education, this is not to say that management science has not strongly influenced the formation of software engineering as a discipline. In tracing the history of software engineering, it has been found that many of the important concepts of software engineering, such as specification, requirement analysis, design, testing, process, and quality were borrowed from or inspired by the methods and practices developed in management science and other engineering disciplines.

Early software project organization was mostly ad hoc. There were no established processes or best practices. Those that managed or oversaw the work were not traditionally trained managers, rather they could be thought of as accidental managers, who were in charge because they were the best programmers. As it turns out, the skills required for managers do not usually come hand-in-hand with those for writing good software. It becomes evident that software projects follow the same classic project phases as other 'traditional' projects: analysis, design, implementation, and testing. This becomes increasingly evident as software becomes more pervasive throughout all industries, and software projects are now accountable to non-software managers. This has led to increased attention on software project management and organization methodologies as the author and his colleague wrote [Wang and King, 2000a]:

> "In the software industry, the central role is no longer that of the programmers, because project managers and corporate management also have critical roles to play. As programmers require programming technologies, the software corporation managers seek organization and decision making methodologies, and the project managers seek management and software quality assurance methodologies. These needs have together formed the modern domain of software engineering which to summarize includes three important aspects: development methodology, organization, and management."

Therefore, beyond programming and technical aspects of software development, software engineering deals with questions of organization and management infrastructures. The work of the project manager is to balance competing demands for project scope, time, cost, risk, and quality; they must satisfy stakeholders with differing needs and expectations and meet identified requirements.

In the remainder of this chapter, the management science foundations of software engineering will be presented in four sections. Section 11.2

reviews classic management thought, formalizes a set of empirical principles of management science, and introduces the work coordination and organization theory and laws developed by Wang. Section 11.3 presents a comprehensive set of decision theories, especially the latest development of the formal game theory and decision grid theory, which may be used for dynamic decision making in complex real-world applications when a series of interlinked decisions are needed. Section 11.4 describes quality systems for management, in which a formal treatment of quality and its assurance is provided. Section 11.5 deals with complicated management issues in software engineering by introducing the approach of process-based software engineering.

11.2 Principles of Management Science

The development of management as a scientific discipline can be traced back to the work of Frederick Taylor on the improvement of operations in production in the 1890s [Taylor, 1911]. Henry Gantt studied project scheduling and developed the control chart in the 1900s, known as Gantt Chart [Gantt, 1919], for minimizing interrelated job completion times. In the 1920s, William Shewhart introduced statistics into management and developed the control charts for statistical process and quality control [Shewhart, 1939]. In the 1930s, John von Neumann and his colleagues studied strategies in competitive situations known as game theory [von Neumann and Morgenstern, 1980; Osborne and Rubinstein, 1994; Myerson, 1997]. In the 1950s, project scheduling was well studied and the Program Evaluation and Review Technique (PERT) [Dougherty and Stephens, 1984; Hagstrom, 1988; Schmenner and Swink, 1998] and Critical Path Method (CPM) [Kelley, 1961; Schonberger, 1981] were developed. Queuing theory was developed by E. Erlang and John Little in the 1910s and the 1960s, respectively [Little, 1961; Ramaswami and Wirth, 1997]. Various programming methods were proposed to solve optimization problems for a given objective and a number of constraints such as linear programming in the 1940s [Murty, 1983], nonlinear programming and dynamic programming in the 1950s and later [Bertsekas, 1995; Donnelly et al., 1998; Schmenner and Swink, 1998]. Philip Crosby, Edwards Deming, Genichi Taguchi, and Joseph Juran worked on quality systems and developed a number of quality control principles and methodologies in the 1970s and the 1980s [Crosby, 1977; Deming, 1982/86; Taguchi,1986; Juran, 1988/89; Juran et al, 1962/80].

11.2.1 CLASSIC MANAGEMENT THOUGHT

Frederick Taylor's work on *Principles of Scientific Management* published in 1911 inaugurates management as a formal branch of human inquiry on group work and industrial engineering [Taylor, 1911]. Then, classic management thought was further developed in Henri Fayol's work on *General and Industrial Management* in 1929 [Fayol, 1929], and James Mooney's work on *The Principle of Organization* in 1947 [Mooney, 1947].

Henri Fayol was interested in the basic principles of management on determining "soundness and good working order [Fayol, 1929]." Based on experience as the manager of a large coal company in France, Fayol proposed a framework for the art of management with the following principles:

- *Division of labor:* Work can be divided into the smallest feasible elements to take advantage of gain from specialization.
- *Parity of authority and responsibility:* Sufficient authority must be delegated to each jobholder for carrying out assigned job responsibility.
- *Unity of command:* An employee must receive orders from and be accountable to only one superior.
- *Unity of direction:* Activities with the same purpose must be organized together and operated under an integrated plan.
- *Team work:* Employees must be encouraged to unite their effort, goals, and interest with those of the organization. The general interest of the organization takes precedence over those of individuals.
- *Fair remuneration:* Pay must be based on achievement of assigned job objectives.
- *Order:* Each job and its relationship to other jobs must be clearly defined.

- *Equity:* Established rules and agreements must be enforced fairly.
- *Stability of personnel:* Employees must be encouraged to establish loyalty to the organization via a long-term commitment.
- *Initiative:* Employees must be encouraged to exercise independent judgment within their job authority.

James Mooney views organization as the technique of relating specific duties or functions in a coordinated whole. Therefore, management is to devise an appropriate organization [Mooney, 1947]. The classical management thought believed that natural laws of organization and management existed. The objective of management science is to seek the laws and principles for business, industry, and system organization and management. However, despite of a variety of empirical principles and heuristic strategies, a few formal theories and laws has been developed in contemporary management science on the basis of the classic thought of management as reviewed in the beginning of this section.

11.2.2 ARCHITECTURE OF MANAGEMENT SCIENCE

Definition 11.2 *Management* is a coordination process that organizes activities and efforts of a group to achieve goals and results not possible by individuals.

11.2.2.1 Functions of Management

The functions of management identified in management science are planning, organizing, controlling, and optimizing as shown in Fig. 11.1. A manager is responsible for: a) Planning the process, labor, time, resource allocation, and quality requirements of a work; b) Organizing various inputs via the processes to produce certain product or service; c) Controlling individual and process outputs in terms of productivity, costs, and quality; and d) Optimizing the input allocations and organizational objectives via feedback of processes and customers.

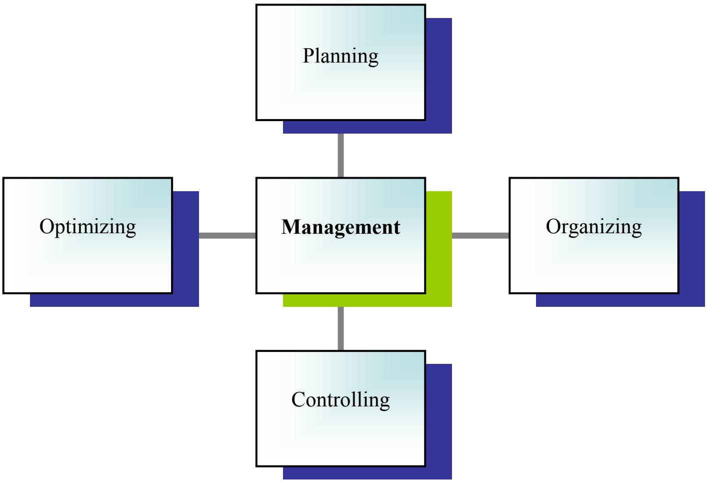


Figure 11.1 The functions of management

Definition 11.3 *Planning* is a management process for organization, coordination, and estimation of project time and related labor and resource allocation.

Planning is required at all levels of management from the strategic level, the technical level, to the operating level. The work products of planning are objectives, decisions, people and resources allocations, and the implementation processes.

The basic techniques for project planning are as follows:

- Partition a given task into detailed subtasks
- Analyze interrelationships between the subtasks, identify parallel and/or serial relations and constraints
- Estimate time needed for completing each subtask
- Decide on the sequence of work allocation (scheduling)
- Define outputs (deliverables) of each task and subtasks
- Assign personnel for each task and subtask
- Allocate resources to each task and subtask

Planning in management encompasses forecasting and scheduling. Formal descriptions of theories and laws on project planning, forecasting, and decision making will be discussed in Sections 11.2.3 and 11.3. The concept of scheduling is briefly introduced below.

Definition 11.4 *Scheduling* is a management process that maps the planned activities onto the time axis in a parallel or serial structure or their combinations.

Project planning and scheduling focus on the overall synchronization and coordination of all processes, tasks, and people working on them. Plans and schedules enable project managers and members to check if any activity is started and/or completed on time and within the scope of allocated resources. Without scheduling, one may realize that a project is late when all planned time and resources have been consumed.

Formal descriptions of theories and laws on project scheduling have been provided in Section 8.5. It is noteworthy that there are natural laws that constrain the allocation of labor and time for a given project. In other words, the optimal allocation of labor, time, and resources is not arbitrary and simply empirical; certain laws and natural constraints exist as described in Section 8.5, particularly by Theorems 8.2, 8.4, and 8.7.

Definition 11.5 *Organizing* is a management process that coordinates and allocates essential means, such as labor, resources, and processes, in order to implement a planned work.

Organizing methodologies play a central rule in management because organization is the major means of management. The fundamental requirement for organization, and theories of optimal allocation of labor and time will be discussed in Section 11.2.3.

Definition 11.6 *Controlling* is a management process that monitors and ensures the planned work process and outcomes in operation conforming to predefined requirements, standards, and schedules.

Controlling is a management process parallel with the production or operation process. When a nonconforming result is identified in the operation, the process should be reviewed and the causality should be identified. The establishment of a quality system is the key methodology for project and organizational controlling in management. Some recurring and systematic inconformity in process may indicate the need to adjust the planned process, technology, labor allocation, and/or the initial schedule.

Definition 11.7 *Optimizing* is a management process that continuously improves the results of an organization or project in terms of higher productivity, better quality, more accurate scheduling, more efficient process, and lower costs.

J. Lawrence, Jr. and B. Pasternack (2002) provided a best explanation of management optimizing that says: “Doing the best you can with the resources at your disposal … .” Management always seeks the optimal solutions, products, services, and systems. Various linear/nonlinear programming models, decision theories, and process improvement methods are developed to support the management activities towards system optimization, which will be described in the remainder of this chapter.

11.2.2.2 The System Model of Management

It is noteworthy that, although there are various objectives in management, the key objective of management science is not *management* but *work*. That is, management science studies how human work may be done coordinately, efficiently, qualitatively, and profitably in a systematic approach [Wang, 2006*l*/06d/07d].

Management science as a system science can be described in Fig. 11.2. In the management system, managers organize and coordinate the production or service processes to transfer the inputs into expected outputs. As shown in Fig. 11.2, the inputs of a management system encompass three essences known as labor, time, and resources; while the outputs of a management system encompass other three essences known as productivity, profit/cost, and quality.

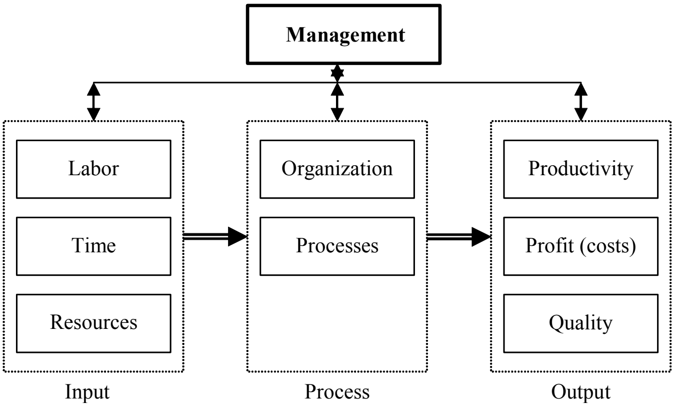


Figure 11.2 Structure of a management system

11.2.3 FUNDAMENTAL THEORY OF MANAGEMENT SCIENCE

Although management has been recognized before the establishment of management science, the foundation of management science is still mainly empirical. Prior to discussing detailed management theories, this section formally examines the fundamental requirement for management in groups, organizations, societies, economies, businesses, and academic research. This leads to the management laws of gain of management and gain of division of labor [Wang, 2005i].

11.2.3.1 Why Management is Needed in Work Organization?

Based on empirical intuition, management has been recognized as a necessary overhead in work organization. The primitive task of management is to *synchronize* the work of a group of people. In an experiment on rebuilding Stonehenge by not using modern tools in England, the project involved hundreds of volunteers. One of the key findings in the project is that a manager is needed as a chanteyman for synchronizing the team. Otherwise, no matter how many people push and pall the huge stone, it will not be moved for an inch.

However, a number of fundamental questions remain unexplained in management science. For instances: Why is management universally needed in work organization? What are the natural laws behind this generic phenomenon? What are the optimal organizational forms in industry and engineering?

The answers to the above questions may be sought by studying and analyzing the nature of working groups of people [Wang, 2005i].

Definition 11.8 A *natural group* is a working group of people with peers in which work is carried out via temporal pairwise coordination when work has to be done by any pair of the peers.

Definition 11.9 A *managed group* is a working group of people with peers and a manager, in which work is carried out via one-to-many coordination by the manager.

Definition 11.10 The *size of a group n* is the number of people working together toward a common goal in production or service.

Definition 11.11 The *number of interpersonal coordination* $C_2(n)$ needed in a natural group of size n, $n \geq 3$, can be determined by:

$$\begin{aligned} C_2(n) &= 2 \bullet \mathrm{C}_{\mathrm{n}}^{2} \\ &= n \bullet (n - 1) \end{aligned} \tag{11.1}$$

where a coordination between peers a and b is asymmetric, i.e., $a \text{ r } b \neq b \text{ r } a$.

When all possible forms of interpersonal coordination in a group is considered, i.e., taking into account of all possible combinations of $\mathrm{C}_\mathrm{n}^\mathrm{k}$ for any k, $0 \leq k \leq n$, the total number of interpersonal coordination $C(n)$ is obtained as follows:

$$\begin{aligned} C(n) &= 2 \bullet \sum_{k=0}^{n} \mathrm{C}_n^k \\ &= 2^{n+1} \end{aligned} \qquad (11.2)$$

Eq. 11.2 results in a geometrical progression series that is exponentially increasing with the size of group n. The complexity is easily to be out of control considering that a ten-person group yields $C(10) = 2^{11} = 2{,}048$ possible forms of required coordination.

Definition 11.12 The *number of interpersonal coordination* $C_m(n)$ needed in a managed group of size n, $n \geq 3$, can be determined by:

$$C_\mathrm{m}(n) = n + 1 \qquad (11.3)$$

where the addition person is the manager.

Definition 11.13 The *management gain* $\Delta m(n)$ of a managed group over a natural group is the difference between the coordination efforts needed in these two organization forms, i.e.:

$$\begin{aligned} \Delta m(n) &= C_2(n) - C_m(n) \\ &= n \bullet (n \text{ - } 1) - \ (n + 1) \\ &= n^2 - 2n - 1 \end{aligned} \qquad (11.4)$$

Definition 11.14 The *management efficiency* $e(n)$ of a managed group over a natural group is a ratio between the management gain and the coordination efforts without management, i.e.:

$$\begin{aligned} e(n) &= \frac{\Delta m(n)}{C_2(n)} \bullet 100\% \\ &= (1 - \frac{c_m(n)}{c_2(n)}) \bullet 100\% \\ &= (1 - \frac{\mathrm{n{+}1}}{n \bullet (n \text{ - } 1)}) \bullet 100\% \end{aligned} \qquad (11.5)$$

Eqs. 11.1 through 11.5 indicate that management is needed because it helps to reduce the complexity of working group organization. The benefit of management in terms of the management gain $\Delta m(n)$ and the management efficiency $e(n)$ are illustrated in Fig. 11.3. It is obvious that $\lim_{n\to\infty} \Delta m(n) = \lim_{n\to\infty} (n^2 - 2n - 1) = \infty$ and $\lim_{n\to\infty} e(n) = \lim_{n\to\infty} (1 - \frac{1}{n}) \bullet 100\%$ = 100% [Wang, 2005i].

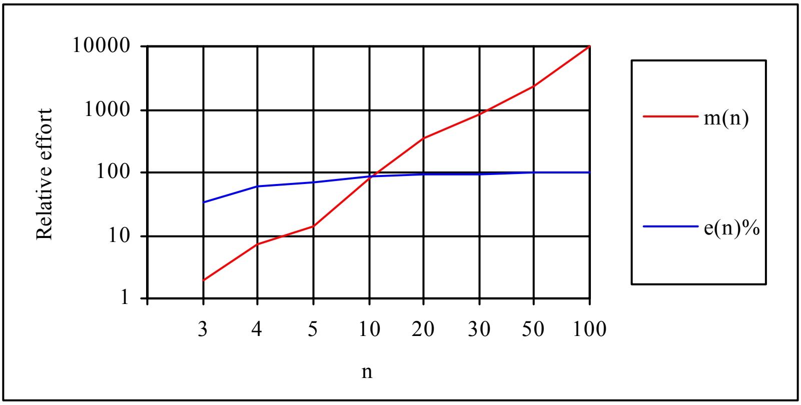


Figure 11.3 Gain and efficiency of management

11.2.3.2 The First Principle of Management

On the basis of the discussions in previous subsection, the relationships among interpersonal coordination in a natural group (Eq. 11.1) and a managed group (Eq. 11.3), as well as the management gain (Eq. 11.4) and the management efficiencies (Eq. 11.5), can be quantitatively analyzed as shown in Table 11.1. This result formally establishes the following principle for management science [Wang, 2005i].

Table 11.1
Gains of Efficiency by Management

N	**1**	**2**	**3**	**4**	**5**	**10**	**20**	**30**	**50**	**100**
C_2(n)	0	2	6	12	20	90	380	870	2450	9900
C_m(n)	2	3	4	5	6	11	21	31	51	101
Δm(n)	-	-	2	7	14	79	359	839	2399	9799
e(n)%	-	-	33.3	58.0	70.0	87.8	94.5	96.4	97.9	99.0

The 42nd Law of Software Engineering

Theorem 11.1 The *gain of management* states that management is required to reduce the complexity of working group organization, to improve the efficiency of groups, and to simplify the forms of interpersonal coordination.

It is empirically observed that management is a necessary overhead of any human-based system. Management functions like a switching center for a group as a system. Without the management, individual component of the system cannot work properly and efficiently.

The natural function of management is system synchronization. Although the basic elements of management are planning, organization, control, and optimization, the essence of all management principles is system synchronization, which is also identified as one of the fundamental principles of system science [Wang, 2005*l*] as described in Chapter 10, as well as in economics [Marshall, 1938].

11.2.3.3 Gains from Division of Labor

The advantage of *division of labor*, also known as *specialization* of skills, has been observed empirically by Confucius (551 – 479BC) in ancient bureaucracy, Adam Smith in economics in 1776 [Smith, 1776], and Frederick Taylor in management science in 1911 [Taylor, 1911]. Therefore, it is perceived that division of labor is the fifth great invention of Chinese civilization to the world in classic management science and sociology in addition to the four great technical inventions known as *typography, papyrus, powder,* and *compass*.

Adam Smith (1723-1790), the proposer of *the invisible hand* in economics, investigated the advantages of specialization during the industrial revolutions [Smith, 1776; Cannan, 1994]. He observed the operation of a pin factory as follows:

> "One workman could scarce, perhaps, with his utmost industry, make ten pin in a day, and certainly could not make twenty."

However, when pin making has become specialized in the factory, said he:

> "One man draws out the wire, another straightens it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving the head, …"

As a result, Smith estimated that ten workers, working in the team in the factory, could produce 48,000 pins a day in the above specialized process. That would be at least 240-fold improvement of productivity than these same ten people could produce while working alone.

Based on this observation, Smith found three distinct advantages of specialization as follows:

- The workers get good at their jobs – better than they would be if they went from one function to another.
- They do not waste time shifting from one task to another.
- Tools or machines may be invented or purchased for the specialized and repetitive work.

Smith's observation on specialization has then been adopted as one of the important principles for improving productivity. It is then called *division of labor* by Marshall in *Principles of Economics* [Marshall, 1938]. This principle is generally true because if all of us got really good at something and concentrated on that one specialty, we could produce much more than we could if we tried to do everything ourselves.

Specialization or division of labor can be implemented in two ways. One way is to use different people to conduct different processes. The other is to repetitively conduct a particular process by one person, and then repeat the next process by the same person. According to cognitive informatics, the latter approach can also save the overhead of cognitive complexity and mental power between the switching of working processes [Wang, 2007j]. Exercise 11.35 provides an interesting example for testing the gain of division of labor in the second approach.

The remainder of this subsection explores the natural laws behind the well known phenomenon of division of labor in many areas of human coordinative work organization.

Definition 11.15 *Division of labor* (DOL), or specialization on a specific subtask in a process, is a work organization method in which a task is divided into a sequence of multiple subtasks, and a person is only specialized in a repeatable subtask.

Work organization by DOL can be illustrated in Fig. 11.4. The first advantage of DOL is that it results in higher productivity and better quality when a work involves only a limited number of subtasks. The second advantage of DOL is that it eases work planning, because labor allocation, replacement, and training are simplified when work is divided into simple and limited basic processes.

To formally explain the potential gains of DOL, the differences of relative effort of an individual spent in specialized work allocation and natural work allocation can be comparatively analyzed below [Wang, 2005i].

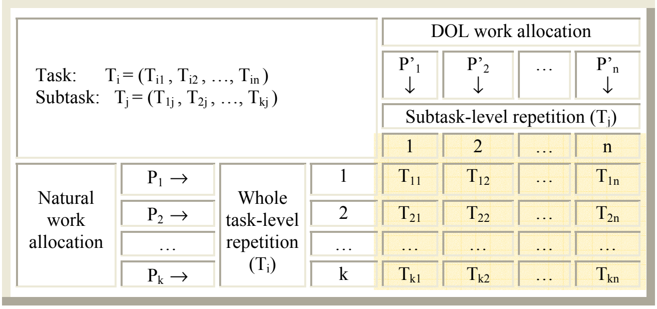

DOL work allocation
Task: $T_i = (T_{i1}, T_{i2}, \ldots, T_{in})$
Subtask: $T_j = (T_{1j}, T_{2j}, \ldots, T_{kj})$
$P'_1 \downarrow$
$P'_2 \downarrow$
…
$P'_n \downarrow$
Subtask-level repetition (T_j)
1
2
…
n
Natural work allocation
$P_1 \rightarrow$
$P_2 \rightarrow$
…
$P_k \rightarrow$
Whole task-level repetition (T_i)
1
2
…
k
T_{11} T_{12} … T_{1n}
T_{21} T_{22} … T_{2n}
… … … …
T_{k1} T_{k2} … T_{kn}

Figure 11.4 Division of labor: labor specializes (repeats) at the subtask-level

Definition 11.16 The *natural work allocation* is a form of loosely coupled work organization that requires an invariable effort $E(1)$ with a relative value 1, i.e.:

$$E(1) = 1 \tag{11.6}$$

Definition 11.17 The *specialized work allocation* $E_{dol}(1)$ is a work organization method that allocates tasks via DOL, which results in the saving of effort proportional to times of repetition k in an inversed exponential rate determined by a constant e/c, i.e.:

$$E_{dol}(1) = (\frac{e}{c})^{k-1} \tag{11.7}$$

where $e = 2.72$ and c is determined empirically based on the skilled rate of repetition for a given task in the range of $1 < c < e$.

Based on Eqs. 11.6 and 11.7, the gain of DOL can be illustrated in Fig. 11.5 where $c = 2.5$. When the relative effort of natural work allocation is set as one, the advantage of DOL is an inversed exponential curve that decreases proportionally to the number of task repetitions k.

Definition 11.18 The *effort of natural work allocation* of a group $E(k)$ is proportional to the number of persons k who is working on the task, i.e.:

$$\begin{aligned} E(k) &= k \bullet E(1) \\ &= k \end{aligned} \tag{11.8}$$

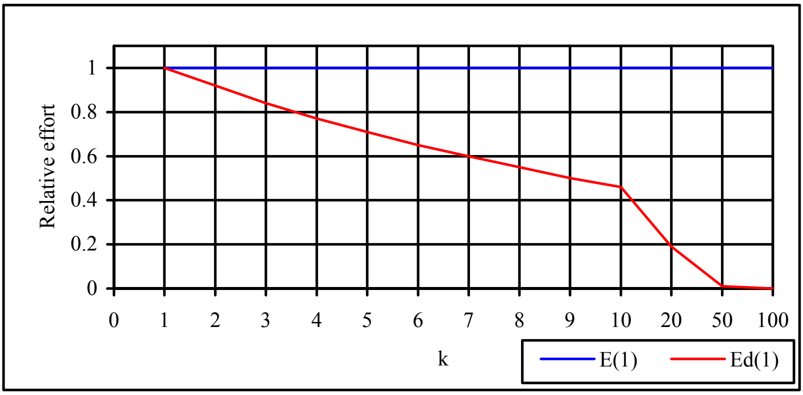


Figure 11.5 Gains of division of labor

Definition 11.19 The *effort of specialized work allocation* of a group $E_{dol}(k)$ is proportional to the repetitive times k in an inversed exponential rate determined by a constant e/c, i.e.:

$$E_{dol}(k) = \sum_{i=1}^{k} \frac{1}{(\frac{e}{c})^{k\text{-}1}} \tag{11.9}$$

where c, $c < e$, is determined empirically based on the specialization rate of a repetitive task.

When the relative values of both E(k) and $E_{dol}(k)$ are determined, the gain of DOL can be described by a ratio or a relative difference between them.

Definition 11.20 The *gain from DOL, g*(k), by specialized work allocation over the natural work allocation is a ratio between the work efforts needed for these two organizational forms, i.e.:

$$\begin{aligned} g(k) &= \frac{E(k)}{E_{dol}(k)} \\ &= \frac{k}{\sum_{i=1}^{k} \frac{1}{(\frac{e}{c})^{k\text{-}1}}} \end{aligned} \tag{11.10}$$

Definition 11.21 The *relative gain from DOL* $g_r(k)$ of a specialized work allocation over the natural work allocation is a ratio between the relative difference and the work efforts needed without DOL, i.e.:

$$\begin{aligned} g_r(k) &= \frac{E(k) - E_{dol}(k)}{E(k)} \bullet 100\% \\ &= (1-\frac{E_{dol}(k)}{E(k)}) \bullet 100\% \qquad (11.11) \\ &= (1-\frac{\sum_{i=1}^{k} \frac{1}{(\frac{e}{c})^{k-1}}}{k}) \bullet 100\% \end{aligned}$$

On the basis of Eqs. 11.10 and 11.11, the simulation results and curves of gains of DOL in forms of the absolute gain $g(k)$ and relative gain $g_r(k)$% are shown in Table 11.2 and Fig. 11.6, respectively, where $c = 2.5$.

Table 11.2
Gains of Division of Labor

k	1	2	3	4	5	6	7	8	9	10	20	50	100
E(k)	1	2	3	4	5	6	7	8	9	10	20	50	100
E_{dol}(k)	1	1.92	2.76	3.53	4.24	4.89	5.49	6.04	6.54	7	9.93	11.87	12.1
g(k)	1	1.04	1.09	1.13	1.18	1.23	1.28	1.32	1.38	1.43	2.01	4.21	8.26
g_r(k)%	0	4	8	11.8	15.2	18.5	21.6	24.5	27.3	30	50.4	76.3	87.9

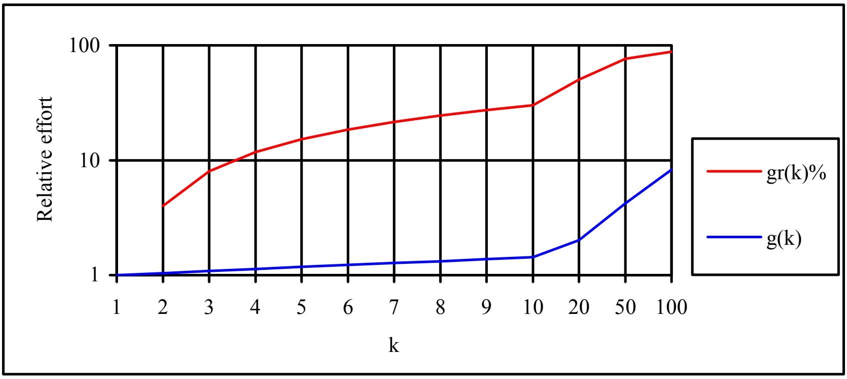


Figure 11.6 Gains of division of labor ($e/c = 1.09$)

When a higher skill rate in specialization is used, i.e., $e/c = 2.72$, the curves of gains via DOL as shown in Fig. 11.6 are increased sharply. In other words, the higher the skill rate via specialization and repetition in a task, the larger the gains via DOL.

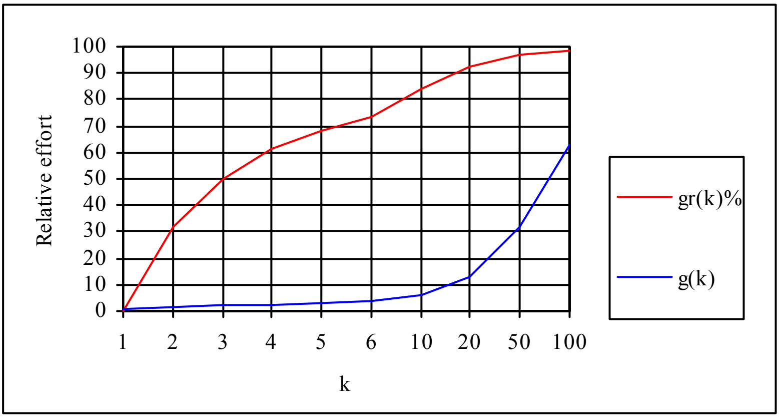


Figure 11.7 Gains of division of labor (*e/c* = 2.72)

11.2.3.4 The Second Principle of Management

Based on the discussions in Section 11.2.3.3, this subsection derives the second principle of management on the benefit of division of labor in work organization [Wang, 2005i].

The 43rd Law of Software Engineering

Theorem 11.2 The *gain of division of labor* states that the relative gain $g_r(k)$ via division of labor in work organization is proportional to the repetitive times k at specialized subtask-level, i.e.:

$$\begin{aligned} g_r(k) &= \frac{E(k) \text{ - } E_{dol}(k)}{E(k)} \bullet 100\% \\ &= (1\text{-}\frac{E_{dol}(k)}{E(k)}) \bullet 100\% \\ &= (1\text{-}\frac{\sum_{i=1}^{k} \frac{1}{(\frac{e}{c})^{k\text{-}1}}}{k}) \bullet 100\% \end{aligned} \qquad (11.12)$$

where c is a positive constant, $1 < c < e$.

The fundamental principles of management as described in both Theorems 11.1 and 11.2 [Wang, 2005i] can be applied in any field of human coordinative work organization. Law 43 can be extended to division of work for multiple subtasks conducted by a single person. That is, as shown in Figs. 11.5 through 11.6, when there are n subtasks needed to be repeated for k times by one person, T_{ij}, $1 \leq i \leq n$, $1 \leq j \leq k$, the person may carry out the work by repeating each subtask $T_j = (\mathrm{T}_{1j}, \mathrm{T}_{2j}, \ldots, \mathrm{T}_{kj})$, $1 \leq j \leq k$, in order to save effort and time.

Corollary 11.1 By adopting division of work, a single person can gain the advantage of division of labor by repetitively working on the same subtask, when a task can be decomposed into a series of subtasks.

11.2.3.5 Wang's Work Organization Theory for Coordinative Work Management

As revealed in Section 11.2.2.2, the ultimate object under study in management science is human work rather than management. Therefore, the theoretical framework for management science is the laws and principles developed so far in Section 11.2.3 [Wang, 2005i] and the formal coordinative work organization theory [Wang, 2007d] as presented in Section 8.5. Human coordination in groups and projects is a widely generic phenomenon studied in large-scale engineering organization in management science, system science, and software engineering [Brooks, 1975; Klir, 1992; Ritzer, 1983]. *Coordinative work* is needed when separated individuals cannot carry out a given work or solve a certain problem. Therefore, a theory of coordinative work organization is at the center of management and system sciences.

Despite of a whole spectrum of empirical studies on the age-old problems that observed the influence and impact of the extent of human coordination to work performance [Fayol, 1929; Mooney, 1947; Ritzer, 1983/93], there was a lack of a rigorous theory of coordinative work organization and management, and the inherent nature of the problem was hidden by too many trivial factors.

The work described in Section 8.5 developed a generic theory of coordinative work organization based on intensive studies in the complicated software engineering environment. The basic properties and characteristics of coordinative work and their mathematical models are established, which explains the transformability between labor and time in coordinative work and the role of the overhead for interpersonal coordination. A set of Wang's laws of abstract work organization is derived in Theorems 8.4 through 8.11, which provide a foundation for rigorously analyzing the work duration and effort in coordinative project organization. The laws have been revealed, for the first time, that the interpersonal coordination rate in groups is the black hole that has resulted in the failures of so many large-scale projects due to the exponential growing of unexpected actual workload under nonoptimal labor and work allocation.

On the basis of the coordinative work organization theory, a set of decision optimization strategies can be derived towards optimal project organization for the best labor allocation, the shortest project duration, and the lowest effort. The exchangeability and its constraints between labor and time in work organization are formally explained.

11.3 Decision Theories

Decision making is one of the basic cognitive processes of human brains [Wang et al., 2006; Wang and Ruhe, 2007d], by which a preferred option or a course of actions is chosen from among a set of alternatives based on certain criteria. Decision theories are widely applied in a number of disciplines encompassing cognitive science, computer science, management science, economics, sociology, psychology, political science, and statistics. A number of decision strategies have been proposed from different angles and application domains such as the maximum expected utility and Bayesian method. However, there is still a lack of a fundamental and mathematical decision model and a rigorous cognitive process for decision making.

In this section, a decision making process is modeled as a sequence of Cartesian-product based selections. A rigorous description of the fundamental decision process in RTPA is presented. Different decision making strategies are comparatively analyzed. The result shows these strategies can be well fit in the formally described decision process. The cognitive process of decision making may be applied in a wide range of decision-based systems, such as cognitive informatics, software agent systems, expert systems, and decision support systems.

A decision making process chooses a preferred option or a course of actions from among a set of alternatives on the basis of given criteria or strategies [Simon, 1960; Wilson and Keil, 2001; Wang and Ruhe, 2007d]. Decision making is one of the 39 fundamental cognitive processes modeled in LRMB [Wang et al., 2006]. The study on decision making is interested in multiple disciplines, such as cognitive informatics, cognitive science, computer science, psychology, management science, economics, sociology, political science, and statistics [Wald, 1950; Berger, 1990; Pinel, 1997; Matlin, 1998; Payne and Wenger, 1998; Edwards and Fasolo, 2001; Hastie, 2001; Wilson and Keil, 2001; Wang et al., 2006]. Each of those disciplines has emphasized on a special aspect of decision making. It is recognized that there is a need to seek an axiomatic and rigorous model of the cognitive decision-making process in the brain, which may be served as the foundation of various decision making theories. This approach is based on the basic understanding that, although the cognitive capacities of decision makers may be greatly varying, the core cognitive processes of the human brain share similar and recursive characteristics and mechanisms [Wang et al., 2006; Wang and Ruhe, 2007].

Decision theories can be categorized into two paradigms: the *descriptive* and *normative* theories. The former is based on empirical observation and on experimental studies of choice behaviors; and the latter assumes a rational decision-maker who follows well-defined preferences that obey certain axioms of rational behaviors. Typical normative theories are the expected utility paradigm [Osborne and Rubinstein, 1994] and the Bayesian theory [Wald, 1950; Berger, 1990]. W. Edwards and B. Fasolo proposed a 19-step decision making process [Edwards and Fasolo, 2001] by integrating Bayesian and multi-attribute utility theories. W. Zachary and his colleagues [Zachary et al., 1982] perceived that there are three constituents in decision making known as the *decision situation*, the *decision maker*, and the *decision process*. Although the cognitive capacities of decision makers may be greatly varying, the core cognitive processes of the human brain share similar and recursive characteristics and mechanisms [Wang, 2003a; Wang and Gafurov, 2003; Wang et al., 2006; Wang and Ruhe, 2007].

An overview of the taxonomy and classification of decision theories and related rational strategies that will be discussed in this section can be illustrated as shown in Fig. 11.8. Fig. 11.8 can be used as a guideline for studying the whole framework of decision theories that will be extended in the following subsections.

11.3.1 THE MATHEMATICAL MODEL OF DECISION MAKING

Decision making as one of the fundamental cognitive processes of human beings is widely used in determining rational, heuristic, and intuitive selections in complex scientific, engineering, economical, and management situations, as well as in almost each procedure of daily life. Since decision making is a meta mental process, it occurs every few seconds in the thinking courses of human minds consciously or subconsciously.

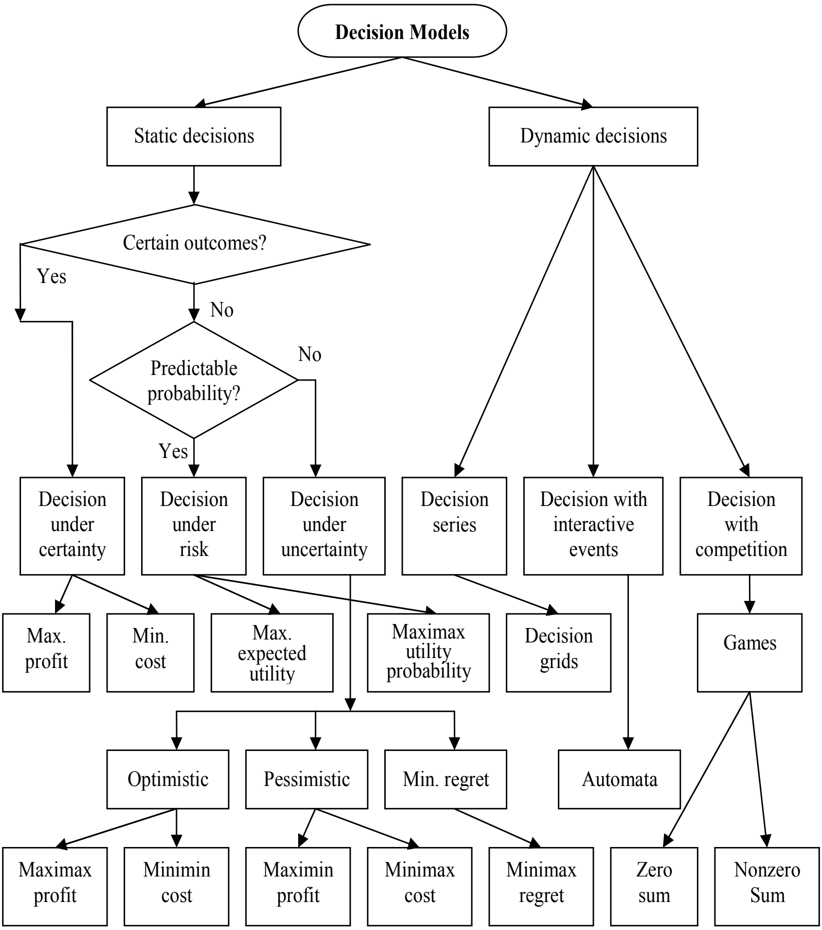


Figure 11.8 Overview of decision theories and decision strategies

This subsection explores the nature of selection, decision, and decision making, and their mathematical models. A rigorous description of decision making and its strategies is developed.

11.3.1.1 The Principle of Choices

The philosophy of the *axiom of choice* [Lipschutz, 1964] is adopted to describe decision theories, which identifies the following three essences for decision making known as the *decision goals*, a set of *alternative choices*, and a set of *selection criteria* or strategies. According to this theory, decision makers are only the engine or executive of a decision making process. If the three essences of decision making are defined, a decision making process may be rigorously carried out by either a human decision maker or by an intelligent system. This is a cognitive foundation for implementing expert systems, agent systems, and decision supporting systems [Wang et al., 2006; Wang, 2007a].

Definition 11.22 Let I be a nonempty indexing set, S be a collection of sets, $\{A_i \mid i \in I\}$ be a collection of disjoint sets, $A_i \subseteq S$, and $A_i \neq \varnothing$. A *choice function c* can be described as:

$$c\text{: } \{A_i \mid i \in I\} \rightarrow A_i \tag{11.13}$$

where $c(A_i) = a_i,\ a_i \in A_i \subseteq S,\ U$ the universal set, I a set of natural numbers, and A_i is called the set of alternatives.

Lemma 11.1 The *axiom of choice selection* states that there exists a choice function for any nonempty collection S of nonempty disjoint sets of alternatives $A_i \subseteq S$, $i \in I$ [Lipschutz, 1964].

In addition to the axiom of choice, the *additive* and *multiplicative* principles of choices given below are useful when solving composed decision problems.

Lemma 11.2 The *additive principle of choices* states that the number of choices between two arbitrary sets of alternatives *A or B* is the *sum* of all alternatives provided in them, i.e.:

$$\#(A \vee B) = \#A + \#B \tag{11.14}$$

Lemma 11.3 The *multiplicative principle of choices* states that the number of choices between two arbitrary sets of alternatives *A and B* is the *product* of the number of all alternatives provided in both of them, i.e.:

$$\#(A \wedge B) = \#A \bullet \#B \tag{11.15}$$

11.3.1.2 Decisions and Decision Making

On the basis of the axiom and function of choice, a decision can be rigorously defined as follows.

Definition 11.23 A *decision d* is a selected alternative $a \in \mathcal{A}$ from a nonempty set of alternatives $\mathcal{A}$, $\mathcal{A} \subseteq U$, based on a given set of criteria C, i.e.:

$$\begin{aligned} d &= f(\mathcal{A}, C) \\ &= f\colon \mathcal{A} \times C \to \mathcal{A},\ \ \mathcal{A} \subseteq U, \mathcal{A} \neq \varnothing \end{aligned} \tag{11.16}$$

where × represents a Cartesian product.

It is noteworthy that the choice criteria C can be a simple one or a complex one. The latter is the combination of a number of joint criteria depending on multiple factors.

Definition 11.24 *Decision making* is a process of decision selection from available alternatives against the chosen criteria for a given decision goal.

According to Definition 11.24, the *number of possible decisions*, n, can be determined by the sizes of $\mathcal{A}$ and C, i.e.:

$$n = \#\mathcal{A} \bullet \#\mathrm{C} \tag{11.17}$$

where # is the cardinal calculus on sets, and $\mathcal{A} \cap C = \varnothing$.

Eq. 11.17 indicates that in case $\#\mathcal{A} = 0$ and/or $\#C = 0$, no decision may be derived for the given case.

The above definitions provide a generic and fundamental mathematical model of decision making, which reveal that the factors determining a decision are the alternatives $\mathcal{A}$ and criteria C for a given decision making goal. A unified theory on fundamental and cognitive decision making can be developed based on the axiomatic and recursive cognitive process elicited from the simplest decision-making categories as shown in Table 11.3.

11.3.1.3 Strategies and Criteria for Decision Making

According to Definition 11.24, the outcome of a decision making process is determined by the decision-making strategies selected by decision makers, when a set of alternative decisions has been identified. It is obvious that different decision making strategies require different decision selection criteria. There is a great variation of decision-making strategies developed in traditional decision and game theories, as well as cognitive science, system science, management science, and economics.

The taxonomy of strategies and corresponding criteria for decision making may be classified into four categories known as *intuitive, empirical, heuristic,* and *rational* as shown in Table 11.3 [Wang and Ruhe, 2007]. It is noteworthy in Table 11.3 that the existing decision theories provide a set of criteria (*C*) for evaluating alternative choices for a given problem.

As summarized in Table 11.3, the first two categories of decision making, *intuitive* and *empirical*, are in line with human intuitive cognitive psychology and there is no specific rational model for explaining those decision criteria. The *rational* decision making strategies will be described by two subcategories, the static and dynamic strategies and criteria, in Sections 11.3.3 and 11.3.4, respectively. The *heuristic* decision-making strategies are frequently used by human beings as a decision maker. Details of the heuristic decision-making strategies may be referred to cognitive psychology and AI [Matlin, 1998; Payne and Wenger, 1998; Hastie, 2001].

It is interesting to observe that the most simple decision making theory can be classified into the intuitive category, such as arbitrary and preference choices based on personal propensity, hobby, tendency, expectation, and/or common senses. That is, not necessarily to be an expert, a layperson may still be able to make important and perhaps wise decisions every day, even every few seconds. Therefore, the elicitation of the most fundamental and core process of decision making shared in human cognitive processes is yet to be sought. Recursive applications of such a core process of decision making will be helpful to solve complicated decision problems in the real-world.

11.3.1.4 The Structure of Rational Decision Making

According to Table 11.3, rational and complex decision making strategies can be classified into the static and dynamic categories. Most existing decision-making strategies are static because the changes of environments of decision makers are independent of the decision makers' activities. Also, different decision strategies may be selected in the same situation or environment based on the decision makers' values and attitudes towards risk and their prediction on future outcomes. When the environment of a decision maker is interactive with his/her decisions or the environment changes according to the decision makers' activities and the decision strategies and rules are predetermined, this category of decision making needs are classified into the category of dynamic decisions, such as games and decision grids [von Neumann and Morgenstern, 1980; Osborne and Rubinstein, 1994; Wang, 2005b/05e].

Table 11.3
Taxonomy of Strategies and Criteria for Decision Making

No.	Category	Strategy	Criterion
1	**Intuitive**		
1.1		Arbitrary	Based on the most easy or familiar choice
1.2		Preference	Based on propensity, hobby, tendency, or expectation
1.3		Common senses	Based on axioms and judgment
2	**Empirical**		
2.1		Trial and error	Based on exhaustive trial
2.2		Experiment	Based on experiment results
2.3		Experience	Based on existing knowledge
2.4		Consultant	Based on professional consultation
2.5		Estimation	Based on rough evaluation
3	**Heuristic**		
3.1		Principles	Based on scientific theories
3.2		Ethics	Based on philosophical judgment and belief
3.3		Representative	Based on common rules of thumb
3.4		Availability	Based on limited information or local maximum
3.5		Anchoring	Based on presumption or bias and their justification
4	**Rational**		
4.1	Static		
4.1.1		Minimum cost	Based on minimizing energy, time, money
4.1.2		Maximum benefit	Based on maximizing gain of usability, functionality, reliability, quality, dependability
4.1.3		Maximum utility	Based on cost-benefit ratio
4.1.3.1		- Certainty	Based on maximum probability, statistic data
4.1.3.2		- Risks	Based on minimum loss or regret
		- Uncertainty	
4.1.3.3		- Pessimist	Based on maximin
4.1.3.4		- Optimist	Based on maximax
4.1.3.5		- Regretist	Based on minimax of regrets
4.2	Dynamic		
4.2.1		Interactive events	Based on automata
4.2.2		Games	Based on conflict
4.2.2.1		- Zero sum	Based on $\sum$ (gain + loss) = 0
4.2.2.2		- Non zero sum	Based on $\sum$ (gain + loss) $>$ 0
4.2.3		Decision grids	Based on a series of choices in a decision grid

Definition 11.25 The *dynamic strategies and criteria* of decision making are those that all alternatives and criteria are dependent on both the environment and the effect of the historical decisions made by the decision maker.

Classic dynamic decision making methods are decision trees [Friedman, 1996; Edwards and Fasolo, 2001]. A new theory of decision grids is developed in [Wang, 2005b] for serial decision makings. Decision making under interactive events and competition is modeled by games [von Neumann and Morgenstern, 1980; Osborne and Rubinstein, 1994; Wang, 2005b/05e]. Wang (2005e) presents a formal model of games, which rigorously describes the architecture or layout of games and their dynamic properties and behaviors.

Decision making is the process of constructing the choice criteria (or functions) and strategies and using them to select a decision from a set of possible alternatives. In this view, existing decision theories are about how a choice function may be created for finding a good decision. Different decision theories provide different choice functions.

An overview of the classification of decisions and related rational strategies has been provided in Fig. 11.8. It can be seen that games are used to deal with the most complicated decision problems, which are dynamic, interactive, and under uncontrollable competitions. Decision models may also be classified among other points of views such as structures, constraints, degrees of uncertainty, clearness and scopes of objectives, difficulties of information processing, degrees of complexity, utilities and beliefs, ease of formalization, time constraints, and uniqueness or novelty.

11.3.2 DECISION MAKING PROCESSES

Decision making is one of the fundamental cognitive processes modeled in LRMB [Wang et al., 2006]. The decision making process can be explained based on the OAR model, *OAR* = (*O, A, R*), as developed in Section 9.4.2, where *O* is a given set of objects identified by an abstract name, $\mathcal{A}$ is a set of attributes for characterizing the object, and *R* is a set of relations between the object and other objects or attributes of them.

11.3.2.1 The Cognitive Process of Decision Making

On the basis of the LRMB [Wang et al., 2006] and OAR [Wang, 2007g] models developed in Chapter 9, the cognitive process of decision making may be informally described by the following procedures:

a) To comprehend the decision making problem, and to identify the decision goal in terms of an Object (*O*) and its attributes (*A*).

b) To search in the abstract layer of LTM for alternative solutions ($\mathcal{A}$) and criteria or useful decision strategies (C).

c) To quantify $\mathcal{A}$ and C, and determine if the search should be going on.

d) To build a set of decisions by using $\mathcal{A}$ and C as obtained in above searches.

e) To select the preferred decision(s) on the basis of satisfaction of decision makers.

f) To represent the decision(s) in a new sub-OAR model.

g) To memorize the sub-OAR model in LTM.

A detailed cognitive process model of decision making is shown in Fig. 11.9, where a double-ended rectangle block represents a function call that involves a predefined process as provided in the LRMB model.

The first step is to understand the decision-making problem. According to the cognitive process of comprehension [Wang and Gafurov, 2003], the object (goal) of decision will be identified, and an initial sub-OAR model will be created. The object, its attributes, and known relations are retrieved and represented in the sub-OAR model. Then, alternatives and strategies are searched, which result in two sets of A_i and C, respectively. The results of search will be quantified in order to form a decision as defined in Eq. 11.16, i.e., $d = f\!: \mathcal{A} \times C \to \mathcal{A}$, where $i \in I$, $\mathcal{A} \subseteq S$, and $\mathcal{A} \neq \varnothing$.

When the decision d is derived, the initial sub-OAR model will be updated with d and related information. Then, the decision maker may consider whether the decision is satisfied according to the current states of nature and personal judgment. If yes, the sub-OAR model for the decision is memorized in the LTM. Otherwise, the decision-making process has to be repeated until a satisfied decision is found, or the decision maker chooses to quit with no satisfied decision. During the decision making process, the mental states of the decision maker, the global OAR model in the brain, changes from time to time. Although the state of nature will not be changed in a short period during decision making, its perception may be changed with the effect of the updating OAR model.

The process of decision making is a higher-layer cognitive process defined at Layer 6 of LRMB. The decision making process interacts with other processes underneath this layer such as *Search, Representation* and *Memorization*; and the processes at the same layer such as *Comprehension, Qualification, Quantification,* and *Problem solving*. Relationships between the decision-making process and other related processes have been described in Section 9.3.1 [Wang et al. 2006].

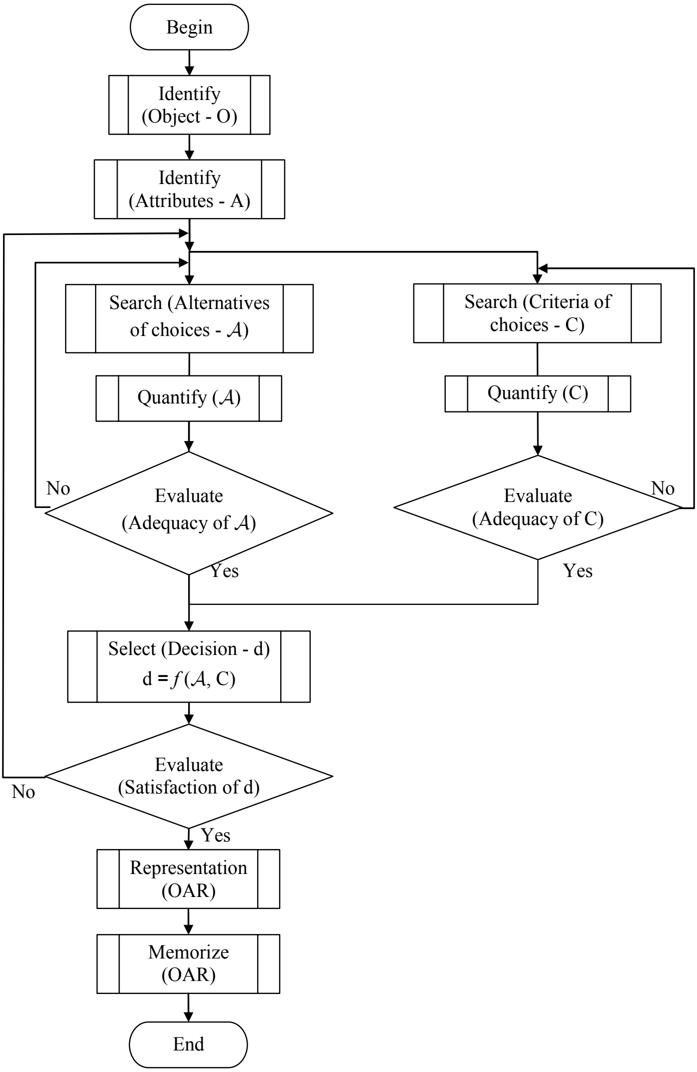

Begin
Identify (Object - O)
Identify (Attributes - A)
Search (Alternatives of choices - $\mathcal{A}$)
Quantify ($\mathcal{A}$)
Evaluate (Adequacy of $\mathcal{A}$)
No
Yes
Search (Criteria of choices - C)
Quantify (C)
Evaluate (Adequacy of C)
No
Yes
Select (Decision - d) d = f($\mathcal{A}$, C)
Evaluate (Satisfaction of d)
No
Yes
Representation (OAR)
Memorize (OAR)
End

Figure 11.9 The cognitive process of decision making

11.3.2.2 Formal Description of the Decision Making Process

On the basis of the cognitive model of decision making as described in Fig. 11.9, a rigorous cognitive process can be specified using RTPA [Wang, 2002a]. RTPA is designed for describing the architectures, static, and dynamic behaviors of software systems as well as human cognitive behaviors and sequences of actions.

The formal model of the cognitive process of decision making in RTPA is presented in Fig. 11.10. According to the LRMB and OAR models of internal knowledge representation in the brain, the result of a decision in the mind of the decision maker is a new sub-OAR model, or an updated version of the global OAR model of knowledge in the human brain.

```
The Decision Making Process

Decision_Making (I:: O$\mathbf{S}$; O:: OAR(d$\mathbf{S}$)$\mathbf{ST}$) $\triangleq$
{ //  I. Form decision goal(s)
$\rightarrowtail$ Identify (O)                 // The decision making goal
$\rightarrowtail$ Identify (A)                 // Sub decision making goals
$\rightarrow$ {  $\overset{\mathbf{T}}{\underset{\text{Satisfication of As} = \mathbf{F}}{R}}$   ( $\rightarrowtail$ Search ($\mathcal{A}$)
                                   $\rightarrowtail$ Quantify ($\mathcal{A}$)
                                   $\rightarrowtail$ Evaluate ($\mathcal{A}$)
                                  )
    ||  $\overset{\mathbf{T}}{\underset{\text{Satisfication of C} = \mathbf{F}}{R}}$   ( $\rightarrowtail$ Search (C)
                                   $\rightarrowtail$ Quantify (C)
                                   $\rightarrowtail$ Evaluate (C)
                                  )
  }

//  II. Select decisions
$\rightarrow$ d = $f$: $\mathcal{A} \times$ C $\rightarrow \mathcal{A}$            // Refers to Eq. 11.6
$\rightarrowtail$ Evaluate (d)
$\rightarrow$ ( � s(d) $\geq$ k                     // k: a satisfaction threshold
        $\rightarrowtail$ Memorize (OAR$\mathbf{ST}$)
        $\rightarrow \otimes$
   | � ~                               // Otherwise
        $\rightarrow$ (  � GiveUp$\mathbf{BL}$ = $\mathbf{F}$
                 $\rightarrowtail$ DMP_Process(I:: O$\mathbf{S}$; O:: OAR(d)$\mathbf{ST}$)
            | � ~
                 $\rightarrow \otimes$
          )
   )

//  III. Represent decisions
   $\rightarrow$ R = <d, $\mathcal{A}$, C>            // Form new relation on d
   $\rightarrow$ OAR$\mathbf{ST}$ = <O, A, R>         // Form new OAR model for d
}
```

Figure 11.10 The RTPA definition of the cognitive process of decision making

As shown in Fig. 11.10, a decision-making process is started by defining the goal of decision in terms of the object and attributes. Then, an exhaustive search of the alternative decisions ($\mathcal{A}$) and useful criteria (C) are carried out in parallel. The searches are conducted in both the brain of a decision maker internally, and through external resources based on the knowledge, experiences, and goal expectation. The results of searches are quantitatively evaluated until the searching for both $\mathcal{A}$ and C is satisfied. If nonempty sets are obtained for both $\mathcal{A}$ and C, the n decisions in d have already existed as determinable by Eq. 11.17.

One or a number of suitable decisions are selected from the set of d by decision makers via evaluating the satisfaction levels. Satisfied decisions will be represented in a sub-OAR model, which will be added to the entire knowledge of the decision maker in LTM by the process of memorization.

11.3.3 STATIC DECISION MAKING STRATEGIES

In the previous section it has been seen that the strategies and selection criteria are vital in a decision making process, particularly for making rational and complex decisions [Simon, 1960; Wang and Ruhe, 2007]. The rational strategies and criteria for decision making can be classified into static and dynamic decisions. This section describes common static decision making strategies and their evaluation criteria. The dynamic ones will be discussed in the following sections.

Definition 11.26 A *static strategy and criterion* of decision making is an evaluation and selection method for which all alternatives $\mathcal{A}$ and criteria C are determinable and only one optimal decision $a_i \in \mathcal{A}$ is expected for a given situation.

Let us consider three typical static decision making strategies known as *decision making under certainty, risks,* and *uncertainty*. The latter may be further divided into the pessimistic, optimistic, and regret decision making under uncertainty according to Fig. 11.8.

It is noteworthy that practical decisions for a given problem are usually made under partial certainty, empirical estimation, or heuristic prediction, because not all required information is available, no suitable decision strategy is aware of, and/or no acceptable cost to thoroughly search all possible alternatives. This observation can be formally described as the principle of bounded rationality [Simon, 1957] in decision making.

Lemma 11.4 The principle of *bounded rationality* states that a decision-maker in a real-world situation will never have all information necessary for making an optimal decision.

A convenient technique to represent the conditions for a decision is using a matrix called payoff table, which lists the value of utilities or costs of all alternative decisions against different situations known as the *states of nature*.

Definition 11.27 A *payoff table* is a 2-D matrix as shown in Table 11.4 that quantifies the utility, value, or level of satisfaction, u_{ij}, for each given pair of alternative a_i and situation s_j, where $1 \leq i \leq n$, and $1 \leq j \leq k$.

Table 11.4
The Structure of a Payoff Table

Alternative ($\mathcal{A}$)	**Situation (S)**			
	s_1	**s_2**	**...**	**s_k**
a_1	u_{11}	u_{12}	…	u_{1k}
a_2	u_{21}	u_{22}	…	u_{2k}
...	…	…	…	…
a_n	u_{n1}	u_{n1}	…	u_{nk}

Example 11.1 Consider a decision making problem for a software engineering project. This project is designed with three alternative *Architectures* a_1 to a_3. Each architecture would be implemented in different *Results* s_1 to s_4 that are predictable or unpredictable in various situations.

The payoff table of this project with utilities (gains) in k$ can be described as shown in Table 11.5.

Table 11.5
The Payoff Table of a Software Engineering Project

Alternative ($\mathcal{A}$)	**Situation (S)**			
	Result 1 (s_1)	**Result 2 (s_2)**	**Result 3 (s_3)**	**Result 4 (s_4)**
Architecture (a_1)	100	10	40	60
Architecture (a_2)	-10	50	200	30
Architecture (a_3)	50	20	5	130

The following subsection will take the above software engineering project as an example to illustrate how a wide range of decision making strategies and criterion can be used in the decision making process.

11.3.3.1 Decision Making under Certainty

Decision making under certainty is an ideal situation where all necessary information and strategies are available, and the outcomes are objectively determinable.

Definition 11.28 A *decision making under certainty* d_{max} or d_{min} is a selection of an certain alternative a_i among $\mathcal{A}$ that meets a given criterion C which is either the maximum of utility or profit $max(u_i)$, and the minimum of costs or effort $min(e_i)$, i.e.:

$$\begin{aligned} d_{max} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (u_i) \wedge a_i \in \mathcal{A}\} \end{aligned} \tag{11.18a}$$

or

$$\begin{aligned} d_{min} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid min\ (e_i) \wedge a_i \in \mathcal{A}\} \end{aligned} \tag{11.18b}$$

Example 11.2 Consider the software engineering project given in Example 11.1. When the criterion C is to take the maximum utility, and the project will definitely achieve Result 4 under the certain situation, which system architecture should be selected for this project?

The answer under the given conditions is direct forward according to Eq. 11.18a since the criterion C is to maximize the project gain, i.e.:

$$\begin{aligned} d_{max} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (u_{14}, u_{24}, u_{34})\} \\ &= \{a_3 \mid u_{34} = 130\} \end{aligned}$$

The solution indicates that the optimal decision for this given project with the maximum criterion is ($\boldsymbol{a_3}$, s_4), which will result in a maximum project gain $u_{max} = u_{34} = \$130{,}000$.

11.3.3.2 Decision Making under Uncertainty

Definition 11.29 A *decision making under uncertainty* is a selection of an alternative a_i among $\mathcal{A}$ that meets a given criterion C, when the probability of each possible situation is unknown.

The strategies for decision making under uncertainty can be divided into three categories known as the optimistic, pessimistic, and regret decisions.

11.3.3.2.1 Optimistic Decision Making under Uncertainty

When the occurrence probabilities of possible future situations or states of the nature are unknown, one of the solutions in decision making is based on an optimistic or aggressive strategy to try to gain the maximum utility or to spend the minimum cost.

Definition 11.30 An *optimistic decision making under uncertainty* $d_{maximax}$ or $d_{minimin}$ yields a decision with the *maximum-maximum* strategy for utility or a *minimum-minimum* strategy for cost, respectively, i.e.:

$$\begin{aligned} d_{maximax} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\,(max\,(u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k)\} \qquad (11.19a) \end{aligned}$$

or

$$\begin{aligned} d_{minimin} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid min\,(min\,(u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k\,)\} \qquad (11.19b) \end{aligned}$$

Example 11.3 Consider the software engineering project given in Example 11.1. A maximax or an optimistic uncertainty decision can be made based on the project gains for different architecture-result combinations as shown in Table 11.6.

Table 11.6
Maximax Decision Making for the Software Engineering Project

Alternative ($\mathcal{A}$)	Situation (*S*)				Criterion (maximax utility)
	Result 1 (s_1)	Result 2 (s_2)	Result 3 (s_3)	Result 4 (s_4)	
Architecture (a_1)	100	10	40	60	
Architecture (a_2)	- 10	50	200	30	u_{23} = $200k
Architecture (a_3)	50	20	5	130	

According to Eq. 11.19a, the maximax decision under uncertainty is as follows:

$$\begin{aligned} d_{maximax} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (max\ (u_{ij} \mid 1 \le i \le 3) \mid 1 \le j \le 4)\} \\ &= \{a_i \mid max\ (u_{11}, u_{23}, u_{34})\} \\ &= \{a_2 \mid u_{23} = 200\} \end{aligned}$$

The solution indicates that the optimal decision for this given project with the maximax criterion is ($\boldsymbol{a_2}$, $\boldsymbol{s_3}$) that will result in a maximum project gain $u_{max} = u_{23} =$ \$200,000.

It is noteworthy that, by choosing this solution, there is a risk to lose \$10,000 if the uncertain project outcomes turn out to be Result 1 with u_{21}. A more conservative but safe decision without any loss for this project may be made based on another decision strategy as discussed in the next subsection.

11.3.3.2.2 Pessimistic Decision Making under Uncertainty

When the occurrence probabilities of possible future situations or states of the nature are unknown, another solution in decision making is based on a conservative or pessimistic strategy to try to gain the maximum utility or to spend the minimum cost.

Definition 11.31 A *pessimistic decision making under uncertainty* $d_{maximin}$ or $d_{minimax}$ yields a decision with the *maximum-minimum* strategy for utility or a *minimum-maximum* strategy for cost, i.e.:

$$\begin{aligned} d_{maximin} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (min\ (u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k)\} \qquad (11.20a) \end{aligned}$$

or

$$\begin{aligned} d_{minimax} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid min\ (max\ (u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k)\} \qquad (11.20b) \end{aligned}$$

Example 11.4 Consider the software engineering project given in Example 11.1. A maximin or a pessimistic uncertainty decision can be made based on the project gains for different architecture-result combinations as shown in Table 11.7.

Table 11.7
Maximin Decision Making for the Software Engineering Project

Alternative ($\mathcal{A}$)	Situation (S)				Criterion (maximin utility)
	Result 1 (s_1)	Result 2 (s_2)	Result 3 (s_3)	Result 4 (s_4)	
Architecture (a_1)	100	10	40	60	$u_{12} = \$10k$
Architecture (a_2)	- 10	50	200	30	
Architecture (a_3)	50	20	5	130	

According to Eq. 11.20a, the maximin decision under uncertainty is as follows:

$$\begin{aligned} d_{maximax} &= f\!: \mathcal{A} \times C \rightarrow \mathcal{A} \\ &= \{a_i \mid max\ (min\ (u_{ij} \mid 1 \le i \le 3) \mid 1 \le j \le 4)\} \\ &= \{a_i \mid max\ (u_{12}, u_{21}, u_{33})\} \\ &= \{a_1 \mid u_{12} = 10\} \end{aligned}$$

The solution indicates that the conservative decision for this given project with the maximin criterion is (a_1, s_2), which will result in a maximin project gain $u_{max} = u_{12} = \$10,000$.

It is noteworthy that, by choosing this solution, there is a chance to lose the opportunity gain of \$200,000 if the uncertain project outcomes turn out to be Result 3 with u_{23}. However, in any case, this decision can prevent the project from a negative result.

11.3.3.2.3 Minimum Regret Decision Making under Uncertainty

As discussed in the proceeding subsection, when the conservative strategy is taken, a decision under uncertainty may result in a loss of a better opportunity. The loss of an opportunity gain or an opportunity save of costs is called a *regret*. A strategy exists for minimizing the regret in decision making under uncertainty.

Definition 11.32 A *regret* r_{ij} is the loss of the best opportunity by selecting a conservative decision under uncertainty, i.e.:

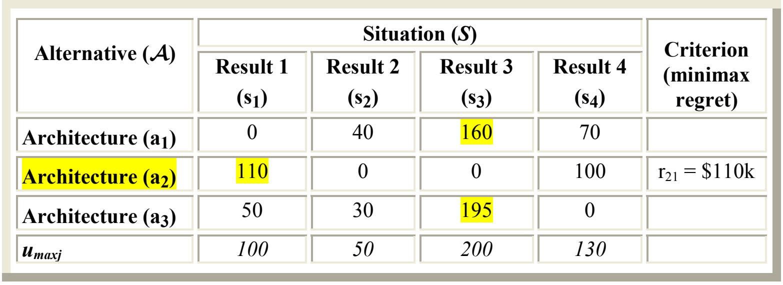
$$r_{ij} = u_{maxj} - u_{ij}\,,\ 1 \le j \le \mathrm{k},\ 1 \le i \le n \qquad (11.21)$$

Definition 11.33 A *minimum regret decision making under uncertainty* $d_{minimax}$ yields a decision with the *minimum-maximum regret* strategy for utility gain or cost save, i.e.:

$$\begin{aligned} d_{minimax} &= f: \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid min\ (max\ (r_{ij} \mid 1 \leq i \leq n)\} \end{aligned} \quad (11.22)$$

Example 11.5 Consider the software engineering project given in Example 11.1. A minimax regret decision under uncertainty can be made based on the regret r_{ij} determined by Eq. 11.22 for different architecture-result combinations as shown in Table 11.8.

Table 11.8
Minimax Regret Decision Making for the Software Engineering Project



Alternative ($\mathcal{A}$)	Situation (S)				Criterion (minimax regret)
	Result 1 (s_1)	Result 2 (s_2)	Result 3 (s_3)	Result 4 (s_4)	
Architecture (a_1)	0	40	160	70	
Architecture (a_2)	110	0	0	100	r_{21} = $110k
Architecture (a_3)	50	30	195	0	
u_{maxj}	*100*	*50*	*200*	*130*	

According to the above regret payoff table and Eq. 11.22, the minimax regret decision under uncertainty is as follows:

$$\begin{aligned} d_{minimax} &= f: \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid min\ (max\ (r_{ij} \mid 1 \leq i \leq n)\} \\ &= \{a_i \mid min\ (r_{13}, r_{21}, r_{33})\} \\ &= \{a_2 \mid r_{21} = 110\} \end{aligned}$$

The solution indicates that the decision for this given project with the minimax regret criterion under uncertainty is (a_2, s_1), which will result in a minimum regret for possible lost opportunities $r_{min} = r_{21} = \$110,000$.

11.3.3.3 Decision Making under Risks

The previous subsections deal with decisions where the probabilities of future situations are uncertain or their probabilities are assumed to be identical. When the future situations or the states of the nature for a given problem are individually predictable, i.e., the probabilities or likelihoods are known, the risk for a decision can be better estimated. In this case, decision making process will be directed based on the weights of probabilities for each payoff.

Definition 11.34 A *decision making under risk* is a selection of an alternative a_i among A that meets a given criterion C, when the likelihood or probability of each possible situation is known or can be predicated.

Decision making under risk can be carried out by two strategies based on the analysis of the maximum expected utility or maximax utility probability.

11.3.3.3.1 Decision Making under Risk with Maximum Expected Utility

The criterion for a decision making under risk can be based on the maximum expected utility of alternatives.

Definition 11.35 An *expected utility EU* is a weighted sum of all utilities u_j for each decision alternative based on known probabilities for each possible situation p_j, i.e.:

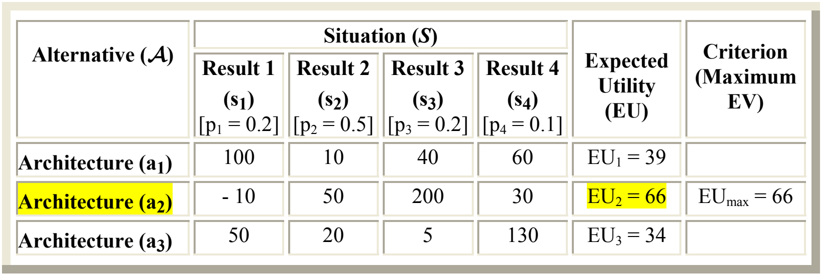
$$EU_i = \sum_{j=1}^{k} u_{ij} \bullet p_j, \;\; 1 \le i \le n \tag{11.23}$$

Definition 11.36 A *decision making under risk with maximum expected utility* d_{maxEU} yields a decision with the *maximum expected utilities* of all alternatives, i.e.:

$$\begin{aligned} d_{maxEU} &= f\text{: } \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (EU_i \mid 1 \le i \le n)\} \end{aligned} \tag{11.24}$$

Example 11.6 Consider the software engineering project given in Example 11.1. A decision under risk with maximum expected utility can be made based on the *EUs* determined by Eq. 11.23 for different decision alternatives as shown in Table 11.9.

Table 11.9
Decision Making based on the Maximum Expected Utility for the Software Engineering Project



Alternative ($\mathcal{A}$)	Situation (S)				Expected Utility (EU)	Criterion (Maximum EV)
	Result 1 (s_1) [$p_1 = 0.2$]	Result 2 (s_2) [$p_2 = 0.5$]	Result 3 (s_3) [$p_3 = 0.2$]	Result 4 (s_4) [$p_4 = 0.1$]		
Architecture (a_1)	100	10	40	60	$EU_1 = 39$	
Architecture (a_2)	- 10	50	200	30	$EU_2 = 66$	$EU_{max} = 66$
Architecture (a_3)	50	20	5	130	$EU_3 = 34$	

After the expected utilities for all three alternatives are obtained as shown in Table 11.9, the best decision with the maximum expected utility can be determined according to Eq. 11.24 as follows:

$$\begin{aligned} d_{maxEU} &= f\colon \mathcal{A} \times C \to \mathcal{A} \\ &= \{a_i \mid max\ (EU_i \mid 1 \leq i \leq n)\} \\ &= \{a_i \mid man\ (EU_1, EU_2, EU_3)\} \\ &= \{a_2 \mid EU_2 = 66\} \end{aligned}$$

The solution indicates that the decision under risk for this given project with the maximum expected utility criterion is Architecture a_2 that will result in a maximum weighted sum EU_2 = $66,000.

Decision making under risk with the maximum expected utility d_{maxEU} can be described by a backward-inducted *decision tree* as shown in Fig. 11.11. The decision tree provides another approach to derive the maximum expected utility in two steps [Friedman, 1996]. First, the individual weighted utilities of all the alternatives are calculated according to Eq. 11.23, which yields EU_i, $1 \leq i \leq 3$, represented by the three middle nodes. Then, the maximum utility EU_{max} is selected from these three middle nodes according to Eq. 11.24, which yields node A represented by decision d_2 with EU_{max} = 66.

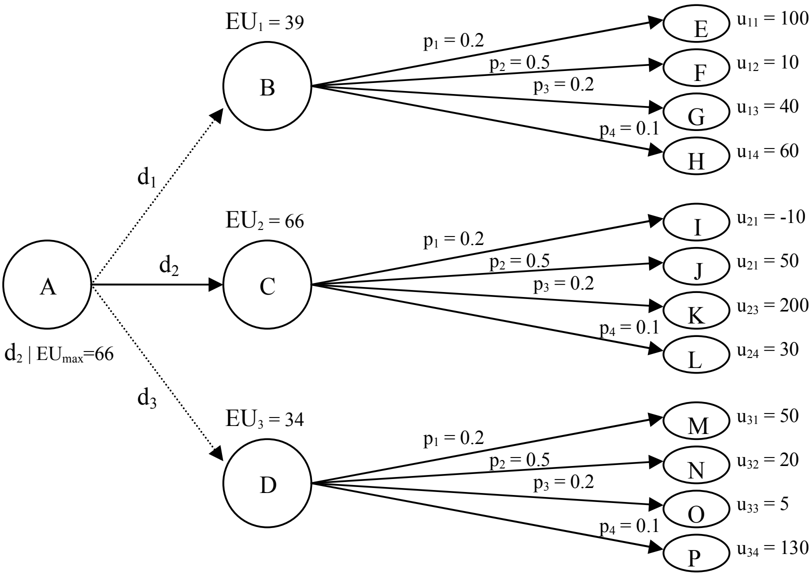


Figure 11.11 A decision tree based on the strategy of maximum expected utility

11.3.3.3.2 Decision Making under Risk with Maximax Utility Probability

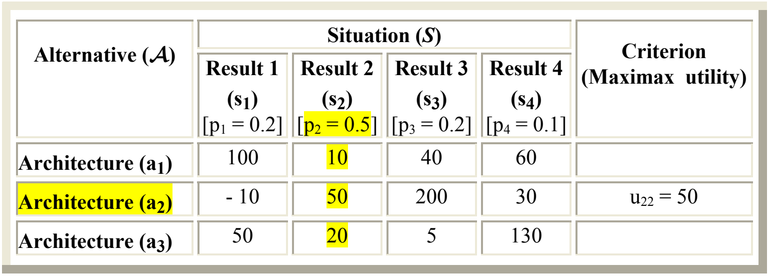
Observing the strategy presented in Section 11.3.3.3.1, it can be seen that the decision based on expected values are dominated by the largest probability of situations or the states of nature. Therefore, a simplified method of decision making under risk with maximum probability can be derived without calculation of the expected values.

Definition 11.37 A *decision making under risk with maximum utility of maximum probability* $d_{maximax\text{-}p}$ yields a decision with the *maximum* utility of the *maximum* probability of outcome of all alternatives, i.e.:

$$\begin{aligned} d_{maximax\text{-}p} &= f\text{: } \mathcal{A} \times C \rightarrow \mathcal{A} \\ &= \{a_i \mid max\ (u_{ij} \mid (max\ (p_j \mid 1 \leq j \leq \text{k})),\ 1 \leq i \leq n\} \end{aligned} \tag{11.25}$$

Example 11.7 Consider the software engineering project given in Example 11.1. A decision under risk with maximum utility of maximum probability can be made based on Eq. 11.25 for different decision alternatives as shown in Table 11.10.

Table 11.10
Decision Making based on the Maximax Utility Probability for the Software Engineering Project



Alternative ($\mathcal{A}$)	Situation (*S*)				Criterion (Maximax utility)
	Result 1 (s_1) [$p_1 = 0.2$]	Result 2 (s_2) [$p_2 = 0.5$]	Result 3 (s_3) [$p_3 = 0.2$]	Result 4 (s_4) [$p_4 = 0.1$]	
Architecture (a_1)	100	10	40	60	
Architecture (a_2)	- 10	50	200	30	$u_{22} = 50$
Architecture (a_3)	50	20	5	130	

According to Eq. 11.25, the expected values for all three alternatives are obtained as shown in Table 11.10. The best decision with the maximum expected values can be determined as follows:

$$\begin{aligned} d_{maximax\text{-}p} &= f\text{: } \mathcal{A} \times C \rightarrow \mathcal{A} \\ &= \{a_i \mid max\ (u_{ij} \mid (max\ (p_j \mid 1 \leq j \leq \text{k})),\ 1 \leq i \leq n\} \\ &= \{a_i \mid max\ (u_{12}, u_{22}, u_{32})\} \\ &= \{a_2 \mid u_{22} = 50\} \end{aligned}$$

The solution indicates that the decision under risk for this given project with the maximax probability based utility is Architecture a_2 that will result in a maximum possible utility u_{22} = \$50,000.

11.3.4 GAME THEORY

In Section 11.3.3 we explored a wide range of the decision making strategies known as the static strategies, because, although the environment of decision makers may change, its changes are independent of the decision makers' activities or expectations. Also, different decision strategies may be selected in the same situation or environment based on the decision makers' values and attitudes towards risk and their predictions on future outcomes.

In classic decision and operations theories [Bronson and Naadimuthu, 1997], although the states of nature or environment may be both deterministic or nondeterministic, its state of nature as an outcome of the environment will not be changed or affected by the decision maker's actions. In other words, there are natural rules but no adaptive competitors in the static decision making processes. However, more decision making situations are dynamic rather than static, where the decision maker is under competition in games.

Definition 11.38 A *game* is a decision process under competition where opponent players or opponent groups of players compete for the maximum gain or a success state in the same environment according to the same predetermined rules of the game.

Games traditionally deal with probability-based static payoff tables. However, this method is found inadequate to deal with the dynamic behaviors of games and to rigorously determine the outcomes of games. This section presents a formal treatment of games by a set of mathematical models on both of the layout and behaviors of games [Wang, 2005e].

11.3.4.1 The Formal Model of Games

The architecture or layout of a game can be formally described by the following mathematical model, where the behaviors of the game are modeled by a series of matches between the players.

Definition 11.39 A *formal game G* is a 4-tuple, i.e.:

$$G = (P, D, M, S) \tag{11.26}$$

where

- P is a finite set of *players* $P = \{p_1, p_2, \ldots, p_n\}$, and n is the number of players, $n \geq 2$.

- D is a finite set of *decisions* for certain *moves*, $D = \{d_1, d_2, \ldots, d_k\}$, $k \geq 1$. All players in G have the same number of alternative decisions.

- M is a finite set of *matches* between player, $M = \{m_1, m_2, \ldots, m_q\}$, $q \geq 1$.

- S is a finite set of cumulated *scores* for each players, $S = \{s_1, s_2, \ldots, s_n\}$.

For a generic game, the matches, which represent the behaviors of the game, can be further described below.

Definition 11.40 A *match* $m \in M$ of a game $G = (P, D, M, S)$ is a function that maps a set of n decisions made by each player into a set of n scores S for each of the players, i.e.:

$$m = f_m : D_1 \times D_2 \times \ldots \times D_n \rightarrow S \tag{11.27}$$

A match corresponds to an individual block preset in a given payoff table of the game. A set of matches in the given game is constrained by a set of generic rules [Wang, 2005e].

Lemma 11.5 In a formal game $G = (P, D, M, S)$, the following generic rules for matches should be obeyed in order to yield stable and predictable game behaviors and scores:

***Rule* (*a*):** All players are supposed to pursue the maximum gains on the basis of the same predefined payoff table.

***Rule* (*b*):** Whenever the first player initiates a move in a specific set of matches, the remaining moves (actions) of all players in the set of matches are determined according to Rule (a).

***Rule* (*c*):** Each match preset in the payoff table may only be used once in the set of matches.

The above rules form the constraints of formal games and make a game to be deterministic and its outcomes of all sets of matches are predictable. Rules (a) and (b) guarantee that all matches of a game are determinable on the basis of the given payoff table. Rule (c) assurances that a set of matches in a game is finite and the number of matches in the set is a constant.

Lemma 11.6 The *number of individual matches* n_m in a set of matches for a given game $G = (P, D, M, S)$ is determined by:

$$n_m = k^{\mathrm{n}} \tag{11.28}$$

where n is the number of players in a game, and k is the number of alternative decisions (moves) defined in the game for each player.

Example 11.8 An $n \times k = 2 \times 2$ game $G_1 = (P, D, M, S)$ can be formally described according to Definition 11.39 as follows:

- *Players* $P = \{a, b\}, n = 2$
- *Decisions* $D = \{d_1, d_2\}, k = 2$
- *Scores* $S = \{s_a, s_b\}$
- *Matches* $M = \{m_{11}, m_{12}, m_{21}, m_{22}\}, n_m = k^n = 2^2 = 4$

where letting a_1 and a_2 be the alternative decisions of player A, and b_1 and b_2 the alternative decisions of player B, the four matches can be determined according to Eq. 11.31 as shown below:

$$
\begin{aligned}
m_{ab} &= D_a : D_b \rightarrow S(s_a : s_b)\\
m_{11} &= a_1 : b_1 \rightarrow 0 : 0\\
m_{12} &= a_1 : b_2 \rightarrow \text{-}1 : 1\\
m_{21} &= a_2 : b_1 \rightarrow \text{-}2 : 2\\
m_{22} &= a_2 : b_2 \rightarrow 3 : \text{-}3
\end{aligned}
$$

The above matches can be represented by a payoff table as shown in Table 11.11.

Table 11.11 The Payoff Table of M = {m_{11}, m_{12}, m_{21}, m_{22}}

	b_1	**b_2**
a_1	0 : 0	-1 : 1
a_2	-2 : 2	3 : -3

This is the static architecture or layout of the game G_1. Its dynamic behaviors on the basis of the layout will be discussed in the following subsections.

11.3.4.2 Properties of Games

This subsection analyzes the common properties of formal games that may be interesting for all players. The properties of formal games can be used to predicate possible outcomes of games and to select optimal strategies or moves in games.

When a game G is set according to Definitions 11.39 and 11.40, the properties of G, such as the number of matches, the number of sets of matches, and the winner, are determined. Game theory may be used to predict and select the optimal combination of individual strategies. However, the score for any individual strategy in G has already been fixed according to the payoff table.

The 38th Principle of Software Engineering

Theorem 11.3 The *properties of games* state that a formal game G is *deterministic* and *conservative*. That is, once the game $G = (P, D, M, S)$ is set, the properties of G are determined and predictable, but not changeable by any player in the game.

According to Theorem 11.3 [Wang, 2005e], game theory may be used to predict and select the optimal combinations of individual strategies for a player in a given game G. However, the optimal strategies may not necessarily result in a win situation rather than a minimal loss in some cases, because the scores for individual moves and their combination strategies in G are determined by the settings of the game.

Corollary 11.2 The outcomes of a formal game $G = (P, D, M, S)$ are constrained by the settings of the game. Although an individual strategy may result in the maximum gain, the final score of a player in a whole set of games is fixed by the payoff table in a particular match, which may not necessarily result in a win situation.

The objective of decision makers in a game is to make the score of a player to the maximum. However, according to Corollary 11.2, max(s_i) may not mean a winning score due to the settings of a given game.

Definition 11.41 A *set of matches* is a series of matches in a game $G = (P, D, M, S)$ in which all players may use each pair of their alternative strategies only once.

Lemma 11.7 The *total sets of matches* n_s in a game $G = (P, D, M, S)$, in which all players may use each pair of their alternative strategies only once determined according to the current move of opponent and the rule of the maximum gains based on the given layout of the game, can be determined by:

$$n_s = n \bullet k \tag{11.29}$$

where k is the number of alternative decisions (moves) defined in the game, and n is the number of players.

Lemma 11.8 The *total number of matches* q of a game $G = (P, D, M, S)$ is determined by the number of sets of matches n_m and number of matches in each set n_s, i.e.:

$$\begin{aligned} q &= n_s \bullet n_m \\ &= nk \bullet k^n \\ &= n \bullet k^{n+1} \end{aligned} \tag{11.30}$$

Lemmas 11.7 through 11.8 can be used to determine the properties and attributes of any given game. The attributes of typical games can be predicated using theses lemmas as shown in Table 11.12.

Table 11.12 Attributes of Typical Games

n	$k \Rightarrow (n_m = k^n \mid n_s = n \bullet k \mid q = n_m\, n_m = nk^{n+1})$											
	1			**2**			**3**			**4**		
2	1	2	2	4	4	16	9	6	54	16	8	128
3	1	3	3	8	6	48	27	9	243	64	12	768
4	1	4	4	16	8	128	81	12	972	256	16	4096
5	1	5	5	32	10	320	243	15	3645	1024	20	20480

It can be seen in Table 11.12 that the complexity of games is explosively increasing. This explains why games are so complicated and difficult to be modeled and formally treated on the basis of conventional game theory [von Neumann and Morgenstern, 1980; Osborne and

Rubinstein, 1994; Bronson and Naadimuthu, 1997]. For example, when the number of players n = 5 and the number of alternative strategies of each player k = 4, the total number of matches of the game may easily exceeded 20,000. However, the formal game theory developed in this section is able to analyze any games no matter how large n and k would be based on the generic mathematical model of abstract games [Wang, 2005e].

Since games with multiple players can be divided into a number of pairwise games, the following sections will focus on binary games as shaded in Table 11.12.

Definition 11.42 A *binary game* $G = (P_2, D, M, S_2)$ is a game with only two players $n = 2$, where $P_2 = \{p_1, p_2\}$ and $S_2 = \{s_1, s_2\}$. It will simply called a game in the remainder of this section.

The attributes of binary games are shown in the first row of Table 11.12. The properties and dynamic behaviors of binary games can be analyzed in the categories of zero-sum and nonzero-sum games.

11.3.4.3 Behaviors of Zero-Sum Games

Definition 11.43 A *zero-sum game* is a game where the total scores of all n players in the game is zero, i.e.:

$$\sum_{i=1}^{n} s_i = 0 \tag{11.31}$$

In the case of a binary game, Eq. 11.31 can be expressed as follows:

$$s_1 = - s_2 \tag{11.32}$$

where Eq. 11.32 models a decision making situation that is known as one player's gain is another's loss.

Lemma 11.9 The *condition for a zero-sum game* is *iff* that each of the n_m individual matches is zero-sum, i.e.:

$$s_q = 0,\ 1 \le q \le n_m \tag{11.33}$$

Example 11.9 The game $G_1 = (P, D, M, S)$ as shown in Example 11.8 and Table 11.11 is a zero-sum game. The properties and behaviors of G_1 can be formally analyzed according to Eqs. 11.31 through 11.33 as follows:

The properties of game $G_1 = (P, D, M, S)$ are:

- *Number of sets of matches:* $n_s = n \bullet k = 2 \bullet 2 = 4$
- *Number of matches in a set:* $n_m = k^n = 2^2 = 4$
- *Total number of matches in the game:*
  $q = n_s \bullet n_m = n \bullet k^{n+1} = 2 \bullet 2^3 = 16$

The behaviors of game $G_1 = (P, D, M, S)$ can be described by the four sets of matches as illustrated in Fig. 11.12.

$$\begin{array}{ll} & s_a : s_b \\ \text{Set 1: } a_1 \xrightarrow{-1:1} b_2 \xrightarrow{3:-3} a_2 \xrightarrow{-2:2} b_1 \xrightarrow{0:0} a_1 & \Rightarrow 0 : 0 \\ \text{Set 2: } a_2 \xrightarrow{-2:2} b_1 \xrightarrow{0:0} a_1 \xrightarrow{-1:1} b_2 \xrightarrow{3:-3} a_2 & \Rightarrow 0 : 0 \\ \text{Set 3: } b_1 \xrightarrow{0:0} a_1 \xrightarrow{-1:1} b_2 \xrightarrow{3:-3} a_2 \xrightarrow{-2:2} b_1 & \Rightarrow 0 : 0 \\ \text{Set 4: } b_2 \xrightarrow{3:-3} a_2 \xrightarrow{-2:2} b_1 \xrightarrow{0:0} a_1 \xrightarrow{-1:1} b_2 & \Rightarrow 0 : 0 \end{array}$$

Figure 11.12 Sets of matches in the zero-sum game G_1

Lemma 11.10 The *final scores of all sets of matches* of formal games G are the same, no matter who moves first and which strategy (decision alternative) is selected for the first move.

Corollary 11.3 The *scores* of a $2 \times k$ formal game G, $s_a : s_b$, is predetermined by the settings of the payoff table, i.e.:

$$\begin{aligned} s_a : s_b &= (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{a}) : (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{b}) \\ &= (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{a}) : (-\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{a}) \end{aligned} \qquad (11.34)$$

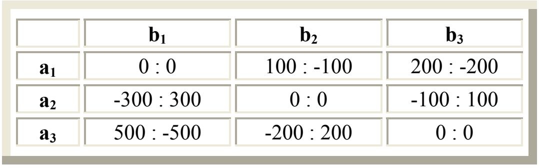
where k is the number of alternative decision strategies and k is the same for all players.

According to Corollary 11.3, the results of all possible sets of matches for a given zero-sum game can be predicated using Eq. 11.34. For instance, the final score of Example 11.9 can be calculated according to Eq. 11.34 as follows:

$$\begin{aligned} s_a : s_b &= (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{a}) : (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{b}) \\ &= (s_{11}^{a} + s_{12}^{a} + s_{21}^{a} + s_{22}^{a}) : \\ &\quad (s_{11}^{b} + s_{12}^{b} + s_{21}^{b} + s_{22}^{b}) \\ &= (0 - 1 - 2 + 3) : \\ &\quad (0 + 1 + 2 - 3) \\ &= 0 : 0 \end{aligned}$$

Example 11.10 For a 2 × 3 game $G_2 = (P, D, M, S)$ with the following payoff table, try to determine its properties and behaviors.

Table 11.13
The Payoff Table of $G_2 = (P, D, M, S)$



	b₁	**b₂**	**b₃**
a₁	0 : 0	100 : -100	200 : -200
a₂	-300 : 300	0 : 0	-100 : 100
a₃	500 : -500	-200 : 200	0 : 0

The properties of $G_2 = (P, D, M, S)$ are:

- *Number of sets of matches:* $n_s = n \bullet k = 2 \bullet 3 = 6$
- *Number of matches in a set:* $n_m = k^n = 3^2 = 9$
- *Total number of matches in the game:* $q = n_s \bullet n_m = n \bullet k^{n+1} = 2 \bullet 3^3 = 54$

The above properties can also be obtained from Table 11.13.

According to Corollary 11.3, the final scores of $G_2 = (P, D, M, S)$ are as follows:

$$\begin{aligned} s_a : s_b &= (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{a}) : (\sum_{i=1}^{k}\sum_{j=1}^{k} s_{ij}^{b}) \\ &= (s_{11}^{a} + s_{12}^{a} + s_{13}^{a} + s_{21}^{a} + s_{22}^{a} + s_{23}^{a} + s_{31}^{a} + s_{32}^{a} + s_{33}^{a}) : \\ &\quad (s_{11}^{b} + s_{12}^{b} + s_{13}^{b} + s_{21}^{b} + s_{22}^{b} + s_{23}^{b} + s_{31}^{b} + s_{32}^{b} + s_{33}^{b}) \\ &= (0 + 100 + 200 - 300 + 0 - 100 + 500 - 200 + 0) : \\ &\quad (0 - 100 - 200 + 300 + 0 + 100 - 500 + 200 + 0) \\ &= 200 : \text{-}200 \end{aligned}$$

The behaviors of $G_2 = (P, D, M, S)$ can be modeled by the 54 detailed matches in the following 6 sets as shown in Fig. 11.13.

$$s_a : s_b$$

$$\begin{aligned}
\text{Set 1: } & a_1 \xrightarrow{0:0} b_1 \xrightarrow{500:-500} a_3 \xrightarrow{-200:200} b_2 \xrightarrow{100:-100} a_1 \\
& \xrightarrow{200:-200} b_3 \xrightarrow{0:0} a_3,\ a_2 \xrightarrow{-300:300} b_1, \\
& a_2 \xrightarrow{0:0} b_2,\ a_2 \xrightarrow{-100:100} b_3 \qquad \Rightarrow 200 : -200 \\
\text{Set 2: } & a_2 \xrightarrow{-300:300} b_1 \xrightarrow{500:-500} a_3 \xrightarrow{-200:200} b_2 \xrightarrow{100:-100} a_1 \\
& \xrightarrow{0:0} b_1, a_1 \xrightarrow{200:-200} b_3,\ a_2 \xrightarrow{0:0} b_2, \\
& a_2 \xrightarrow{-100:100} b_3,\ a_3 \xrightarrow{0:0} b_3 \qquad \Rightarrow 200 : -200 \\
\text{Set 3: } & a_3 \xrightarrow{-200:200} b_2 \xrightarrow{100:-100} a_1 \xrightarrow{0:0} b_1 \xrightarrow{500:-500} a_3 \\
& \xrightarrow{0:0} b_3 \xrightarrow{200:-200} a_1,\ a_2 \xrightarrow{-300:300} b_1, \\
& a_2 \xrightarrow{0:0} b_2,\ a_2 \xrightarrow{-100:100} b_3 \qquad \Rightarrow 200 : -200 \\
\text{Set 4: } & b_1 \xrightarrow{500:-500} a_3 \xrightarrow{-200:200} b_2 \xrightarrow{100:-100} a_1 \xrightarrow{0:0} b_1 \\
& \xrightarrow{-300:300} a_2 \xrightarrow{-100:100} b_3,\ b_2 \xrightarrow{0:0} a_2, \\
& b_3 \xrightarrow{200:-200} a_1,\ b_3 \xrightarrow{0:0} a_3 \qquad \Rightarrow 200 : -200 \\
\text{Set 5: } & b_2 \xrightarrow{100:-100} a_1 \xrightarrow{0:0} b_1 \xrightarrow{500:-500} a_3 \xrightarrow{-200:200} b_2 \\
& \xrightarrow{0:0} a_2 \xrightarrow{-300:300} b_1,\ b_3 \xrightarrow{200:-200} a_1, \\
& b_3 \xrightarrow{-100:100} a_2,\ b_3 \xrightarrow{0:0} a_3 \qquad \Rightarrow 200 : -200 \\
\text{Set 6: } & b_3 \xrightarrow{200:-200} a_1 \xrightarrow{0:0} b_1 \xrightarrow{500:-500} a_3 \xrightarrow{-200:200} b_2 \\
& \xrightarrow{100:-100} a_1, a_3 \xrightarrow{0:0} b_3,\ b_1 \xrightarrow{-300:300} a_2, \\
& b_2 \xrightarrow{0:0} a_2,\ b_3 \xrightarrow{-100:100} a_2 \qquad \Rightarrow 200 : -200
\end{aligned}$$

Figure 11.13 Sets of matches of the 2 × 3 zero-sum game G_2

Corollary 11.4 The scores of a given game G, $s_a : s_b$, can be evaluated as follows:

$$\begin{cases} s_a > s_b\text{: Player A won} \\ s_a = s_b\text{: Tied} \\ s_a < s_b\text{: Player B won} \end{cases} \tag{11.35}$$

Therefore, the final score predicated for Example 11.9, $s_a : s_b = 0 : 0$, shows a tied game; while that of Example 11.10, $s_a : s_b = 200 : -200$, indicates that Player A wins.

11.3.4.4 Behaviors of Nonzero-Sum Games

A more general type of games is nonzero-sum games, where all players involved share a certain pie with a fixed size. From this view, the zero-sum game discussed in previous subsection is a special case of nonzero-sum games where the size of the pie is zero.

Definition 11.44 A *nonzero-sum game* is a game where the total scores of all players in the game is a positive nonzero value, i.e.:

$$\sum_{i=1}^{n} s_i > 0 \tag{11.36}$$

A group on a common project or a set of partners bidding for a contract is typical examples of nonzero-sum games.

The most interesting property of decision making in nonzero-sum games is that there is an ideal state of result known as the win-win situation.

Definition 11.45 A *win-win game* is a game in which all players gain a satisfied score constrained by Eq. 11.36.

Lemma 11.11 A *win-win game* can only exist in nonzero-sum games.

According to Lemma 11.11, if all the competitive players in a nonzero-sum game are coordinated, i.e., a superset of partnership is established in the game, every party may gain a certain benefit.

The 39th Principle of Software Engineering

Theorem 11.4 The *conditions of win-win decisions* state that a win-win decision can be achieved when the following condition of a nonzero-sum game is satisfied:

$$\sigma \geq \frac{1}{n_s} \sum_{i=1}^{n} s_i \tag{11.37}$$

where σ is the sum of the game that is a positive nonzero constant, s_i is the expected score of player *i*, and n_s is the number of sets of matches in the game.

Based on Theorem 11.4 [Wang, 2005e], a win-win game may satisfy all players when the constant sum σ is large enough as described by Eq. 11.37.

Example 11.11 Given a 2 × 2 nonzero-sum game $G_3 = (P, D, M, S)$ with the following payoff table and σ = 100, try to determine its properties and behaviors.

Table 11.14
The Payoff Table of $G_3 = (P, D, M, S)$

	b₁	**b₂**
a₁	70 : 30	20 : 80
a₂	60 : 40	90 : 10

The properties of $G_3 = (P, D, M, S)$ are:

- *Number of sets of matches:* $n_s = n \bullet k = 2 \bullet 2 = 4$
- *Number of matches in a set:* $n_m = k^n = 2^2 = 4$
- *Total number of matches in the game:*
  $q = n_s \bullet n_m = n \bullet k^{n+1} = 2 \bullet 2^3 = 16$

According to Theorem 11.4, the final scores of $G_3 = (P, D, M, S)$ are as follows:

$$
\begin{aligned}
s_a : s_b &= \left(\sum_{i=1}^{k}\sum_{j=1}^{k} s^{a}_{ij}\right) : \left(\sum_{i=1}^{k}\sum_{j=1}^{k} s^{b}_{ij}\right) \\
&= (s^{a}_{11} + s^{a}_{12} + s^{a}_{21} + s^{a}_{22}) : \\
&\quad (s^{b}_{11} + s^{b}_{12} + s^{b}_{21} + s^{b}_{22}) \\
&= (70 + 20 + 60 + 90) : \\
&\quad (30 + 80 + 40 + 10) \\
&= 240 : 160
\end{aligned}
$$

This result indicates that the four sets of matches defined by G_3 will result in the average score in each match as 60 : 40, in which Players A and B share the σ = 100. This can be proved by the following four sets of matches as shown in Fig. 11.14.

$$\begin{array}{lr}
 & s_a : s_b \\
\text{Set 1: } a_1 \xrightarrow{20:80} b_2 \xrightarrow{90:10} a_2 \xrightarrow{60:40} b_1 \xrightarrow{70:30} a_1 & \Rightarrow 240 : 160 \\
\text{Set 2: } a_2 \xrightarrow{60:40} b_1 \xrightarrow{70:30} a_1 \xrightarrow{20:80} b_2 \xrightarrow{90:10} a_2 & \Rightarrow 240 : 160 \\
\text{Set 3: } b_1 \xrightarrow{70:30} a_1 \xrightarrow{20:80} b_2 \xrightarrow{90:10} a_2 \xrightarrow{60:40} b_1 & \Rightarrow 240 : 160 \\
\text{Set 4: } b_2 \xrightarrow{90:10} a_2 \xrightarrow{60:40} b_1 \xrightarrow{70:30} a_1 \xrightarrow{20:80} b_2 & \Rightarrow 240 : 160
\end{array}$$

Figure 11.14 Sets of matches of the 2 × 2 nonzero-sum game G_3

It may be observed that for a given game in a certain context, it would appear to be competitive between conflict interests of players. However, at a higher level of an enlarged scope of the given game, it may be perceived differently as noncompetitive for all parties involved. Based on this systematical view, the following corollary for management attitude and skills can be derived.

Corollary 11.5 The *art of management*, to a certain extent, is to create a win-win environment for all members, partners, and parent organizations involved in a game context.

11.3.5 DECISION GRID THEORY

Traditional decision theories have been focused on static and single decision making techniques. However, a wide range of problems and real-world challenges require a series of dynamic decision makings. There is a lack of coherent theoretical framework for such kind of decision making requirements. Especially, there is a need for a theory that deals with the issues when a mistake or multiple mistakes are made in a decision chain such as: How can mistakes be recovered? What are the consequences or costs of mistakes in decision chains?

This section introduces a new decision and operations theory, known as the decision grids [Wang, 2005b], for modeling and supporting dynamic and sequential decision making. The mathematical models of decision grids are introduced and their properties are rigorously described. The formal treatment of serial decision makings with both limited and unlimited trials are modeled by decision grids. This new theory can be applied in a wide range of serial and dynamic decision making situations in management science, operations studies, cognitive science, sociology, economy, software engineering, systems engineering, political science, statistics, as well as everyday lives.

11.3.5.1 The Formal Model of Decision Grids

Definition 11.46 A *decision grid* is a directed network of a series of decisions over time where each decision possesses only two possible outcomes, right or wrong, where the effort spent to make a right decision is assumed to be identical with that of a wrong decision.

A decision grid is illustrated as shown in Fig. 11.15.

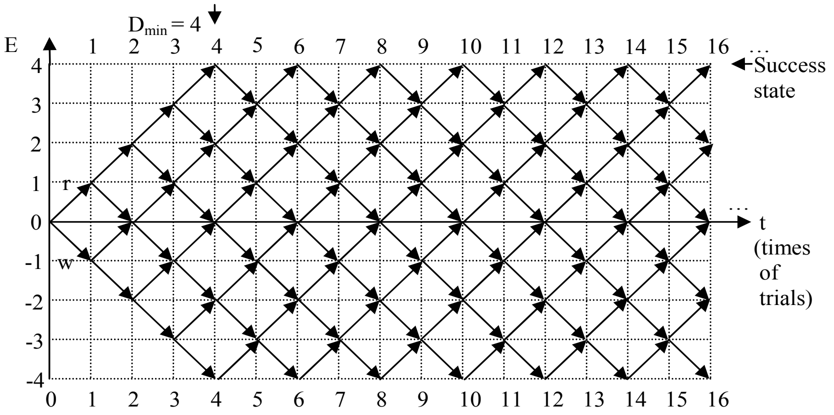


Figure 11.15 A decision grid DG_1 with (D_{min} = 4)

Definition 11.47 The *formal model of a decision grid DG* is a 4-tuple, i.e.:

$$DG = (T, D, E, S) \tag{11.38}$$

where

- T is a finite or infinite set of *trials* $T = \{t_1, t_2, …, t_n\}$, and n is the time points of trials where n may be infinitive.

- D is the *decision distance* of a series of decision trials, $D = t_i - t_0 = t_i$, $1 \leq i \leq n$.

- E is the effort of a specific trial towards the success state in the grid, $0 \leq E \leq n$.

- S is a finite or infinite set of *success states* of the grid, $S = \{s_1, s_2, …, s_k\}$, $1 \leq k \leq n$.

Example 11.12 A decision grid DG_1 with the minimum effort of four right decisions to achieve the success state, as shown in Fig. 11.15, can be defined as follows:

$$DG_1 = (T, D, E, S)$$

where

- $T = \{t_1, t_2, \ldots, t_{16}, \ldots, t_n\}$, $n = \infty$
- $D = t_i = 4$
- $0 \leq E \leq n$
- $S = \{s_1, s_2, \ldots, s_k\}$, $1 \leq k \leq n$

Observing the decision grid $DG_1 = (T, D, E, S)$, the effort E spent in a right or wrong decision is treated as equivalent. That is, a combined effort of a wrong decision followed by a right decision, or vice versa, results in no effect towards the success state, except that time for two trials has been lost.

Decision grids can be classified into categories of unlimited and limited grids according to the scope of times for allowable trials.

Definition 11.48 When the allowable times of trials t in a decision grid are infinitive, the decision grid is called an *unlimited decision grid*; otherwise, it is a *limited decision grid*.

The unlimited decision grid is a suitable model for the series of decisions toward a success state no matter how many trials are needed, such as an experimental process, a research project, or a person's pursuit towards a goal in life. The limited decision grid is a serial decision model for a short period of trials, such as a student towards a degree, an assessment process, or a deadline-specific process. The following subsections discuss the properties and decision processes of unlimited decision grid first. Most of the properties of unlimited decision grids will be found applicable to limited decision grids.

11.3.5.2 Serial Decision Making with Unlimited Trials

Definition 11.49 The *decision distance D_t* in a decision grid is the number of decisions made from the initial state t_0 towards a success state t_i over time by any path in the decision grid, i.e.:

$$\begin{aligned} D_t &= t_i + t_0 \\ &= t_i \end{aligned} \tag{11.39}$$

It is noteworthy that in a decision grid the decision distance D_t from the initial decision point $d(0, 0)$ to another decision $d(t, e)$ at certain trial point t is a constant, which is not dependent on the paths or the combination of any series of right/wrong decisions.

The 40th Principle of Software Engineering

Theorem 11.5 The *properties of decision grid* state that the decision distance D_t in a decision grid is a constant that is determined by the number of decision trials t_i spent in the time series, i.e.:

$$\begin{aligned} D_t &= t_i \\ &= d_r + d_w \end{aligned} \tag{11.40}$$

where d_r and d_w represent numbers of right and wrong decisions, respectively.

Theorem 11.5 reveals an interesting property of decision grids that the decision distance, or number of decisions needed towards a success goal, is always equal to t_i no matter how the right and wrong decisions are interleaved. This can be proven by observing Fig. 11.15.

Example 11.13 In Fig. 11.15 the decision distances of D_8 and D_{16} are:

$$\begin{aligned} D_8 &= d_r + d_w \\ &= 6 + 2 \\ &= 8 \end{aligned}$$

and

$$\begin{aligned} D_{16} &= d_r + d_w \\ &= 10 + 6 \\ &= 16 \end{aligned}$$

Different combinations of d_r and d_w in the above cases will result in multiple paths towards D_8 or D_{16}. However, the decision distance D_t remains constant.

Corollary 11.6 The *shortest decision distance* D_{min} between the initial state $d(0, 0)$ and the success state $d(t_{min}, D_{min})$ in a decision grid is a series of pure successful decisions where no wrong decision has been made, i.e.:

$$D_{min} = t_{min} = d_r \tag{11.41}$$

Example 11.14 For the given decision grid as shown in Fig. 11.15, the shortest decision distance D_{min} is:

$$\begin{aligned} D_{min} &= t_{min} \\ &= d_r = 4 \end{aligned}$$

Corollary 11.7 The *last decision* d_n of a successful series of decisions S_r is always a right decision, i.e.:

$$\forall S_r = (d_0, d_1, ..., d_n) \Rightarrow d_n \equiv \text{right} \qquad (11.42)$$

Observing that in the decision grid as shown in Fig. 11.15, any number of continuous or separated wrong decisions d_w should be recovered by the same number of right decisions d'_r, before the success state is achieved, i.e.:

$$d'_r \equiv d_w \qquad (11.43)$$

Therefore, replacing d_r with $D_{rmin} + d'_r$ in Eq. 11.40, the following relation can be obtained:

$$\begin{aligned} D_t &= d_r + d_w \\ &= (D_{\min} + d'_r) + d_w \qquad (11.44) \\ &= D_{\min} + 2d_w \end{aligned}$$

Solving Eq. 11.44, the allowable number of wrong decisions for a given decision grid is obtained as follows:

$$d_w = \frac{D_t - D_{\min}}{2} \qquad (11.45)$$

Corollary 11.8 The *maximum number of allowable wrong decisions* $d_{w\text{-}max}$ in a decision series that may achieve the success state in a given decision grid is determined by the times of trials t_i (or D_t) and the minimum decision distance D_{min}, i.e.:

$$d_{w\text{-}max} = \left\lceil \frac{t_i \text{ - } D_{min}}{2} \right\rceil \qquad (11.46)$$

Corollary 11.9 The *minimum number of required right decisions $d_{r\text{-}min}$* in a decision series that may achieve the success state in a given decision grid is a complement number of $d_{w\text{-}max}$ to the given times of trials t_i, i.e.:

$$\begin{aligned} d_{r\text{-}min} &= t_i - d_{w\text{-}max} \\ &= t_i - \left\lceil \frac{t_i - D_{min}}{2} \right\rceil \\ &= \left\lfloor \frac{t_i + D_{min}}{2} \right\rfloor \end{aligned} \tag{11.47}$$

Definition 11.50 The *allowable rate of wrong decisions r_w* in a decision series that eventually achieves the success state in a decision grid is a ratio between the number of wrong decisions and the total times of trials t_i, i.e.:

$$r_w = \frac{d_w}{t_i} \bullet 100\% \tag{11.48}$$

Definition 11.51 The *relative cost of wrong decisions C_w* is the relative difference between the total number of decision trials t_i and the minimum decision distance D_{min}, i.e.:

$$C_w = \frac{t_i - D_{min}}{t_i} \tag{11.49}$$

where C_w is usually represented in a percentage form.

Definition 11.52 The *efficiency of decisions e_r* is the ratio between the minimum decision distance D_{min} and the total number of decision trials spent t_i, i.e.:

$$\begin{aligned} e_r &= \frac{D_{min}}{t_i} \\ &= 1 - C_w \end{aligned} \tag{11.50}$$

where e_r is usually represented in a percentage form.

The properties of the above five attributes of decision grids, in terms of the maximum number of allowable wrong decisions (Eq. 11.46), the minimum number of required right decisions (Eq. 11.47), the allowable rate of wrong decisions (Eq. 11.48), the relative cost of wrong decisions (Eq. 11.49), and the efficiency of decisions (Eq. 11.50), can be illustrated in Fig. 11.16. Detailed data for generating the curves of Fig. 11.16 are given in Table 11.15. It can be seen that, when t_i = 100, for given D_{min} = 4, the efficiency of decisions e_r may drop to zero percent, and on other hand, the relative cost due to multiple wrong decisions C_w may increase to nearly 100 percent.

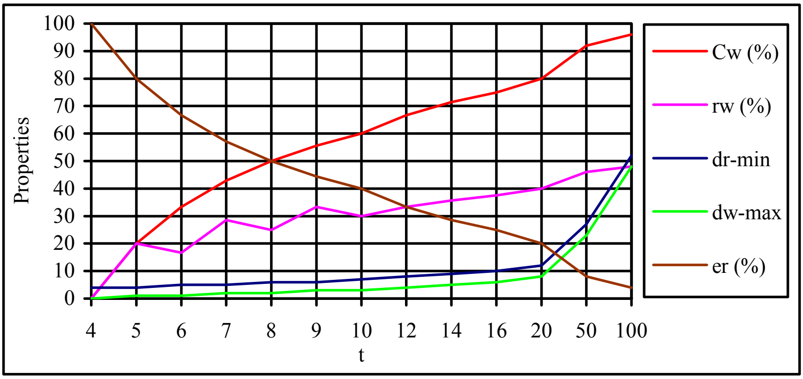


Figure 11.16 Properties of decision grid (D_{min} = 4)

Table 11.15
Properties of Decision Grid (D_{min} = 4)

T	4	5	6	7	8	9	10	12	14	16	20	50	100
C_w (%)	0	20	33.3	42.9	50	55.6	60	66.7	71.4	75	80	92	96
r_w (%)	0	20	16.7	28.6	25	33.3	30	33.3	35.7	37.5	40	46	48
d_r	4	4	5	5	6	6	7	8	9	10	12	27	52
d_w	0	1	1	2	2	3	3	4	5	6	8	23	48
e_r (%)	100	80	66.7	57.1	50	44.4	40	33.3	28.6	25	20	8	4

Based on Fig. 11.16 and Table 11.15, the following corollary can be derived.

Corollary 11.10 The later the wrong decision is corrected, the higher the cost of the decision series; The earlier the wrong decision is corrected, the more efficient of a decision series.

The essence of Corollary 11.10 is a formal description of the empirical wisdoms in everyday life, such as Shakespeare's "all is well that ends well," and the antonym's "a good kick off is worth a half of the success."

It is noteworthy that Fig. 11.16 shows that the allowable rate of wrong decisions r_w towards the success state will not exceed 50% no matter how large t_i is. This observation leads to the following theorem [Wang, 2005b].

The 41st Principle of Software Engineering

Theorem 11.6 The *random series of unlimited trials* states that random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under unlimited trials.

According to Theorem 11.6 and Fig. 11.16, it is impossible to win a game by random decisions, because the probability of wrong or right decisions in random, when t_i is large enough, is exactly 50%. However, the allowable rate of wrong decisions r_w(%) for any series of decisions is always below 50%.

It is intuitive that both time and effort have been lost when any wrong decision is made in the decision grid. The following corollary provides a quantitative explanation of how worse this could be when multiple wrong decisions have been made in a decision process.

Corollary 11.11 A wrong decision results in both losses of time t_l and effort E_l, which can be estimated by the relative differences between the number of decision trials and the minimum decision distance D_{min} (or t_{min}), i.e.:

$$(a)\ t_l = t_i - t_{min} \tag{11.51}$$

$$(b)\ E_l = \frac{D - D_{min}}{D_{min}} = \frac{t_i - t_{min}}{t_{min}} \tag{11.52}$$

The result of Corollary 11.11 can be illustrated as shown in Fig. 11.17. The curves show that, when the number of wrong decisions increases, although there is still changes to reach the success state in a decision grid if $d_w \leq d_{w\text{-}max}$, the loss of time is close to 100%, and the loss of effort is hundreds even thousands times higher than that of the best decision series where no wrong decision had been introduced.

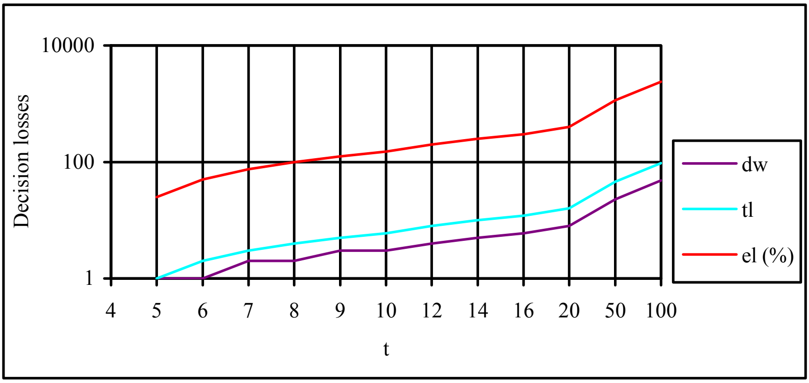


Figure 11.17 Time and effort losses of wrong decisions in decision grid (D_{min} = 4)

Example 11.15 A new software engineering project is expected to be completed by three processes in the best case that involves three continuous right decisions at the beginning of each process. Draw a decision grid for this project and analyze the seven decision attributes on the following cases:

Case 1: The maximum times of trials should be no more than 3 times.

Case 2: The maximum times of trials should be no more than 9 times.

Case 3: If 5 wrong decisions have been made in this project, what is the earliest completion time in terms of number of trials?

A decision grid $DG_2 = (T, D, E, S)$ for the given software engineering project is described in Fig.11.18. It is noteworthy that according to Theorem 11.5, the interesting property of the decision grid is that the decision distance D from the initial decision point $d(0, 0)$ to another decision $d(t_i, e)$ at any given trial t_i is a constant, which is not dependent on the path or the combinations of d_r and d_w.

Solutions for Case 1 through Case 3 can be derived as shown respectively in Table 11.16, where an appropriate equation for each decision attribute of the decision grid is referred. Note that data provided in the square brackets are given from the problem.

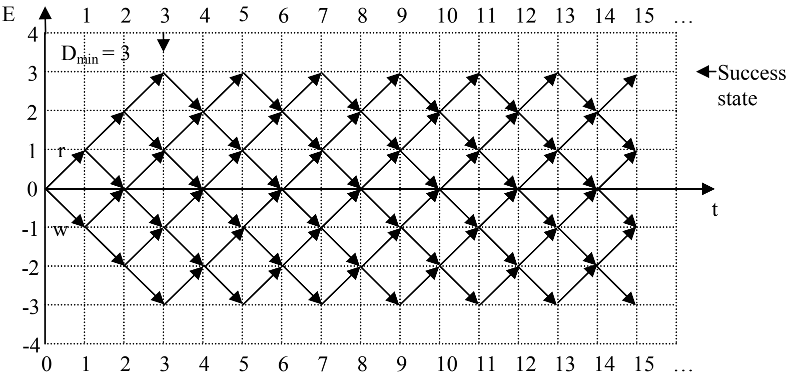


Figure 11.18 A decision grid DG_2 with (D_{min} = 3)

11.3.5.3 Serial Decision Making with Limited Trials

Serial decision making under limited trials can be modeled by a limited decision grid, where the maximum number of decision trials t_i is a finite constant. The limited decision grid is suitable for dealing with serial decisions of a time constrained process. The seven attributes and related theorems and corollaries for unlimited decision grids developed in Section 11.3.5.2 are also applicable to limited decision grids.

Table 11.16
Properties of the Decision Grid DG_2

Attributes of DG_2		Case 1	Case 2	Case 3	Remark
Attribute	**Symbol**				
Minimum distance or minimum times of trials	$D_{min} = t_{min}$	[3]	[3]	[3]	Given
Maximum trials (distance)	D_t	[3]	[9]	13	Eq.11.44
Max. no. of allowable wrong decisions	$d_{w\text{-}max}$	0	3	[5]	Eq.11.46
Min. no. of required right decisions	$d_{r\text{-}min}$	3	6	8	Eq.11.47
Allowable rate of wrong decisions	r_w (%)	0	33.3	38.5	Eq.11.48
Relative cost of wrong decisions	C_w (%)	0	66.7	76.9	Eq.11.49
Efficiency of decisions	e_r (%)	100	33.3	23.1	Eq.11.50
Time loss in decisions	t_l (trials)	0	6	10	Eq.11.51
Effort loss in decisions	E_l (%)	0	200	330	Eq.11.52

Note: Data in square brackets [x] are given.

For decision making under unlimited trials modeled by an unlimited decision grid, there is always a next chance to achieve the success state no matter how many wrong decisions have been made and what are their costs in terms of time and effort. However, decision making with limited trials modeled by a limited decision grid is constrained by the maximum number of trials one may try, or the maximum wrong decisions one may make.

According to Corollary 11.7, the maximum number of allowable wrong decisions is determined by a certain t_i. Therefore, for a given limited decision grid, once the number of wrong decisions made d_w is larger than the maximum allowable mistakes, the decision series should be considered over, because there is no more chance to reach the success state.

Corollary 11.12 For a given limited decision grid, the following condition determines a failure of a decision series under limited trials t_i, i.e.:

$$\begin{aligned} d_w &> d_{w\text{-}max} \\ &= \left\lceil \frac{t_i \text{ - } D_{min}}{2} \right\rceil \end{aligned} \qquad (11.53)$$

Example 11.15 can also be used to analyze a series of decision making cases under limited trials with a limited decision grid.

The 42nd Principle of Software Engineering

Theorem 11.7 The *random series of limited trails* states that random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under limited trials.

It can be seen that both Theorems 11.6 and 11.7 rule out the success possibility of random decision series in either unlimited or limited trials. In a limited decision series, the relative cost and efficiency of the decision series are much worse.

This section shows that decision grids are a powerful means and methodology for dealing with dynamic and serial decision makings with limited or unlimited trials. Decision grids can be applied in a wide range of serial and dynamic decision making situations in management science, operation studies, sociology, economy, software engineering, systems engineering, as well as everyday lives.

11.4 Quality Systems

As discussed in Section 8.2.4, quality is one of the fundamental objectives of engineering. In modern industrial organization, quality has become a major concern of management in virtually all sectors of industries, services, government, health care, and education.

This section reviews classic thought on quality in management science and explains new perceptions and formal treatment of quality and quality systems. Then, it discusses what constitutes a quality control system, and what are the necessary conditions and basic quality assurance techniques to implement it. Typical quality management systems will be described, which covers total quality management, the ISO 9000 quality system, and the ISO 9126 quality attributes.

11.4.1 QUALITY PRINCIPLES

Studies on quality and quality control principles may be traced back to Shewhart in 1939, when the method for statistical quality control was proposed [Shewhart, 1939]. However, quality as one of the essences of management science was perceived differently. Philip Crosby focused on quality that conforms to requirements [Crosby, 1977]. Edwards Deming said quality is how well something meets customers' needs [Deming, 1992/86]. Joseph Juran perceived quality as fit for use [Juran, 1988/89; Huran et al., 1962/80]. Genichi Taguchi viewed quality as the closeness to an ideal state that implements maximum well-being to the society and users [Taguchi, 1986].

Definition 11.53 *Quality Q* is the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs.

Although quality as a term is vague, nobody doubts its importance. Quality before quantity has been a basic principle in management science.

11.4.1.1 Attributes of Quality

Garvin identified eight attributes known as dimensions of quality that break down the abstract concept of quality into detailed characteristics [Garvin, 1987]. The eight dimensions, according to Garvin, are as follows:

- *Performance*: Primitive operating characteristics of a product, service, or system.

- *Features*: The secondary or extra characteristics.
- *Reliability*: Consistency of performance within a given period.
- *Conformance*: The degree of performance and features meets specific requirements and standards.
- *Durability*: The lifespan of a product or service.
- *Serviceability*: Post sale/delivery technical support and maintenance.
- *Aesthetics*: The appearance or inner attribute of beauty in design, production, or service.
- *Perception of excellence*: Implied quality carried by reputation or brand name.

It is interesting to compare the attributes of quality identified in management science and software engineering as shown in Table 11.17.

Table 11.17
Comparison of Quality Concept in Management Science and Software Engineering

No	**Quality attributes**		**Detailed attributes**
	Management science	**Software engineering**	
1	Performance and features	Functionality	Suitability, accuracy, interoperability, and security
2	Reliability	Reliability	Maturity, fault tolerance, and recoverability
3	Conformance	Conformance to requirements (not included in ISO 9126)	Functionality against standards
4	Durability		Validated period of functionality
5		Usability	Understandability, learnability, and operability
6		Efficiency	Time behavior, and resource behavior
7	Serviceability	Maintainability	Analyzability, changeability, Stability, and testability
8		Portability	Adaptability, installability, conformance, and replaceability
9	Aesthetics		Appearance or inner attributes of appreciation
10	Perception on excellence		User satisfaction

The quality attributes of software engineering shown in Table 11.17 are adopted from ISO 9126 (1991), which will be extensively described in Section 11.4.3. Both disciplines cover the quality attributes such as *functionality, reliability, conformance,* and *maintainability*. It indicates that management science has strongly influenced the perception on quality in software engineering. However, software engineering overlooked the subjective attributes of aesthetics and perception on excellence identified in management science. These subjective attributes may be as equally important as those of the other attributes, because software engineering is dealing with complicated creative artifacts and customers always have the final say on quality of a software system. There are three special features in software engineering known as *usability, efficiency,* and *portability.* These features indicate the unique concerns on quality in software engineering.

Because customers are decisive in evaluating the quality of a system or service, surveys on customer feedback are important techniques and a necessary process in any quality control system.

11.4.1.2 Formal Models of Quality

Quality is a generic measure of the degree of excellence of a product or service against a given standard. More specifically, quality is a common attribute of any product or service that characterizes the quantity of both utility and durability of the product or service. Therefore, the general view on quality can be defined as follows [Wang, 2001b].

Definition 11.54 *Quality Q* is a generic and collective attribute of a product, a service, or a system that is proportional to both its average utility *U* and the available duration *T* of the utility, i.e.:

$$Q = U \bullet T \quad \text{[Fh]} \tag{11.54}$$

where the unit of utility is *function* (F), and the unit of duration is *hour* (h), and these result in the unit of quality as *Function-hour* or shortly Fh.

According to Definition 11.54, for a given product, service, or system, there is no quality if there is a lack of either utility ($U = 0$) or availability of the utility ($T = 0$).

Quality defined in Definition 11.54 is the average quality. A more generic form of quality that represents the dynamic aspect of quality when the utility is a function of time is given below.

Definition 11.55 A generic *dynamic utility function U*(*t*) is an inversed exponential function over time, i.e.:

$$U(t) = U(1 - e^{t-T}) \quad [\text{F}] \qquad (11.55)$$

where both *U* and *T* are a positive constant.

With the above definition of *U*(*t*) in the curve as that is shown in Fig. 10.15, the value of the dynamic quality can be determined by the following lemma.

Lemma 11.12 The *integrated quality* with dynamic utility, *Q*(*t*), is an integral of the utility function *U*(*t*) over the entire lifecycle of the utility [0, *T*], i.e.:

$$\begin{aligned} Q(t) &= \int_0^T U(t)dt \\ &= \int_0^T U(1 - e^{t-T})dt \\ &= U(e^{-T} + T - 1) \\ &= UT \text{ - } U(1 \text{ - } e^{-T}) \\ &= Q \text{ - } U(1 \text{ - } e^{-T}) \quad [\text{Fh}] \end{aligned} \qquad (11.56)$$

where *U* is the initial quality of the product, service, or system.

Lemma 11.12 shows that the integrated quality of a dissimilating utility system or product is always smaller than that of constant utility.

The quality formulae defined by Eqs. 11.54 and 11.56 are an absolute value. In practice, a relative measure of quality may provide more information when a standard or benchmark on the quality of a given system or product is available.

Definition 11.56 The *relative quality* *q*(*t*) is a relative degree of difference between the quality of a product, a service, or a system and the standard or benchmark *S* for the expected quality, i.e.:

$$\begin{aligned} q(t) &= \frac{[\int_0^T U(t)dt)] \text{ - } S}{S} \\ &= \frac{Q \text{ - } S \text{ - } U(1 \text{ - } e^{-T})}{S} \quad [\text{Fh}] \end{aligned} \qquad (11.57)$$

When *U*(*t*) = *U* is a positive constant, a special case of the relative quality is as follows:

$$\begin{aligned} q(t) &= \frac{Q \text{ - } S}{S} \\ &= \frac{(U \bullet T) \text{ - } S}{S} \quad [\text{Fh}] \end{aligned} \qquad (11.58)$$

The utility of a product or a system described in Eq. 11.55 can be classified as *external* and *internal* utility. The external utility encompasses the quality attributes of the product or system when it is treated as a black box, such as *functionality, usability, availability, reliability, efficiency, portability, and maintainability*. The internal utility encompasses the quality attributes of the product or system when it is treated as a white box, such as *completeness, correctness, consistency, clearness* (no ambiguity), *feasibility* (can be implemented in technical and economical terms), and *verifiability* (can be measured).

Corollary 11.13 *Quality* is a collective attribute of a product, service, or system.

Corollary 11.13 indicates that the control, assurance, and improvement of quality must be carried out systematically on multiple attributes of the target product, service, or system.

Corollary 11.14 *Quality* is implemented incrementally via each individual in every process.

Quality is closely related to the cost of a product or service in two ways. One is the perception on benefit of quality as described below.

Definition 11.57 The *benefit of a product or a system B* is the quality gained per unit cost (C) in terms of resources, labor, and time, i.e.:

$$\begin{aligned} B &= \frac{Q}{C} \\ &= \frac{U \bullet T}{C} \quad [\text{Fh/\$}] \end{aligned} \qquad (11.59)$$

The other is the principle of the quality-cost relationship known as the *quality funnel principle* [Bain, 1962].

Corollary 11.15 The *quality funnel principle* states that the nearer to the start of the production process, the lower the cost of quality.

If the whole process of manufacture or service can be divided into the processes of *design, implementation,* and *application*, the costs of quality can be classified into three corresponding phases known as the *prevention* costs, the *appraisal* costs, and the *failure* costs.

Because software engineering is a specific branch of the engineering discipline, it obeys the generic engineering rules as stated in Corollaries 11.13 through 11.15. Among a wide variety of goals identified in Theorem 8.2 on conservation of basic engineering constraints, productivity, efficiency, and quality are recognized as the most fundamental categories of goals in software engineering.

11.4.2 QUALITY CONTROL AND ASSURANCE

According to Corollary 11.13, quality is a collective attribute of a product, service, or system. Therefore, the control and management of quality must focus not only on individual attributes but also on the integration of an entire system. Therefore, quality control and assurance are not only individual techniques but also a fundamental infrastructure system for an organization.

11.4.2.1 Quality Control Systems

A generic quality control system encompasses five subsystems known as quality definition, implementation, appraisal, postmortem, and prevention. The five subsystems of the quality control system are illustrated in Fig. 11.19. Fig. 11.19 also describes the sequence of implementation and cyclic operation of these five subsystems.

Definition 11.58 The *quality definition* subsystem is responsible to identify, partition, and quantify the attributes and characteristics of the products or services produced or provided in an organization.

For the establishment of a new quality control system, the quality definition phase is the most crucial one. If this phase can not be achieved, no quality control system may be implemented.

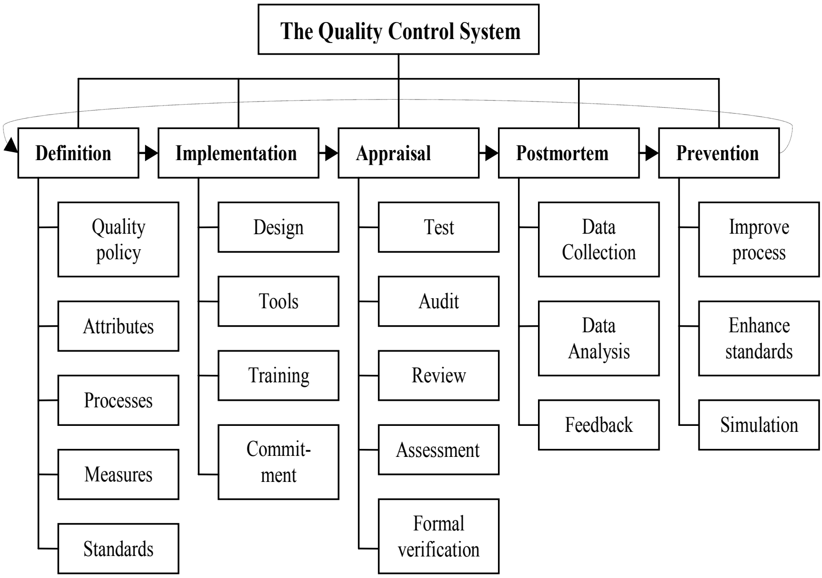


Figure 11.19 The configuration of a generic quality control system

Definition 11.59 The *quality implementation* subsystem distributes quality attributes identified in phase one into individual processes and job functions.

Through training and tools support, each individual in a given process should understand both job functions and quality responsibilities with corresponding expected standards.

Definition 11.60 The *quality appraisal* subsystem is a set of evaluation techniques against the quality standards for each process and each attribute of a given product, service, or system.

Definition 11.61 The *quality postmortem* subsystem is a feedback subsystem that helps to identify existing or potential problems in the process or quality standards on the basis of operating data on current performance.

Definition 11.62 The *problem prevention* subsystem is an adaptive process that prevents recurrent problems or failures from happening through improvement of the current processes and quality standards.

Simulations and *pilot trials* are typical techniques in the phase of problem prevention.

The 43rd Principle of Software Engineering

Theorem 11.8 The *conditions of quality control* state that the *necessary conditions* for implementing a quality control system for a given product, service, or system are that all attributes of its quality can be:

a) Abstractly identified

b) Quantitatively defined, and

c) Independently measurable.

11.4.2.2 Quality Assurance Techniques

Typical quality assurance techniques in quality control systems can be classified into definitive, implemental, appraisable, postmortem, and preventive corresponding to the quality control system model as shown in Fig. 11.19. Effective techniques for quality assurance are quality audit, quality review, quality measurement, and formal verification/validation.

Definition 11.63 *Audit* is an empirical quality assurance technique that uses professional auditors to monitor the quality of products and services on the basis of statistical quality control and review techniques.

Quality audit may be carried out in forms of sampling-based statistical analysis and review. The former is suitable for physical products and mass production systems; the latter is widely used for information-based artifacts and services.

Quality review is a special audit technique for information-based work products such as system designs, plans, software, and documents. Quality review is an effective technique for quality assurance in software engineering.

Definition 11.64 *Measurement* is a quantitative quality assurance technique that evaluates the conformance of products and services against predefined standards or benchmarks.

A major problem in software quality assurance is that there lacks a comprehensive and coherent set of quantitative measures and benchmarked standards for each of the basic attributes of software and software engineering processes. Taking the automobile industry as an example, each and all of the attributes of a car, at a number of thousands level, have been systematically identified and quantitatively defined. Based on this well-defined quality system, each car out of a production line can be guaranteed for the same quality as the standard specified. However, the lack of the measurability in software engineering is the fundamental barrier that prevents the software industry from achieving a predictable and stable quality in software engineering as its counterparts in other industries.

Definition 11.65 *Verification and validation* is a formal quality assurance technique that applies mathematical models, logical inference tools, and simulation systems into quality control of products and services.

It is noteworthy that all practical quality assurance techniques, no matter formal or empirical and quantitative or qualitative, require the establishment of a quality system, which satisfies the conditions as identified in Theorem 11.8 for implementing a quality control system.

Corollary 11.16 A *quality control system* should be designed and implemented as a whole, because any individual quality assurance technique, no matter how effective, can not solve the problem alone in a given quality system.

Corollary 11.16 forms the foundation of total quality management, which will be discussed later in this chapter.

With the influence of W.E. Deming in the 1940s, the concept of quality circles originated in Japan in 1962 on the basis of the statistical quality control theory.

Definition 11.66 The *quality circle* is a four-phase repetitive process for quality improvement that encompasses the phases of problem identification, finding solutions, implementation of the solutions, and evaluation.

In the 1980s, the quality circles are evolved to be the cycle of *plan, do, check,* and *act* as shown in Fig. 11.21, which is known as the *Deming circle*.

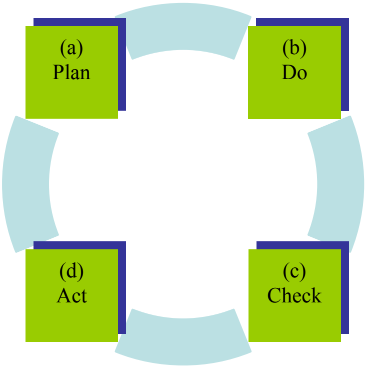


Figure 11.20 The Deming cycle: plan-do-check-action

11.4.3 QUALITY MANAGEMENT SYSTEMS

In the preceding sections it has been observed that quality control and assurance are a management activity at the system level. This section introduces three well-found quality management systems: the total quality management, the ISO 9000 quality system, and the ISO 9126 quality system.

11.4.3.1 Total Quality Management (TQM)

Total quality management has been a main stream methodology in quality management since the 1970s [Deming, 1982; Buckland et al., 1991; EFQM, 1993; Dunn and Ullman, 1994]. Total quality management extends the concept of quality control from the product to the process, from the physical objects to human beings who produce them, from individuals to the entire organization, and from manufacturing to culture in which quality is treated as an integral part of every job function in the organization.

David Garvin summarized three principles of quality known as: a) To set objectives on continuous improvement in quality; b) To focus on the processes that produce products and services; and c) To implement employee's involvement in quality improvement [Garvin, 1991]. These principles cover the essences of total quality management.

Definition 11.67 *Total Quality Management* (TQM) is a systematical management methodology that states quality of products and services of an organization depends on a systematical management of the organization's culture, attitude, and operations through all members' involvement.

TQM adopts all proven quality control techniques such as statistical *quality control, process system, continuous improvement, all employees' involvement,* and *customer feedback*. TQM encompasses a set of key elements such as ethics, integrity, trust, training, teamwork, leadership, recognition, and communication. Among the eight elements, ethics, integrity, and trust are the foundation of the TQM framework. *Training* enables every member of the organization to be integrated into the TQM system. Teamwork and leadership are basic organizational techniques at project or department level. *Recognition* is the means of motivation for everybody involved. Finally, communication is the essential foundation for system synchronization in order to decrease the overhead of interpersonal coordination.

11.4.3.2 The ISO 9000 Quality System

The ISO 9000 quality system has been discussed in Section 8.6.5. A hierarchical structure of the ISO 9001 framework is shown in Table 11.18.

What is software quality and how to measure it? This is a fundamental issue in software engineering for which the formal models of quality as developed in Section 11.4.1 may provide an answer. Usually, *quality software* is perceived as the software that meets users' needs. However, for the same application system, users' needs may be different and informally described. Therefore, the quality of software is difficult to be verified according to this definition. Another definition perceives quality software as the software that contains fewer bugs. However, bugs as an internal feature of software are difficult to identify and measure in practice.

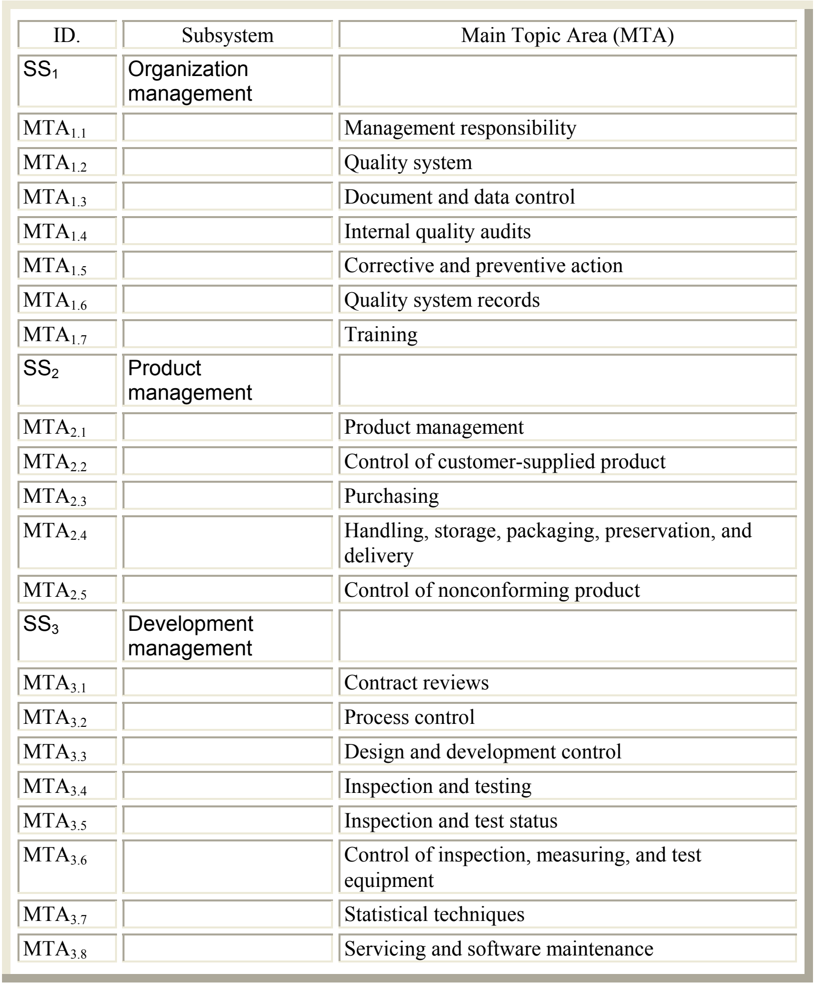
According to the formal model of quality as described in Section 11.4.1.2, software quality can be perceived as follows.

Definition 11.68 *Software quality* is a set of inherent internal and external characteristics of a software system that show relative advantages over similar systems or indicate a conformance to a standard.

The central idea of this definition is to recognize that the quality of software (not quality software) is a relative concept that can be referred to as 'higher' or 'better'.

The design philosophy behind ISO 9001 is a generic quality system perception on software engineering. Although this philosophy has been proven successful in the conventional manufacturing industries, there is still a need for supporting evidence of its effectiveness and impact on the design-intensive software engineering and nonconventional software industries. It appears likely that software engineering is sufficiently unique as an engineering discipline in that it relies upon special foundations and applies a different philosophy as discussed in Chapter 3. Therefore, further studies on common features and differences between conventional mass manufacturing and software engineering are still expected.

Table 11.18
Structure of the ISO 9001 Framework

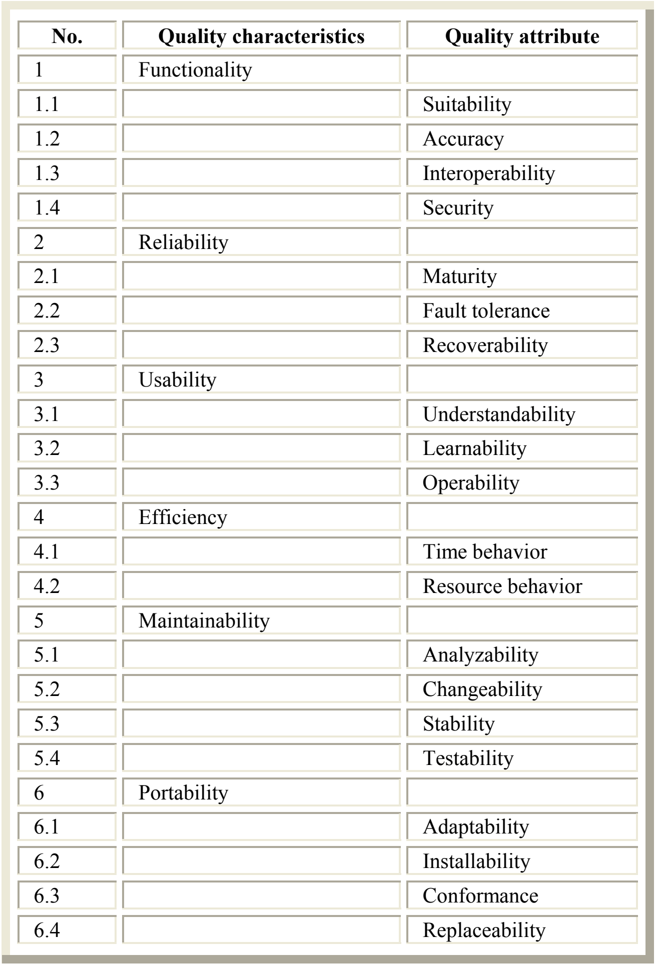


ID.	Subsystem	Main Topic Area (MTA)
SS_1	Organization management	
$MTA_{1.1}$		Management responsibility
$MTA_{1.2}$		Quality system
$MTA_{1.3}$		Document and data control
$MTA_{1.4}$		Internal quality audits
$MTA_{1.5}$		Corrective and preventive action
$MTA_{1.6}$		Quality system records
$MTA_{1.7}$		Training
SS_2	Product management	
$MTA_{2.1}$		Product management
$MTA_{2.2}$		Control of customer-supplied product
$MTA_{2.3}$		Purchasing
$MTA_{2.4}$		Handling, storage, packaging, preservation, and delivery
$MTA_{2.5}$		Control of nonconforming product
SS_3	Development management	
$MTA_{3.1}$		Contract reviews
$MTA_{3.2}$		Process control
$MTA_{3.3}$		Design and development control
$MTA_{3.4}$		Inspection and testing
$MTA_{3.5}$		Inspection and test status
$MTA_{3.6}$		Control of inspection, measuring, and test equipment
$MTA_{3.7}$		Statistical techniques
$MTA_{3.8}$		Servicing and software maintenance

11.4.3.3 The ISO 9126 Quality System

ISO 9126 extends principles of quality control to software engineering and summarizes the major characteristics and attributes of software quality [ISO 9126, 1991] as shown in Table 11.19. ISO 9126 develops a collective way to perceive software quality. According to the philosophy of ISO 9126, software quality is a set of qualitative characteristics and attributes. ISO 9126 provides a software quality model by defining 6 software quality characteristics and 20 attributes, which are intended to be exhaustive.

Table 11.19
ISO 9126 Software Quality Model



No.	Quality characteristics	Quality attribute
1	Functionality	
1.1		Suitability
1.2		Accuracy
1.3		Interoperability
1.4		Security
2	Reliability	
2.1		Maturity
2.2		Fault tolerance
2.3		Recoverability
3	Usability	
3.1		Understandability
3.2		Learnability
3.3		Operability
4	Efficiency	
4.1		Time behavior
4.2		Resource behavior
5	Maintainability	
5.1		Analyzability
5.2		Changeability
5.3		Stability
5.4		Testability
6	Portability	
6.1		Adaptability
6.2		Installability
6.3		Conformance
6.4		Replaceability

The major quality characteristics identified in ISO 9126 are described below:

- *Functionality*: The characteristics that a system can provide specified services that meet users' requirements.

- *Reliability*: The probability that a system will fulfill the service during a given period when ever a user demands.
- *Usability*: The characteristics that a system is ready and easy for use when a user needs its service.
- *Efficiency*: The characteristics that a system uses minimum resources and provides timely response to an application.
- *Maintainability*: The probability that a system can be restored, within a given time after a failure, to provide the originally specified services.
- *Portability*: The characteristics that a system is capable of running on different target machines, operating systems, and network platforms.

Software quality may be characterized by external and internal attributes. External quality characteristics are those that can be evaluated when executing the software; and internal quality characteristics are those that are evaluated by inspecting the internal features of the software. The former is user-oriented and may be verified by black-box testing techniques. However, the latter is oriented to developers and maintainers, and may be verified by white-box testing techniques.

ISO 9126 focuses only on external characteristics of software quality. Substantial internal attributes of software quality, such as of architecture, reuse description, coding styles, test completeness, run-time efficiency, and resource usage efficiency, have not been modeled.

Another gap in ISO 9126's quality characteristic set is the lack of exception handling capability requirements for software as stated in Theorem 5.3, which is an important attribute of software quality that identifies the unexpected circumstances and conditions of a system, and specifies how the system should behave under such conditions. Design for exception handling capability of software is recognized as a good indicator to distinguish naive and professional software engineers and system analysts, even though a customer has not explicitly requested this kind of built-in software quality.

Further, it is found that the concept of software quality might be different between vendor-specified (common system) software and user-specified (applications) software. For the former, quality refers to the software that provides much more usability and higher dependability at a comparable price; while for the latter, quality means the software that meets users' requirements and runs with fewer failures. Also, it is considered that we need to distinguish the quality of software according to its developing processes. For example, we may identify the design quality, implementation quality, test quality, and maintenance quality of a software system, rather than pursuing a hybrid concept of the quality of software.

Generally, the philosophy behind current software quality standards is based on a generic quality system perception on software development. Although this philosophy has been proven successful in conventional manufacturing industry, there is still a need for supporting evidence of its effectiveness and impact on the design-intensive software engineering and the nonconventional software industry.

11.5 Software Engineering Management

As analyzed in Sections 8.5 and 11.2.3, the key difference between knowledge-based and labor-intensive engineering projects is determined by the factor of interpersonal coordination rate *r*. A high degree of interpersonal coordination in a creative software development project, for example, $r >$ 50%, would dramatically changed the behavior of such projects. Therefore, the coordinative work organization theories and laws, management methodologies, decision theories, and quality system theories presented in preceding sections lay the foundation for software engineering project management and quality assurance.

11.5.1 TAXONOMY OF SOFTWARE ENGINEERING MANAGEMENT

The managerial foundations of software engineering are cross fertilized by researches in management science, systems theory, and quality system principles. A brief structure of the management foundations of software engineering is summarized in Table 11.20.

The basic theories for software engineering management listed in Table 11.20 are sociology, anthropology, semiotics, linguistics, and psychology. *Sociology* concerns organizational theory; *anthropology* addresses organizational culture; *semiotics* relates to the theories of communication and knowledge; *linguistics* studies language theory; and *psychology* concerns human behavior and learning.

The system science foundations for software engineering management encompass abstract systems theory, system design and analysis, system modeling and simulation. *Systems theory* is a common foundation for management science and many other engineering disciplines. Systems theory as presented in Chapter 10 has provided interdisciplinary and strategic solutions that are qualitative and quantitative, organized and creative, theoretical and empirical for a wide range of problems.

Table 11.20
Structure of Managerial Foundations of Software Engineering

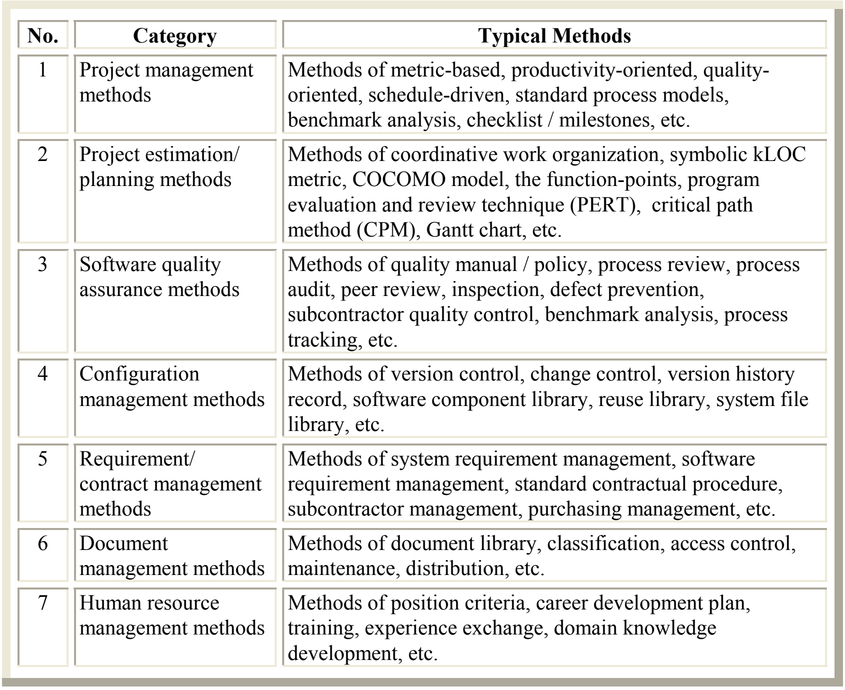
No	Category	Subcategory
1	Basic theories	
1.1		Sociology
1.2		Anthropology
1.3		Semiotics
1.4		Linguistics
1.5		Psychology
2	System science	
2.1		Abstract systems theory
2.2		System design and analysis
2.3		System modeling and simulation
3	Management theories	
3.1		Strategic planning
3.2		Operational theory
3.3		Decision theory
3.4		Organization methods
3.5		Management economics
4	Quality system principles	
4.1		Total Quality Management (TQM)
4.2		Business process reengineering
4.3		The Deming circle: Plan-Do-Check-Act (PDCA)

Management science is a scientific approach to solving system problems in the field of management. It encompasses operational theory [Fabrycky et al. 1984], decision theory [Keen and Morton, 1978; Steven, 1980; Wang and Ruhe, 2007], organization methods [Radnor, 1970; Kolb, 1970], strategic planning [Anthony, 1965; Khaden and Schultzki, 1983; William, 1991], quality theories [Shewhart, 1939; Crosby, 1977; Deming, 1992/86; Juran, 1988/89; Huran et al., 1962/80; Taguchi, 1986], and management economics [Richardson, 1966]. Management science provides management with a variety of decision aids and rules.

A set of *quality system principles* has been developed in management science. The important quality management philosophies that are applicable to software engineering organization and management are TQM [Deming, 1982; EFQM, 1993; Dunn and Ullman, 1994], business process reengineering [Schein, 1961; Johansson et al., 1993; Thomas, 1994; Wang and King, 2000a], and the Deming Circle [Deming, 1982].

Studies on organization and management of software engineering have, over time, covered methodologies for project management, project estimation, project planning, software quality assurance, configuration management, requirement/contract management, document management, and human resource management. Table 11.21 provides a summary of the software engineering organization and management methodologies in practice. It may be seen that software engineering management and organization methodologies have intensively influenced by management science and system science. However, one of its outstanding characteristics is the very high interpersonal coordination rate in organization and management, which makes software engineering unique and requires special care for dealing with the cognitive and organizational constraints according to the cooperative work organization theory presented in Section 8.5 [Wang, 2007].

Table 11.21
Classification of Software Engineering
Organization and Management Methodologies



No.	Category	Typical Methods
1	Project management methods	Methods of metric-based, productivity-oriented, quality-oriented, schedule-driven, standard process models, benchmark analysis, checklist / milestones, etc.
2	Project estimation/ planning methods	Methods of coordinative work organization, symbolic kLOC metric, COCOMO model, the function-points, program evaluation and review technique (PERT), critical path method (CPM), Gantt chart, etc.
3	Software quality assurance methods	Methods of quality manual / policy, process review, process audit, peer review, inspection, defect prevention, subcontractor quality control, benchmark analysis, process tracking, etc.
4	Configuration management methods	Methods of version control, change control, version history record, software component library, reuse library, system file library, etc.
5	Requirement/ contract management methods	Methods of system requirement management, software requirement management, standard contractual procedure, subcontractor management, purchasing management, etc.
6	Document management methods	Methods of document library, classification, access control, maintenance, distribution, etc.
7	Human resource management methods	Methods of position criteria, career development plan, training, experience exchange, domain knowledge development, etc.

11.5.2 THE SOFTWARE ENGINEERING PROCESS REFERENCE MODEL (SEPRM)

The Software Engineering Process Reference Model (SEPRM) [Wang *et al.* 1998b/99a; Wang and King 2000a] identifies a superset of processes and BPAs that covers the domains of current process models and new areas for software engineering environment and supporting tools. The philosophy of SEPRM is to provide a comprehensive and integrated software engineering process system reference model. SEPRM demonstrates a unified process system framework for Process-Based Software Engineering (PBSE), in which all current process models can be located.

SEPRM develops a basis against which process capability levels between existing process models can be systematically and quantitatively compared. It also allows transformation between process capability levels within different process models, and it enables software development organizations to relate their process capabilities to a range of different process models.

The following subsections describe the process model, process capability model, and process assessment method of SEPRM.

11.5.2.1 The SEPRM Process Model

SEPRM provides a hierarchical software engineering process framework with 3 process subsystems, 12 process categories, 51 processes, and 444 Base Process Activities (BPAs). The structure of the SEPRM process model is shown in Fig. 11.21 [Wang and King, 2000a].

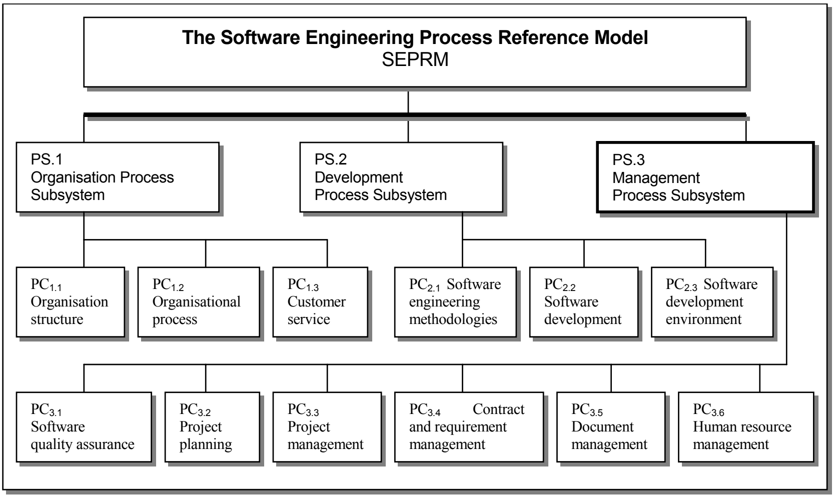
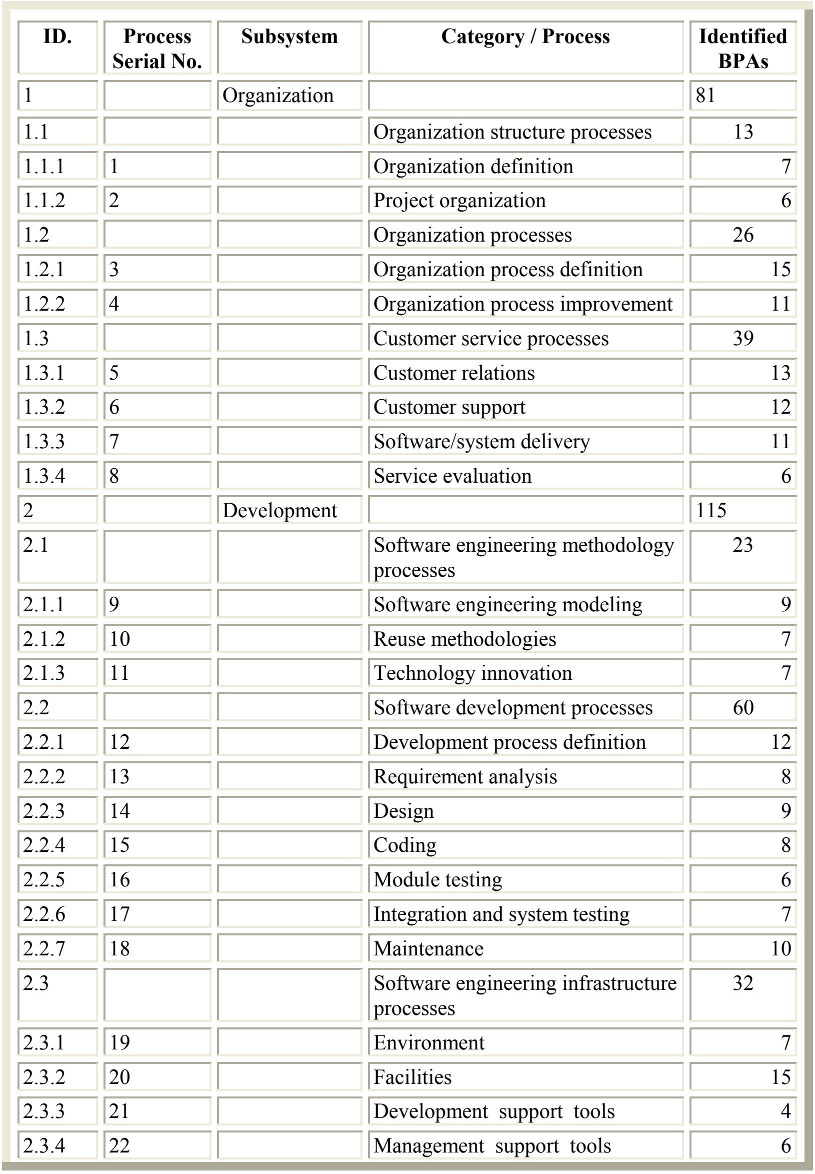

The Software Engineering Process Reference Model
SEPRM
PS.1 Organisation Process Subsystem
PS.2 Development Process Subsystem
PS.3 Management Process Subsystem
$PC_{1.1}$ Organisation structure
$PC_{1.2}$ Organisational process
$PC_{1.3}$ Customer service
$PC_{2.1}$ Software engineering methodologies
$PC_{2.2}$ Software development
$PC_{2.3}$ Software development environment
$PC_{3.1}$ Software quality assurance
$PC_{3.2}$ Project planning
$PC_{3.3}$ Project management
$PC_{3.4}$ Contract and requirement management
$PC_{3.5}$ Document management
$PC_{3.6}$ Human resource management

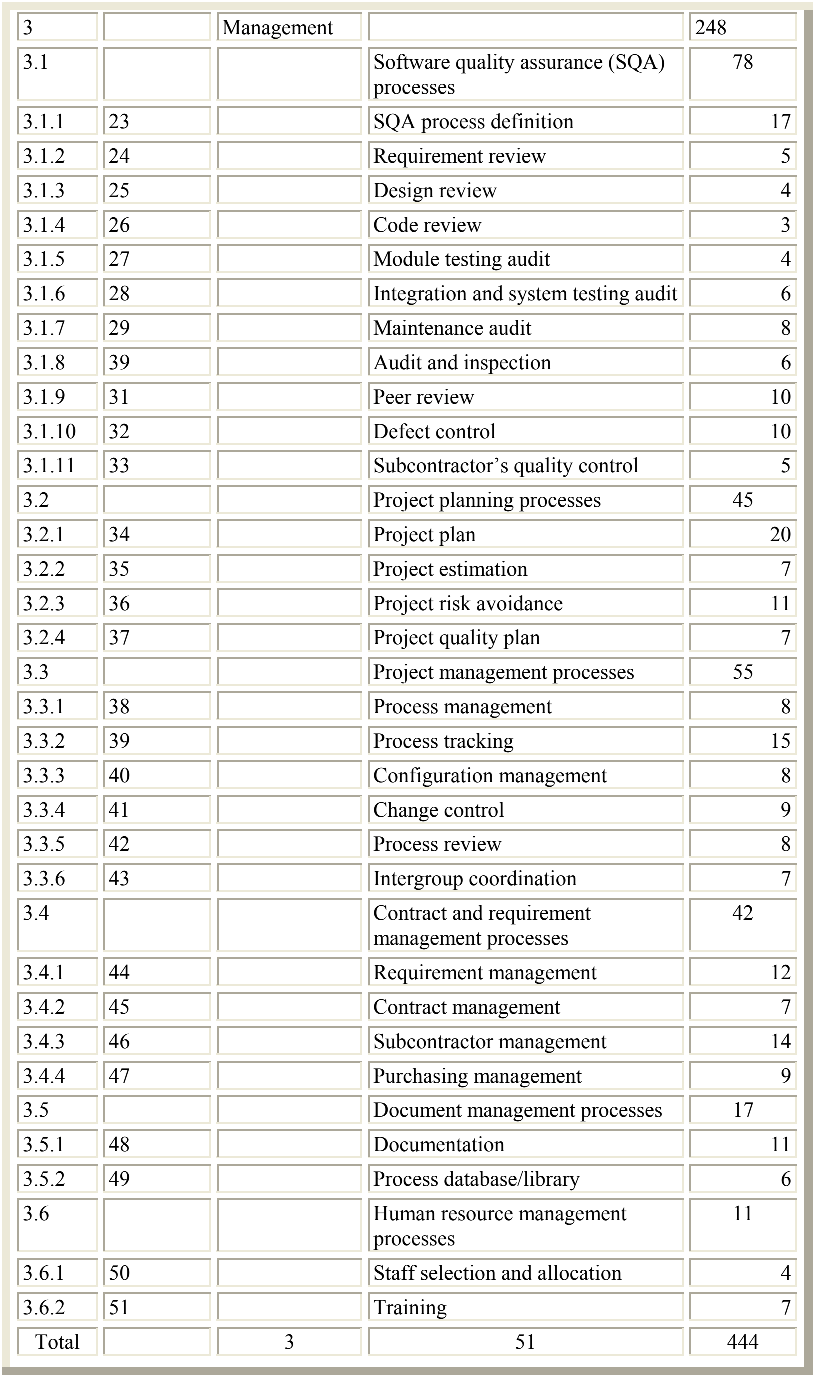
Figure 11.21 The hierarchical structure of SEPRM

The 51 processes of SEPRM and the configuration of the 444 BPAs in the process system are shown in Table 11.22.

Table 11.22
The SEPRM process model



ID.	Process Serial No.	Subsystem	Category / Process	Identified BPAs
1		Organization		81
1.1			Organization structure processes	13
1.1.1	1		Organization definition	7
1.1.2	2		Project organization	6
1.2			Organization processes	26
1.2.1	3		Organization process definition	15
1.2.2	4		Organization process improvement	11
1.3			Customer service processes	39
1.3.1	5		Customer relations	13
1.3.2	6		Customer support	12
1.3.3	7		Software/system delivery	11
1.3.4	8		Service evaluation	6
2		Development		115
2.1			Software engineering methodology processes	23
2.1.1	9		Software engineering modeling	9
2.1.2	10		Reuse methodologies	7
2.1.3	11		Technology innovation	7
2.2			Software development processes	60
2.2.1	12		Development process definition	12
2.2.2	13		Requirement analysis	8
2.2.3	14		Design	9
2.2.4	15		Coding	8
2.2.5	16		Module testing	6
2.2.6	17		Integration and system testing	7
2.2.7	18		Maintenance	10
2.3			Software engineering infrastructure processes	32
2.3.1	19		Environment	7
2.3.2	20		Facilities	15
2.3.3	21		Development support tools	4
2.3.4	22		Management support tools	6



3		Management		248
3.1			Software quality assurance (SQA) processes	78
3.1.1	23		SQA process definition	17
3.1.2	24		Requirement review	5
3.1.3	25		Design review	4
3.1.4	26		Code review	3
3.1.5	27		Module testing audit	4
3.1.6	28		Integration and system testing audit	6
3.1.7	29		Maintenance audit	8
3.1.8	39		Audit and inspection	6
3.1.9	31		Peer review	10
3.1.10	32		Defect control	10
3.1.11	33		Subcontractor's quality control	5
3.2			Project planning processes	45
3.2.1	34		Project plan	20
3.2.2	35		Project estimation	7
3.2.3	36		Project risk avoidance	11
3.2.4	37		Project quality plan	7
3.3			Project management processes	55
3.3.1	38		Process management	8
3.3.2	39		Process tracking	15
3.3.3	40		Configuration management	8
3.3.4	41		Change control	9
3.3.5	42		Process review	8
3.3.6	43		Intergroup coordination	7
3.4			Contract and requirement management processes	42
3.4.1	44		Requirement management	12
3.4.2	45		Contract management	7
3.4.3	46		Subcontractor management	14
3.4.4	47		Purchasing management	9
3.5			Document management processes	17
3.5.1	48		Documentation	11
3.5.2	49		Process database/library	6
3.6			Human resource management processes	11
3.6.1	50		Staff selection and allocation	4
3.6.2	51		Training	7
Total		3	51	444

11.5.2.2 The SEPRM Process Capability Model

Parallel to the SEPRM process dimension, the process capability model describes another dimension of SEPRM that provides an assessment framework for each process defined in the process model.

The SEPRM process capability model consists of a practice performance scale, a process capability scale, and a process capability scope. A practice performance rating scale for the BPAs in SEPRM is defined in Table 11.23. It employs a four-level scale for evaluating a BPA's existence and performance. The rating thresholds provide a set of quantitative measurement for rating a BPA's performance with the scale.

Table 11.23
Performance Rating Scale of the BPAs

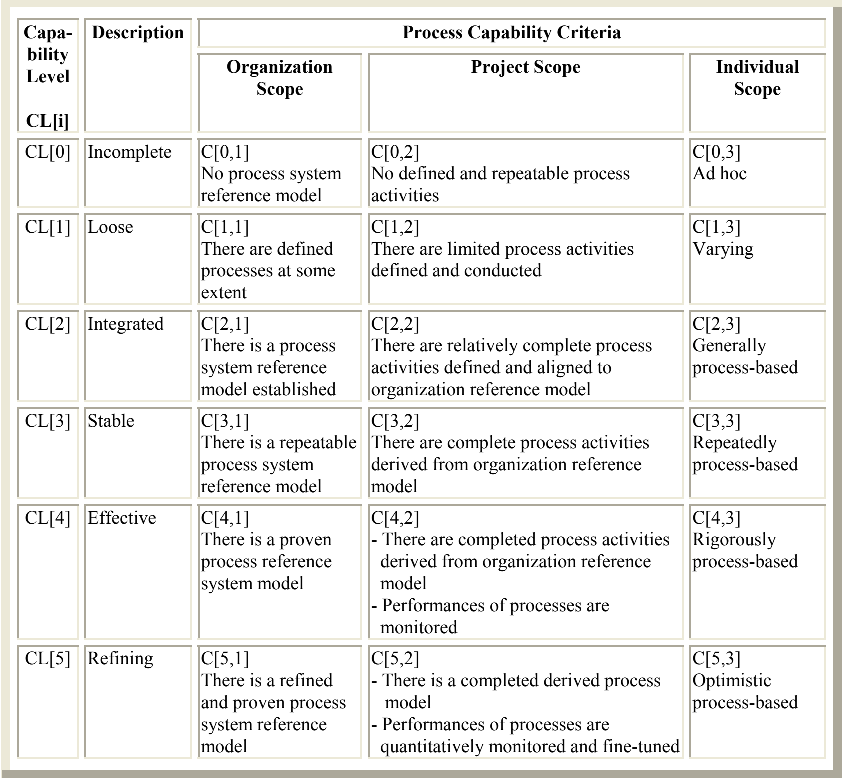
Scale	Description	Rating threshold
5 (F)	Fully adequate	90% - 100%
3 (L)	Largely adequate	70% - 89%
1 (P)	Partially adequate	35% - 69%
0 (N)	Not adequate	0 – 34%

There are three types of process capability scales: the pass-threshold-based, process-management-oriented, and process-oriented as shown in Table 11.24. The SEPRM process capability model is designed for directly rating and characterizing the performance of a process within context, rather than to indirectly evaluate the management maturity level for a process.

SEPRM develops a six-level software process capability model as shown in Table 11.24, with a set of defined criteria for rating the capability of a process. Table 11.24 shows that, in SEPRM, a process as an independent unit is assessed in the organization, project, and individual contexts against the six level process capability criteria. In order to relate the process capability criteria to the performance of BPAs in a process, there is an additional threshold for assessing a process. This is the average performance of the BPAs. Thus, based on both the software process capability model and the BPA performance threshold, an SEPRM process capability scale is described in Table 11.25.

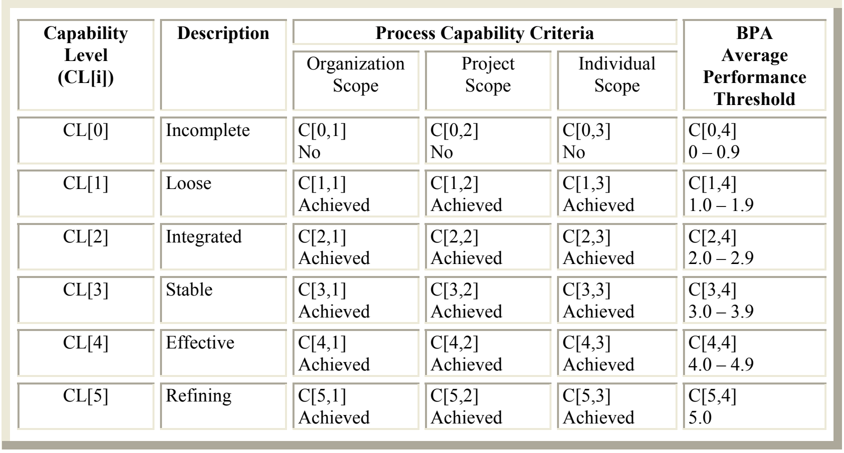
It can be seen in Table 11.25 that there are four criteria that a process has to fulfill at each capability level. The first three are oriented to a process as a whole; while the last one is oriented to BPAs contained in a process. Therefore, the capability of a software development organization to operate a given process is determined by the maximum level *i* that a process achieved for fulfilling all four criteria for that level.

Table 11.24
The SEPRM Process Capability Model



Capa-bility Level CL[i]	Description	Process Capability Criteria		
		Organization Scope	Project Scope	Individual Scope
CL[0]	Incomplete	C[0,1] No process system reference model	C[0,2] No defined and repeatable process activities	C[0,3] Ad hoc
CL[1]	Loose	C[1,1] There are defined processes at some extent	C[1,2] There are limited process activities defined and conducted	C[1,3] Varying
CL[2]	Integrated	C[2,1] There is a process system reference model established	C[2,2] There are relatively complete process activities defined and aligned to organization reference model	C[2,3] Generally process-based
CL[3]	Stable	C[3,1] There is a repeatable process system reference model	C[3,2] There are complete process activities derived from organization reference model	C[3,3] Repeatedly process-based
CL[4]	Effective	C[4,1] There is a proven process reference system model	C[4,2] - There are completed process activities derived from organization reference model - Performances of processes are monitored	C[4,3] Rigorously process-based
CL[5]	Refining	C[5,1] There is a refined and proven process system reference model	C[5,2] - There is a completed derived process model - Performances of processes are quantitatively monitored and fine-tuned	C[5,3] Optimistic process-based

Table 11.25
The SEPRM Process Capability Scale



Capability Level (CL[i])	Description	Process Capability Criteria			BPA Average Performance Threshold
		Organization Scope	Project Scope	Individual Scope	
CL[0]	Incomplete	C[0,1] No	C[0,2] No	C[0,3] No	C[0,4] 0 – 0.9
CL[1]	Loose	C[1,1] Achieved	C[1,2] Achieved	C[1,3] Achieved	C[1,4] 1.0 – 1.9
CL[2]	Integrated	C[2,1] Achieved	C[2,2] Achieved	C[2,3] Achieved	C[2,4] 2.0 – 2.9
CL[3]	Stable	C[3,1] Achieved	C[3,2] Achieved	C[3,3] Achieved	C[3,4] 3.0 – 3.9
CL[4]	Effective	C[4,1] Achieved	C[4,2] Achieved	C[4,3] Achieved	C[4,4] 4.0 – 4.9
CL[5]	Refining	C[5,1] Achieved	C[5,2] Achieved	C[5,3] Achieved	C[5,4] 5.0

The SEPRM process assessment results are reported at the six levels with a decimal value. This means it has the potential to distinguish the process capability at tenth-sublevels. This approach enables a software development organization to fine-tune its process system in continuous process improvement.

11.5.2.3 The SEPRM Process Capability Determination Methodology

Using the formal definitions of the SEPRM process model and process capability model as developed in previous subsections, we can now consider how to apply the latter to the former for the assessment of process capability at practice, process, project, and organization levels. This activity is called process capability determination.

The SEPRM process capability determination methodology can be described as an algorithm as shown bellow.

Algorithm 11.1 The SEPRM Process Capability Determination Algorithm

```
Assume: $N_{PC}(SUBSYS)$                    - Number of process categories in a
                                              process subsystem
        $N_{PROC}(SUBSYS,PC)$               - Number of processes in a category
        $N_{BPA}(SUBSYS,PC,PROC)$  - Number of BPAs in a process
        $BPA(SUBSYS,PC,PROC)$  - A BPA index
        $CL$                                - A capability level
        $PCL_{proc}(SUBSYS,PC,PROC)$        - A process capability level
        $PCL_{proc}$                        - Capability level of a project
Input:   Sample indicators of BPA and processes existence and performance
Output:  A process profile: $PCL_{proc}[SUBSYS,PC,PROC]$, and a project process
         capability level: $PCL_{proj}$
Begin

// Step 1: Initialization
  // Define numbers of processes in each process subsystem and category according to
     [Wang and King 2000]
  // Define numbers of BPAs in each process [Wang and King 2000]

// Step 2: Practice performance rating

  for $SUBSYS := 1$ to $3$ do                         // the process subsystem index
      for $PC := 1$ to $N_{PC}(SUBSYS)$ do            // the process category index
          for $PROC := 1$ to $N_{proc}(SUBSYS, PC)$ do   // the process index
              begin
                 $PP(SUBSYS, PC, PROC) := 0;$

                 for $BPA := 1$ to $N_{BPA}(SUBSYS, PC, PROC)$ do
                                                     // The BPA index
                    begin
                       // Assess a BPA according to Table 11.23, and
                       // record performance rating in $BPA(SUBSYS, PC, PROC)$
                       case $BPA(SUBSYS, PC, PROC)$
                            $F$:  // Fully adequate
```

```
                              PP(SUBSYS, PC, PROC) :=
                                 PP(SUBSYS, PC, PROC) + 5;
                           L:  // Largely adequate
                              PP(SUBSYS, PC, PROC) :=
                                 PP(SUBSYS, PC, PROC) + 3;
                           P:  // Partially adequate
                              PP(SUBSYS, PC, PROC) :=
                                 PP(SUBSYS, PC, PROC) + 1;
                           N:  // Not adequate
                              PP(SUBSYS, PC, PROC) :=
                                 PP(SUBSYS, PC, PROC) + 0;
                           NA: // Does not apply
                               PP(SUBSYS, PC, PROC) :=
                                  PP(SUBSYS, PC, PROC) + 5;
                           end;
                       end;
end;

// Step 3: Process capability determination

  for  SUBSYS := 1 to 3 do                                // the process subsystem index
     for  PC := 1 to $N_{PC}$(SUBSYS)  do                 // the process category index
        for  PROC :=1 to $N_{proc}$(SUBSYS, PC) do        // the process index
           // 3.1 Assess each process against the six level
           // process criteria as defined in Table 11.24
             $CL_{PROC}$(SUBSYS, PC, PROC) :=
                 $max \{ i \mid (C[i,j]$ are fulfilled$) \wedge j=1,2,3\}$;

           // 3.2 Assess mean BPA performance according to Table 11.25
             $CL_{BPA}$(SUBSYS, PC, PROC) := PP(SUBSYS, PC, PROC) /
                                            $N_{BPA}$(SUBSYS,PC,PROC);

           // 3.3 Determine process capability levels
             CL(SUBSYS, PC, PROC) :=
                 min {$CL_{PROC}$(SUBSYS,PC,PROC)+0.9,
                     $CL_{BPA}$(SUBSYS,PC,PROC)};

           // 3.4 Save process capability profile
             $PCL_{proc}$(SUBSYS,PC,PROC) := CL(SUBSYS,PC,PROC);

// Step 4: Project capability determination

  k := 51;                       // Number of PROCs defined in SEPRM
  CL := 0;
  for  SUBSYS := 1 to 3 do                             // the process subsystem index
     for  PC := 1 to $N_{PC}$(SUBSYS)  do              // the process category index
        for  PROC :=1 to $N_{proc}$(SUBSYS, PC) do     // the process index
           // Calculate cumulated process capability value
           CL := CL + $PCL_{proc}$(SUBSYS, PC, PROC);

  // Derive capability level of the project
  $PCL_{proj}$ := CL / k;                              // Calculate project capability level

End
```

Figure 11.22 The SEPRM Process Assessment Algorithm and Method

An SEPRM assessment according to Algorithm 11.1 is carried out in four steps:

a) *Initialization*: This step is designed to specify the numbers of BPAs defined in SEPRM. For obtaining a detailed configuration of BPAs in the SEPRM process model, reference may be made to Appendix C in [Wang and King, 2000].

b) *BPA performance rating*: In this step, all BPAs for each process are rated according to the definitions of practice performance scale in Table 11.23. The basic function of this step is to count the total values of the rated BPAs within individual processes.

c) *Process capability determination*: This step first derives both the process capability ratings by the process criteria (Table 11.24) and the BPA performance criteria (Table 11.25). Next, the capability level of the process is determined by taking into account both of the first three criteria (the qualitative score) and the fourth criterion (the quantitative score) according to the definitions in Table 11.25. Then, a process capability profile of an SEPRM assessment is created.

d) *Project process capability determination*: In the final step, the algorithm derives a process capability level for a software project based on all processes' capability levels derived in Step (c). The project capability level will be reported with the addition of a process capability profile.

SEPRM establishes a comprehensive and unified process system reference model that serves as an infrastructure for software engineering organization via PBSE. The development of SEPRM was based on the inspirations derived from existing process models and experience in empirical software engineering. From this we have gained improved understanding on software engineering and on software process system modeling as a key for organization and implementing software engineering. The SEPRM process model is supported by a set of industrial benchmarking data, and its process capability model is independently operational at levels of organization, project, and individual software engineers. SEPRM enables a derived process capability level to be transformed into other process models. It also allows, for the first time, capability levels from different process models to be related, transformed, and compared [Wang and King, 2000a].

11.6 Summary

Management science is the discipline that studies organizational behaviors, executive decision making, and resource optimization on given internal and external constraints. Management is both an organizational methodology and a profession, which is needed when people work together to achieve a result not possible by individuals acting alone. Management science encompasses operational theories, work organization, decision making, and quality systems. It studies the objects of work, people, resources, and processes, with emphases on productivity and quality.

It is recognized that, beyond programming and technical aspects of software development, software engineering deals with problems of organisation and management infrastructures. The work of software project managers is to balance competing demands for project scope, time, cost, risk, and quality. Therefore, they must satisfy stakeholders with differing needs and expectations and meet identified requirements constrained by the laws of work organization and management.

This chapter has discussed the roles of management in software engineering and the underpinning theoretical and empirical foundations. A set of classic management thought has been reviewed, and the fundamental principles and laws of management science have been formalized. On the basis of the management theories, two threads have been taken in this chapter on decision theories and quality principles. A structure of decision theories on static, dynamic, and serial decision making in complex management contexts, especially the formal game theory and decision grid theory, has been presented. Quality systems and principles for management have been rigorously discussed. Applications of management science and complicated management issues in software engineering have been described via the approach of process-based software engineering (PBSE). As a result, the **management science foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Management Science Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 11. Management Science Foundations of SE

■ Principles of Management Science
- Classic management thought
- Architecture of management science
  - Functions of management
  - The system model of management
- Fundamental theory of management science
  - Why is management needed in work organization?
  - The first principle of management
  - Gains from division of labor
  - The second principle of management
- The coordinative work organization theory and laws

■ Decision Theories
- The mathematical model of decision making
  - The principle of choices
  - Decisions and decision making
  - Strategies and criteria for decision making
  - The structure of rational decision making
- Decision making processes
  - The cognitive process of decision making
  - Formal description of the decision making process
- Static decision making strategies
  - Decision making under certainty
  - Decision making under uncertainty
  - Decision making under risks
- Game theory
  - The formal model of games
  - Properties of games
  - Behaviors of zero-sum games
  - Behaviors of nonzero-sum games
- Decision grid theory
  - The formal model of decision grids
  - Serial decision making with unlimited trials
  - Serial decision making with limited trials

■ Quality Systems
- Quality principles
  - Attributes of quality
  - Formal models of quality

- Quality control and assurance
  - Quality control systems
  - Quality assurance techniques
- Quality management systems
  - Total quality management (TQM)
  - The ISO 9000 quality system
  - The ISO 9126 quality system

■ Software Engineering Management

- Taxonomy of SE management
- The SE process reference model (SEPRM)
  - The SEPRM process model
  - The SEPRM capability model
  - The SEPRM capability determination methodology
- The coordinative work organization theory and laws for SE (Section 8.5)
- The Formal Economic Model for SE Costs (Section 12.6.2)

SIGNIFICANT FINDINGS OF THIS CHAPTER

• Although there are various objectives in management, the **key objective of management science** is not *management* but *work*. That is, management science studies how human work may be done coordinately, efficiently, qualitatively, and profitably in a systematic approach.

• Management science is a system science. In the **management system**, managers organize and coordinate the production or service processes to transfer the inputs into expected outputs. As shown in Fig. 11.2, the **inputs** of a management system encompass three essences known as *labor, time,* and *resources*; while the **outputs** of a management system also encompass three essences known as *productivity, profit/cost,* and *quality*.

• It is noteworthy that there are natural **laws that constrain the allocation of labor and time** for a given project. In other words, the optimal allocation of labor, time, and resources is not arbitrary and simply empirical; certain laws and constraints exist as described in Section 8.5, particularly by Theorems 8.2, 8.4, and 8.7.

• The **natural function of management** is **system synchronization**. Although the basic elements of management are planning, organization, control, and optimization, the **essence** of all management principles is system synchronization, which is also identified as one of the fundamental principles of system science.

- The **factors determining a decision** are the alternatives $\mathcal{A}$ and criteria C for a given decision making goal. A unified theory on fundamental and cognitive **decision making** can be developed based on the axiomatic and recursive cognitive process elicited from the simplest decision-making categories.

- The **taxonomy of strategies** and corresponding criteria for decision making may be classified into four categories known as *intuitive, empirical, heuristic,* and *rational.*

- The principle of **bounded rationality** states that a decision-maker in a real-world situation will never have all information necessary for making an optimal decision.

    - It is noteworthy that practical decisions for a given problem are usually made under partial certainty, empirical estimation, or heuristic prediction, because not all required information is available, no suitable decision strategy is aware of, and/or no acceptable cost to thoroughly search all possible alternatives.

- The **art of management**, to a certain extent, is to create a win-win environment for members, partners, and parent organizations involved in a game context.

- **Quality** is a collective attribute of a product, service, or system.

    - Quality is implemented incrementally via each individual in every process.

    - A major problem in software quality assurance is that there lacks a comprehensive and coherent set of quantitative measures and benchmarked standards for each of the basic attributes of software and software engineering processes. The lack of the measurability in software engineering is the fundamental barrier that prevents the software industry from achieving a predictable and stable quality in software engineering as its counterparts in other industries.

- The **conditions of quality control** states that the *necessary conditions* for implementing a quality control system for a given product, service, or system are that all attributes of its quality can be: a) abstractly identified; b) quantitatively defined, and c) independently measurable.

- A **quality control system** should be designed and implemented as a whole, because any individual quality assurance technique, no matter how effective, can not solve the problem alone in a given quality system.

- The fundamental principles and formal models for **software engineering organization and management:**
    - The coordinative work organization theory and laws (Chapter 8).
    - The system organization trees (SOTs, Chapter 10)
    - The principle of management gains
    - The principle of gains of division of labor
    - The quality models and the principle of quality control systems
    - The infrastructure of process-based software engineering
    - The Formal Economic Model of SE Costs (FEMSEC, Chapter 12)

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Principles of management science

- **Classic management thought and methodologies** can be traced back to the work of Frederick Taylor on operations studies in production, Henry Gantt on project scheduling and the Gantt Chart, William Shewhart on statistical quality control, John von Neumann on game theory, linear programming in the 1940s, Program Evaluation and Review Technique (PERT) in 1950s, nonlinear programming and dynamic programming in the 1950s, Critical Path Method (CPM) in 1960s, E. Erlang and John Little on queuing theory, Philip Crosby, Edwards Deming, Genichi Taguchi, and Joseph Juran on quality systems and quality control principles.

- **Management** is a coordination process that organizes activities and efforts of a group to achieve goals and results not possible by individuals.

- **The functions of management** are planning, organizing, controlling, and optimizing.

- **Planning** is a management process for organization, coordination, and estimation of project time and related labor and resource allocation.

- **Scheduling** is a management process that maps the planned activities onto the time axis in a parallel or serial structure or their combinations.

- **Organizing** is a management process that coordinates and allocates essential means, such as labor, resources, and processes, in order to implement a planned work.

- **Controlling** is a management process that monitors and ensures the planned work process and outcomes in operation conforming to predefined requirements, standards, and schedule.

- **Optimizing** is a management process that continuously improves the results of an organization or project in terms of higher productivity, better quality, more accurate scheduling, more efficient process, and lower costs.

- The **organization forms** in management:

  - A **natural group** is a working group of people with peers in which work is carried out via temporal pairwise coordination when work has to be done by any pair of the peers.

  - A **managed group** is a working group of people with peers and a manager, in which work is carried out via one-to-many coordination by the manager.

- The *number of interpersonal coordination* $C_2(n)$ needed in a **natural group** of size n, $n \geq 3$, can be determined by:

$$\begin{aligned} C_2(n) &= 2 \bullet \mathrm{C}_{\mathrm{n}}^2 \\ &= n \bullet (n - 1) \end{aligned}$$

- The *number of interpersonal coordination* $C_m(n)$ needed in a **managed group** of size n, $n \geq 3$, can be determined by:

$$C_{\mathrm{m}}(n) = n + 1$$

- **The first principle of management:** The *gain of management* states that management is required to reduce the complexity of working group organization, to improve the efficiency of groups, and to simplify the forms of interpersonal coordination.

- **The second principle of management:** *Division of labor* (DOL), or specialization on a specific subtask in a process, is a work organization method in which a task is divided into a sequence of multiple subtasks, and a person is only specialized in a repeatable subtask.

- The **natural work allocation** is a form of loosely coupled work organization that requires an invariable effort $E(1)$ with a relative value 1, i.e., $E(1) = 1$.

- The **specialized work allocation** $E_d(1)$ is a work organization method that allocates tasks via DOL, which results in the saving of effort proportional to times of repetition k in an inversed exponential rate determined by a constant e/c, i.e., $E_d(1) = (\frac{e}{c})^{k-1}$, where c is determined empirically based on the skilled rate of repetition for a given task, $1 < c < e$.

- The **gain of division of labor** states that the relative gain $g_r(k)$ via division of labor in work organization is proportional to the repetitive times k at specialized subtask-level (see Theorem 11.2).

Decision theories

- **Decision making** is one of the basic cognitive processes of human brains, by which a preferred option or a course of actions is chosen from among a set of alternatives based on certain criteria.

- A **decision** d is a selected alternative $a \in \mathcal{A}$ from a nonempty set of alternatives $\mathcal{A}$, $\mathcal{A} \subseteq U$, based on a given set of criteria C, i.e., $d = f\colon \mathcal{A} \times C \to \mathcal{A}$.

- **Decision making** is a process of decision selection from available alternatives against the chosen criteria for a given decision goal.

- The **number of possible decisions**, n, can be determined by the sizes of $\mathcal{A}$ and C, i.e., $n = \#\mathcal{A} \bullet \#C$.

- **Rational and complex decision making** strategies can be classified into the static and dynamic categories.

  - Most existing decision-making strategies are **static decisions** because the changes of environments of decision makers are independent of the decision makers' activities. Also, different decision strategies may be selected in the same situation or environment based on the decision makers' values and attitudes towards risk and their prediction on future outcomes.

- When the environment of a decision maker is interactive with his/her decisions or the environment changes according to the decision makers' activities and the decision strategies and rules are predetermined, this category of decision making needs is classified into the category of **dynamic decisions**, such as games and decision grids.

- The **dynamic strategies and criteria** of decision making are those that all alternatives and criteria are dependent on both the environment and the effect of the historical decisions made by the decision maker.

- **The cognitive process** of **decision making** can be carried out by the following procedures:

a) To comprehend the decision making problem, and to identify the decision goal in terms of an Object (O) and its attributes (A).

b) To search in the abstract layer of LTM for alternative solutions ($\mathcal{A}$) and criteria or useful decision strategies (C).

c) To quantify $\mathcal{A}$ and C, and determine if the search should go on.

d) To build a set of decisions by using $\mathcal{A}$ and C as obtained in above searches.

e) To select the preferred decision(s) on the basis of satisfaction of decision makers.

f) To represent the decision(s) in a new sub-OAR model.

g) To memorize the sub-OAR model in LTM.

- A **static strategy and criterion** of decision making is an evaluation and selection method for which all alternatives $\mathcal{A}$ and criteria C are determinable and only one optimal decision $a_i \in \mathcal{A}$ is expected for a given situation.

- A **decision making under certainty** d_{max} or d_{min} is a selection of a certain alternative a_i among $\mathcal{A}$ that meets a given criterion C which is either the maximum of utility or profit $max(u_i)$, and the minimum of costs or effort $min(e_i)$.

- An ***optimistic decision making under uncertainty*** $d_{maximax}$ or $d_{minimin}$ yields a decision with the *maximum-maximum* strategy for utility or a *minimum-minimum* strategy for cost, respectively.

- A **pessimistic decision making under uncertainty** $d_{maximin}$ or $d_{minimax}$ yields a decision with the *maximum-minimum* strategy for utility or a *minimum-maximum* strategy for cost.

- A **minimum regret decision making under uncertainty** $d_{minimax}$ yields a decision with the *minimum-maximum regret* strategy for utility gain or cost save.

- A **decision making under risk with maximum expected utility** d_{maxEU} yields a decision with the *maximum expected utilities* of all alternatives.

- A **decision making under risk with maximum utility of maximum probability** $d_{maximax\text{-}p}$ yields a decision with the *maximum* utility of the *maximum* probability of outcome of all alternatives.

- The **dynamic strategies and criteria** of decision making are those that all alternatives and criteria are dependent on both the environment and the effect of the historical decisions made by the decision maker.

- In **classic decision theories**, although the states of nature or environment may be deterministic or nondeterministic, its state of nature as an outcome of the environment will not be changed or affected by the decision maker's actions. In other words, there are natural rules but no adaptive competitors in the static decision making processes. However, more decision making situations are dynamic rather than static, where the decision maker is under competition in games.

- A **game** is a decision process under competition where opponent players or opponent groups of players compete for the maximum gain or a success state in the same environment according to the same predetermined rules of the game.

  - Games are traditionally dealt with probability-based static payoff tables. However, this method is found inadequate to deal with the dynamic behaviors of games and to rigorously determine the outcomes of games.

- A **formal game** G is a 4-tuple, i.e., $G = (P, D, M, S)$, where P is a finite set of *n players*, $n \geq 2$; D is a finite set of *k decisions* for certain *moves*, $k \geq 1$. M is a finite set of *q matches* between player, $q \geq 1$; and S is a finite set of cumulated *scores* for each player.

- A **match** $m \in M$ of a game $G = (P, D, M, S)$ is a function that maps a set of n decisions made by each player into a set of n scores S for each of the players, i.e., $m = f_m : D_1 \times D_2 \times \ldots \times D_n \rightarrow S$.

- The **number of individual matches** n_m in a set of matches for a given game $G = (P, D, M, S)$ is determined by $n_m = k^n$, where n is the number of players in a game, and k is the number of alternative decisions (moves) defined in the game for each player.

- The **total sets of matches** n_s in a game $G = (P, D, M, S)$, in which all players may use each pair of their alternative strategies only once determined according to the current move of opponent and the rule of the maximum gains based on the given layout of the game, can be determined by $n_s = n \bullet k$, where k is the number of alternative decisions (moves) defined in the game, and n is the number of players.

- The **total number of matches** q of a game $G = (P, D, M, S)$ is determined by the number of sets of matches n_m and number of matches in each set n_s, i.e., $q = n_s \bullet n_m = n \bullet k^{n+1}$.

- A **zero-sum game** is a game where the total scores of all n players in the game is zero.

- The **condition for a zero-sum game** is *iff* that each of the n_m individual matches is zero-sum.

- The **scores of all sets of matches** of formal games G are the same, no matter who moves first and which strategy (decision alternative) is selected for the first move.

- A **nonzero-sum game** is a game where the total scores of all players in the game is a positive nonzero value.

- A **decision grid** is a directed network of a series of decisions over time where each decision possesses only two possible outcomes, right or wrong, where the effort spent to make a right decision is considered to be identical with that of a wrong decision.

- Decision grids can be applied in a wide range of serial and dynamic decision making situations.

- The **formal model of a decision grid** DG is a 4-tuple, i.e., $DG = (T, D, E, S)$, where T is a finite or infinite set of n *trials* and n is the time points of trials where n may be infinitive; D is the *decision distance* of a series of decision trials, $D = t_i - t_0 = t_i$, $1 \leq i \leq n$; E is the effort of a specific trial towards the success state in the grid, $0 \leq E \leq n$; and S is a finite or infinite set of *success states* of the grid, $S = \{s_1, s_2, \ldots, s_k\}$, $1 \leq k \leq n$.

• When the allowable times of trials t in a decision grid are infinitive, the decision grid is called an **unlimited decision grid**; otherwise, it is a *limited decision grid.*

• The **properties of decision grid** state that the decision distance D_t in a decision grid is a constant that is determined by the number of decision trials t_i spent in the time series, i.e., $D_t = t_i = d_r + d_w$, where d_r and d_w represent numbers of right and wrong decisions, respectively.

• The later the wrong decision is corrected, the higher the cost of the **decision series**. The earlier the wrong decision is corrected, the more efficient of a decision series.

• The **random series of unlimited trials** or equal probability right and wrong trials will not lead to a success in any series of decisions under unlimited trials.

• The **random series of limited trials** or equal probability right and wrong trials will not lead to a success in any series of decisions under limited trials.

Quality theories

• **Quality** is the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Quality before quantity is a basic principle in management science.

• The **Garvin's eight dimensions of quality** are *performance, features, reliability, conformance, durability*, *serviceability*, *aesthetics*, and *perception of excellence*.

• The **external quality attributes of software**: *functionality, reliability, conformance to requirements, usability, efficiency, maintainability,* and *portability.*

• The **internal quality attributes of software:** *completeness, correctness, consistency, clearness* (no ambiguity)*, feasibility* (can be implemented in technical and economical terms), and *verifiability* (can be measured).

- **Quality** is a generic measure of the degree of excellence of a product or service against a given standard. More specifically, quality is a common attribute of any product or service that characterizes the quantity of both utility and durability of the product or service.

    - **Quality** Q is a generic and collective attribute of a product, a service, or a system that is proportional to both its average utility U and the available duration T of the utility, i.e., $Q = U \bullet T$ [Fh], where the unit of utility is *function* (F), and the unit of duration is *hour* (h), and these result in the **unit of quality** as *Function-hour* or shortly Fh.

    - The **dynamic value of quality** $Q(t)$ is an integral of the utility function $U(t)$ on the entire lifecycle of the utility $[0, T]$.

    - The **relative quality** $q(t)$ is a relative degree of difference between the quality of a product, a service, or a system and the standard or benchmark S for the expected quality.

- The **benefit of a product or a system** B is the quality gained per unit cost (C) in terms of resources, labor, and time.

- The **quality funnel principle** states that the nearer to the start of the production process, the lower the cost of quality.

- A **generic quality control system** encompasses five subsystems known as quality definition, implementation, appraisal, postmortem, and prevention.

    - The **quality definition** subsystem is responsible to identify, partition, and quantify the attributes and characteristics of the products or services produced or provided in an organization.

    - The **quality implementation** subsystem distributes quality attributes identified in phase one into individual processes and job functions.

    - The **quality appraisal** subsystem is a set of evaluation techniques against the quality standards for each process and each attribute of a given product, service, or system.

    - The **quality postmortem** subsystem is a feedback subsystem that helps to identify existing or potential problems in the process or quality standards on the basis of operating data on current performance.

    - The **problem prevention** subsystem is an adaptive process that prevents recurrent problems or failures from happening through improvement of the current processes and quality standards.

- Typical **quality assurance** techniques in quality control systems can be classified into *definitive, implemental, appraisable, postmortem,* and *preventive* corresponding to the quality control system model.

  - **Audit** is an empirical quality assurance technique that uses professional auditors to monitor the quality of products and services on the basis of statistical quality control and review techniques.

  - **Review** is a special audit technique for information-based work products such as system designs, plans, software, and documents. Quality review is an effective technique for quality assurance in software engineering.

  - **Measurement** is a quantitative quality assurance technique that evaluates the conformance of products and services against predefined standards or benchmarks.

  - **Verification and validation** is a formal quality assurance technique that applies mathematical models, logical inference tools, and simulation systems into the quality control of products and services.

- **Total Quality Management** (*TQM*) is a systematical management methodology that states quality of products and services of an organization depends on a systematical management of the organization's culture, attitude, and operations through all members' involvement.

Software engineering management

- **Software quality** is a set of inherent internal and external characteristics of a software system that show relative advantages over similar systems or indicate a conformance to a standard.

- **Exception handling** capability is a necessary attribute of software quality, which identifies the unexpected circumstances and conditions of a system, and specifies how the system should behave under such conditions.

  - **Design for exception handling** capability of software is recognized as a good indicator to distinguish naive and professional software engineers and system analysts, even for a customer who has not explicitly required for this kind of built-in software quality.

- The concept of software quality might be different between **vendor-specified** (common system) software and **user-specified** (applications) software. For the former, quality refers to the software that provides much more usability and higher dependability at a comparable price. While for the latter, quality means the software that meets the user's requirements and runs with fewer failures.

- There is a need to distinguish the **quality** of software according to its **developing processes**, such as design quality, implementation quality, test quality, and maintenance quality of a software system.

- The **philosophy** behind current **software quality standards** is based on a generic quality system perception on software development. Although this philosophy has been proven successful in conventional manufacturing industry, there is still a need for supporting evidence of its effectiveness and impact on the design-intensive software engineering and the nonconventional software industry.

- Studies of **software engineering organization and management** have, over time, covered methodologies for project management, project estimation, project planning, software quality assurance, configuration management, requirement/ contract management, document management, and human resource management.

- The **Software Engineering Process Reference Model** (SEPRM) identifies a superset of processes for software engineering. SEPRM provides a hierarchical software engineering process framework with 3 process subsystems, 12 process categories, 51 processes, and 444 **Base Process Activities** (BPAs). SEPRM demonstrates a unified process infrastructure for **PBSE**.

- The **process capability model** of SEPRM describes an assessment framework for each process defined in the process model. The SEPRM process capability model consists of a practice performance scale, a process capability scale, and a process capability scope.

- The **process capability determination methodology** of SEPRM is modeled in four steps: a) Initialization, b) BPA performance rating, c) Process capability determination, and d) Project process capability determination.

Questions and Research Opportunities

11.1 What are the nature and functions of management? Why is the basic object under study in management science not *management* but human *work*? How may this metaphor influence your view towards the nature of management science?

11.2 Explain the system model of management.

11.3 Discuss why management is needed in groups and organizations.

11.4 Why is management universally needed in work organization? What are the natural laws behind this generic phenomenon?

11.5 According to Theorem 11.1, explain what the gain of management is.

11.6 According to Theorem 11.2, explain what the gain of division of labor is.

11.7 Referring to Theorems 8.4 through 8.11, discuss how the coordinative work organization theory may play a fundamental role in management science.

11.8 Given the ideal workload of a software engineering project is expected to be W_1 = 36PM, find the optimum labor allocation and the shortest project durations for r_1 = 10% and r_2 = 50% according to Theorem 8.7.

11.9 Redo Exercise 11.7 for W_1 = 360PM.

11.10 Compare the results obtained in Exercises 11.7 and 11.8. Then, analyze and discuss which laws (as stated in Theorems 8.4 through 8.11) apply to the phenomena you observed.

11.11 Why should a manager of a large-scale software engineering project be cautious when the labor allocation is above 20 persons for the project?

11.12 Referring to the *System Organization Tree* (SOT) as presented in Section 10.3.5, discuss what are the optimal organizational forms for software engineering projects and groups?

11.13 What is a decision and what is the nature of decision making?

11.14 Based on Table 11.3, summarize the features and usages of the categories of intuitive, empirical, heuristic, and rational decision makings.

11.15 Summarize and contrast the formulae of static decisions in the three categories, i.e., decisions under certainty, uncertainty, and risk.

11.16 How may game theories be used in dynamic decision making?

11.17 What are the basic properties of games and why may games be extremely complicated?

11.18 What are the conditions for zero-sum and nonzero-sum games?

11.19 What are the applications of nonzero-sum game principles in management science and software engineering?

11.20 How may decision grid theory be used in dynamic decision making?

11.21 What are the basic properties of decision grids?

11.22 Why will random decisions, or equal probability right and wrong trials, not lead to a success in any series of decisions under limited or unlimited trials?

11.23 What is the nature (physical meaning) of quality? How to quantitatively and rigorously express the generic model of quality?

11.24 Describe the architecture of a generic quality control system.

11.25 What are the three conditions for quality control?

11.26 What are the four basic quality assurance techniques?

11.27 Referring to the philosophical and engineering considerations as presented in Chapter 4 and Chapter 8, discuss whether software quality technologies are necessarily the same as those of conventional manufacturing industries.

11.28 What are the fundamental management issues in software engineering in addition to the technical, cognitive, systematical, and economical issues?

11.29 According to Theorem 8.2, *conservation of basic engineering constraints*, explain why the basic objectives of software engineering such as time, costs, and utility are conservative and interlocked.

11.30 According to the 1st principle of management, Theorem 11.1, discuss how gains of management may be applied and implemented in software engineering.

11.31 According to the 2nd principle of management, Theorem 11.2, discuss how gains of division of labor may be applied and implemented in software engineering.

11.32 An important software engineering management principle is known as the process parallelism as adopted in SEPRM, which states there is a need to identify a management process for each technical processes in software engineering. Based on this principle, try to identify the management techniques in the processes parallel with the software processes of system design and testing.

11.33 Referring to the organizational and management theories presented in this chapter, discuss what kinds of decision optimizations may be implemented in each process of software engineering.

Hint: Use a table with the schema as follows: processes | work products | optimization strategies | decision methods

11.34 The process assessment method of SEPRM has been given in Algorithm 11.1. Try to translate the informal description of Algorithm 11.1 into a formal model described in RTPA.

11.35 The following experiment is designed to empirically prove the 2nd principle of management (Theorem 11.2) – gains of division of labor – by carrying out a text editing task for the given page below in an ordinary sequential way and a categorized way.

All typos underlined in the following page should be correct by corresponding capital letters. Record your time while editing in methods (a) and (b) as given below, i.e., t_1 and t_2, respectively.

a) Correct the typos in the given page sequentially line by line.

b) Correct the typos in the given page word by word vertically in three passes: The first pass checks for all F's, the second pass for all S's, and the third pass for all E's.

Then, calculate the gain of the simulated division-of-labor using the following formula: $G_{dol} = \frac{t_1 - t_2}{t_1} \bullet 100\%$.

Software Engineering Foundations:
A Software Science Perspective

Part I. Principles and Constraints of Software Engineering
1. Introduction
2. Principles of software Engineering

Part II. Theoretical foundations of Software engineering
3. Philosophical Foundations of software engineering
4. Mathematical foundations of Software engineering
5. Computing foundations of software Engineering
6. Linguistics Foundations of Software engineering
7. Information Science foundations of software Engineering

Part III. Organizational Foundations of software Engineering
8. Engineering foundations of software engineering
9. Cognitive Informatics Foundations of Software engineering
10. System Science foundations of software engineering
11. Management Science foundations of software Engineering
12. Economics Foundations of Software Engineering
13. Sociology foundations of software engineering

Part IV. Perspectives on Software Science
14. Retrospect on software engineering
15. Prospect on software Science

11.36 Read the following classic article in software engineering:

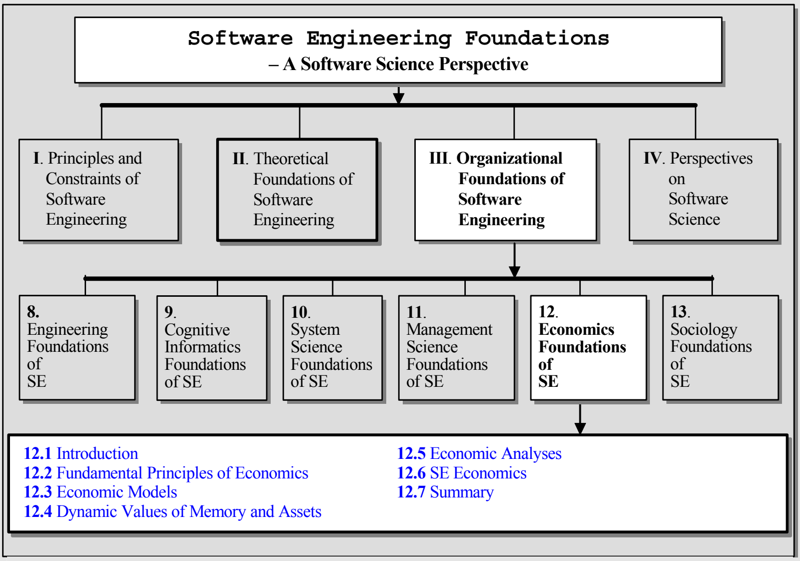
Frederick P. Brooks (1987), No Silver Bullet – Essence and Accident in Software Engineering, *IEEE Computer*, 20(4), pp.10-19.

Discuss the following topics in a group or individually:

- About the author.
- What are the basic constraints of software engineering according to the author in the 1980s?
- Are the author's conclusions too pessimistic? What would be the possible 'silver bullet(s)' in the future?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 12

ECONOMICS FOUNDATIONS OF SOFTWARE ENGINEERING



Software Engineering Foundations
– A Software Science Perspective

I. Principles and Constraints of Software Engineering

II. Theoretical Foundations of Software Engineering

III. Organizational Foundations of Software Engineering

IV. Perspectives on Software Science

8. Engineering Foundations of SE

9. Cognitive Informatics Foundations of SE

10. System Science Foundations of SE

11. Management Science Foundations of SE

12. Economics Foundations of SE

13. Sociology Foundations of SE

12. Economics Foundations of SE

Knowledge Structure

❍ Fundamental principles of economics

- Basic axioms of economics
- Economic equilibrium between demands and supplies
- The behaviors of market systems

❍ Economic models

- Production models
- Cost models
- Market models

❍ Dynamic values of money and assets

- Dynamics of money
- Dynamics of asset's values
- Cumulative values of cash flows

❍ Economic analyses

- Project cost analyses
- Project benefit-cost analyses
- Project payback period analyses
- Project rate of return analyses

❍ Software engineering economics

- Elements of SE costs
- SE project costs estimation using FEMSEC
- SE project costs estimation using COCOMO
- Economic analyses of software projects
- The software legacy cost model

Learning Objectives

- To gain knowledge on fundamental principles of economics and behaviors of economic systems interacting between demands and supplies.
- To understand basic economic models such as those of production, cost, market, equilibrium, benefit-cost ratio, payback period, and rate of return.
- To know dynamics of money, assets, investments, and cash flows, as well as their cumulative values over time.
- To understand the architecture of software engineering economics and the unique features of software and software engineering.
- To understand the formal economic model of software engineering costs (FEMSEC).
- To become familiar with software engineering economic analyses and cost/effort estimations.
- To understand the software legacy maintenance cost model in software engineering.

"Economists are not easy to follow ..., when they talk theory, even their fellow economists have difficulty understanding what they are saying. In fact, there is a theory that if all the economists in the world were laid out end to end, they still would not reach a conclusion."

Stephen L. Slavin (1988)

"Although a good deal of progress has been made in software cost estimation, a great deal remains to be done."

Barry Boehm (1984)

12.1 Introduction

Economics studies how people and resources are organized efficiently, effectively, and profitably for gaining the maximum for individuals, organizations, and the societies. Fundamental economic structures are the underlying forces of socialization and social organization. In turn, the fundamental economic structures are determined by the current and predominantly highest level of unsatisfied fundamental human needs [Wang, 2005d/05k]. Therefore, a successful software engineer requires certain knowledge of economics in addition to science and engineering.

Definition 12.1 *Economics* is the study of how resources are used to produce and distribute commodities and how services are provided in society.

Economics can be classified as *microeconomics* and *macroeconomics*. The former studies the behaviors of individual agents and industrial markets. The latter studies broad aspects of the economy, such as overall employment, export, and prices in a national or global scope.

A universal quantitative measure of commodities and services in economics is money.

Definition 12.2 *Money* is a generic representation of value and utility in terms of quantity of products, quality of services, and effort in production and services.

Therefore, in a certain extent, economics can be perceived as the science of money, or the production, consumption, and transfer of wealth.

Definition 12.3 *Engineering economics* is a branch of microeconomics dealing with engineering related economic decisions.

This chapter presents a transdisciplinary study on economics, its formalization, and its engineering application. The first part of this chapter presents classic thought of economics. A mathematical model of economical equilibrium is developed for formally and quantitatively explaining Adam Smith's genius hypothesis of the *invisible hand* proposed in 1776 [Smith, 1776; Cannan, 1994]. The equilibrium theorem can also be applied to more complicated multivariable equilibrium problems that conventional economic theories could not explain.

The second part of this chapter derives the theories and principles of *software engineering economics*. A set of economic models of software engineering and their formal description is provided. The applications of economic analysis and problem solving methodologies in a variety of software project decision making contexts are discussed. This leads to the development of the law of software legacy maintenance costs, and the finding of a hidden but significant phenomenon in software engineering known as the Software Maintenance Crisis (SMC) [Wang, 2005d], in which the maintenance costs overrun development costs at an exponential speed in the software development organizations. The complete model of MSC and possible solutions will be provided in Chapter 14.

This chapter introduces fundamental principles and methodologies utilized in engineering economics and their applications in software engineering. It also introduces formal methodology into economic analysis and modeling. A set of formal economic models will be developed based on fundamental principles of microeconomics. In the remainder of this chapter, the economic foundations of software engineering will be presented in five sections. Section 12.2 reviews fundamental principles of economics. Section 12.3 develops a set of formal economic models such as the production, costs, and market models. Section 12.4 discusses the dynamic values of money and assets, and their growth patterns. Section 12.5 describes economic analysis methodologies on engineering decisions such as project costs, benefit-cost ratio, payback period, and rate of return. On the basis of the formal treatment of economic theories and principles, Section 12.6 presents software engineering economics, particularly the theories and laws behind it, such as elements of software costs, software engineering project costs estimation, economic analyses of software engineering projects, and the software legacy maintenance cost model. As foundations for this chapter, the laws for optimal work and labor allocation have been discussed in Chapters 8 and 11 on engineering and management science foundations of software engineering, respectively.

12.2 Fundamental Principles of Economics

This section reviews the classic thought in economics with a formal treatment. Basic axioms of economics and the principle of resource scarcity are formally described. Based on the axioms, the profit-driven law of demands and supplies and the equilibrium between demands and supplies are quantitatively explained. The law of economic equilibrium is derived that rigorously describes the adaptive mechanisms of market systems, and serves as a formal proof of Adam Smith's genius hypothesis of *invisible hand* proposed in 1776 [Smith, 1776].

12.2.1 BASIC AXIOMS OF ECONOMICS

The entire theory of economic is based on a number of basic axioms, which are fundamental models of economics, such as generic constraints of resource scarcity, unlimited demanding behaviors of consumers, profit-driven behaviors of producers, and the conservative behaviors of market systems [Slavin, 1988; Cannan, 1994; Brue, 2001; Wang, 2005d]. This subsection explains these basic axioms and fundamental models that are shared by both macro and micro economics.

12.2.1.1 Demand vs. Supply

Demand and supply are a pair of fundamental concepts of economics. They are also the foundation for engineering resources management and organization.

Definition 12.4 *Demand* is the required quantities for a product or service that consumers are willing and able to buy at a given range of prices.

Demands are the fundamental driving force of market systems and the predominant reason behind almost all economic phenomena. The market response to a demand is called supply.

Definition 12.5 *Supply* is the required quantities for a product or service that producers are willing and able to sell at a given range of prices.

Demands and supplies are the fundamental behaviors of dynamic market systems, which form the context of economics.

12.2.1.2 The Principle of Resource Scarcity

The most basic yet important principle of economics is the recognition of a pair of contradictive phenomena [Slavin, 1988], *resource scarcity* vs. *unlimited human demands*, in human activities and the society.

Lemma 12.1 The principle of *resource scarcity* states that the total resources at a given time $R_{\Sigma}(t)$, or the means of production represented by their values, such as land $V_{l}(t)$, building $V_{b}(t)$, materials $V_{m}(t)$, labor $V_{lb}(t)$, and capital $V_{c}(t)$, are constrained by a constant of nature $k(t)$, which is always inadequate to meet the ever growing total demands $D_{\Sigma}(t)$.

$$\begin{aligned} R_{\Sigma}(t) &= V_{l}(t) + V_{b}(t) + V_{m}(t) + V_{lb}(t) + V_{c}(t) \\ &= k(t) \\ &< D_{\Sigma}(t) \end{aligned} \tag{12.1}$$

The principle of resource scarcity forms a fundamental constraint to demands and supplies in the market system. The entire economic theory is oriented to the coordination and balancing of this pair of interacting phenomena. With this view, economics may be seen as a science that studies how the unlimited human demands may be met under the generic constraint of resource scarcity.

12.2.1.3 The Law of Market Conservation

Lemma 12.2 The *law of market conservation* states that the prices of goods or services in a market system behave conservatively and complementally to the quantities of demands and supplies, i.e.:

$$\begin{cases} \left.\begin{aligned} D\uparrow &\;\to \\ S\downarrow &\;\to \end{aligned}\right\rangle \to P\uparrow \\ \left.\begin{aligned} D\downarrow &\;\to \\ S\uparrow &\;\to \end{aligned}\right\rangle \to P\downarrow \end{cases} \tag{12.2}$$

The behaviors of prices responding to the changes of demands and supplies as stated in Lemma 12.2 can be illustrated in Fig. 12.1.

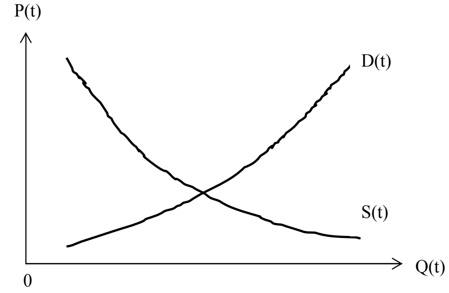


Figure 12.1 Behaviors of prices influenced by demands and supplies

12.2.1.4 The Law of Maximizing Profits

Lemma 12.3 The *ultimate objective of markets*, and of the producers and consumers in them, is to pursue the maximum profit P_{max}, or in other words, to maximize the revenues R_{max} and to minimize the costs C_{min} at the same time, i.e.:

$$P_{max}(t) = R_{max}(t) - C_{min}(t) \tag{12.3}$$

According to Lemma 12.3, the consumer and supplier behaviors in the market driven by the ultimate profit motivation are primarily influenced by the price as described below.

Lemma 12.4 The *law of maximizing profit* states that the demands and supplies of goods or services in a market system are driven by the tendency to maximize profits leveraged by the changes of prices, i.e.:

$$\begin{cases} P\uparrow \; \rightarrow \left\langle \begin{matrix} \rightarrow D\downarrow \\ \rightarrow S\uparrow \end{matrix} \right. \\ P\downarrow \; \rightarrow \left\langle \begin{matrix} \rightarrow D\uparrow \\ \rightarrow S\downarrow \end{matrix} \right. \end{cases} \tag{12.4}$$

The behaviors of demands of consumers and supplies of producers responding to the changes of price as stated in Lemma 12.4 can be illustrated in Fig. 12.2.

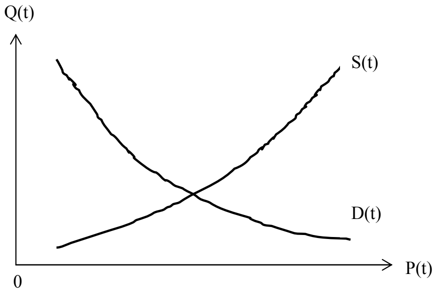


Figure 12.2 Behaviors of demands and supplies influenced by prices

12.2.2 ECONOMIC EQUILIBRIUM BETWEEN DEMANDS AND SUPPLIES

On the basis of Lemma 12.4, it can be seen that both demands and supplies are *price-driven* to meet the ultimate economic goals of consumers and producers.

Lemma 12.5 Price $P(t)$ is an important leverage in the market to autonomously adjust the equilibrium of demands and supplies, known as the *invisible hands,* according to Adam Smith (1776).

However, it should be emphasized that the market is a bi-directional interacting system. That is, the price not only influences demands and supplies (Lemma 12.4), but also is influenced by the demands and supplies (Lemma 12.2). Therefore, the entire mechanisms of a market system and the behaviors of the closed-circle interactions among demands, supplies, and prices are known as the *economic equilibrium*.

The equilibrium between demands and supplies in a given market at a given point of time can be illustrated by Fig. 12.3. Fig.12.3 provides a unified equilibrium model for explaining the equilibrium mechanism of prices as a result of interactions between demands and supplies in a market.

Definition 12.6 *Equilibrium* of demand and supply *e* is a point of quantity $Q_e(t)$ where the demand $D(t)$ equals to the supply $S(t)$, i.e.:

$$e = \{Q_e(t) \mid D(t) = S(t)\} \tag{12.5}$$

where the price at *e*, $P_e(t)$, is called the *equilibrium price*.

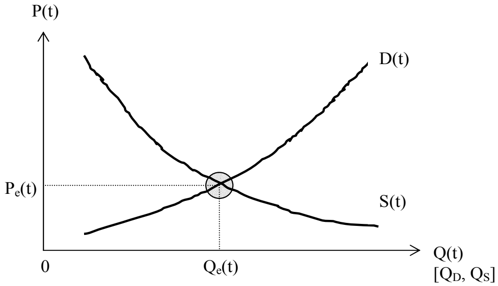


Figure 12.3 The equilibrium between demands *D*(*t*) and supplies *S*(*t*)

The equilibrium as describe by Eq. 12.5 and illustrated in Fig. 12.3 demonstrates that the total quantities of demands as outputs and supplies as inputs in a market system determine the equilibrium point and the equilibrium price. At the same time, the price also enforces feedbacks to influence the equilibrium quantities between the demands and supplies.

Corollary 12.1 The *equilibrium mechanism* interacting between the quantities of demands, supplies, and the prices of them in a market system is the *invisible hand*.

That is, Lemma 12.5 only reveals part of the truth about the mechanisms of economic equilibrium in a market system.

It is noteworthy that conventional economics textbooks provide an upside-down model to explain equilibrium where the curves of *D*(*t*) and *S*(*t*) as shown in Figs. 12.1 and 12.3 are confusingly interchanged [Sepulveda et al., 1984; Slavin, 1988; Frank, 1997; Park et al., 2001]. This convention has made the formal treatment of economic equilibrium very difficult.

The next subsection formally describes the mechanisms of economic equilibrium in a market system that provides a rigorous explanation of Adam Smith's *invisible hand* hypothesis.

12.2.3 THE BEHAVIORS OF MARKET SYSTEMS

According to Corollary 10.11, the functional condition of any self-organization system is the existence of the negative feedback mechanism that is proportional to the incremental or aggressive effects of the system. The market systems as a special type of self-organization systems obey the same law. That is, the adaptive equilibrium of market systems as a result of interactions between demands and supplies is rooted in the negative feedback mechanisms between their quantities and prices.

Definition 12.7 The *equilibrium model* of market systems is a negative feedback system, in which the increase or decrease of price in the market will result in a negated feedback, and so do the changes of quantities of demands and supplies on prices, which intend to resist the tendency of deviating from the current equilibrium.

Based on Definition 12.7 and Lemmas 12.2 and 12.4, the entire behaviors of market systems are constrained by the following theorem [Wang, 2005d].

The 44th Law of Software Engineering

Theorem 12.1 The *adaptive economic equilibrium* states that a market with autonomic interactions between demands *D* and supplies *S* is a self-regulated and self-adaptive system, where any change in demand, supply, or both will be autonomously adjusted via the leverage of price *P* to an equilibrium, i.e.:

$$\begin{bmatrix} \left\langle \begin{matrix} D\uparrow \rightarrow \\ S\downarrow \rightarrow \end{matrix} \right\rangle \rightarrow P\uparrow & \Rightarrow \left\langle \begin{matrix} D\downarrow \rightarrow \\ S\uparrow \rightarrow \end{matrix} \right\rangle \rightarrow P\downarrow \\ \left\langle \begin{matrix} D\downarrow \rightarrow \\ S\uparrow \rightarrow \end{matrix} \right\rangle \rightarrow P\downarrow & \Rightarrow \left\langle \begin{matrix} D\uparrow \rightarrow \\ S\downarrow \rightarrow \end{matrix} \right\rangle \rightarrow P\uparrow \end{bmatrix}$$

$$[\frac{\text{Market conservation}}{\text{Lemma 12.xx}}] + [\frac{\text{Maximizing profits}}{\text{Lemma 12.xx}}]$$

Theorem 12.1 indicates that Lemmas 12.2 and 12.4 are mutually reflexive. Each of them represents an aspect of the entire mechanisms of the market systems. The consequence of an economic equilibrium may be described by the following corollary.

Corollary 12.2 The *result of interactions* between dynamic demands and supplies, through the leverage of prices, results in an automatic stabilization of the market at a new equilibrium that is close to the current equilibrium price.

Theorem 12.1 and Corollary 12.2 are a rigorous enhancement [Wang, 2005d] of Adam Smith's hypothesis of the *invisible hand*. These laws will be explained in the following subsections with both simple and complex modes of economic equilibriums.

12.2.3.1 Simple Modes of Economic Equilibriums

There are four simple modes that may drive a market away from an equilibrium considered in conventional economics. They are demand increase *E*(*D*+), demand decrease *E*(*D*-), supply increase *E*(*S*+), and supply decrease *E*(*D*-). Theorem 12.1 can be applied to each of the above modes and situations as analyzed below.

Mode 1. *Demand Increase E*(*D*+)

The reactions of the equilibrium mechanism to an event of demand increase, *E*(*D*+), can be described by the following reactions:

1) Demand is increased $D\uparrow$ as an event.

2) Price *P* is increased due to $D\uparrow$ according to the *law of scarcity* (Lemma 12.1).

3) Parallel to Reaction 2, quantity of supply is increased $Q_S\uparrow$ according to the *law of maximizing profits* (Lemma 12.4).

4) Price *P* is decreased following $S\uparrow$ in Reaction 3 according to the *law of market conservation* (Lemma 12.2).

5) Price *P* is regulated to a newly established P'_e that is close to P_e due to synthetic result of the effect and negative feedback according to the *law of economic equilibrium* (Theorem 12.1).

The above chain of feedback reactions can be formally described by Eq. 12.6 and illustrated by Fig. 12.4.

$$E(D+) = D\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e \tag{12.6}$$

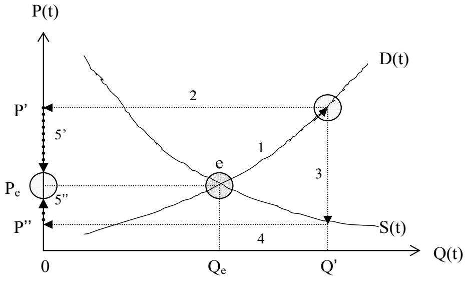


Figure 12.4 The equilibrium mechanism of Mode 1: *E*(*D*+)

On the basis of Fig. 12.4, the newly established equilibrium P'_e, and the increment of price ΔP can be predicated as follows.

The 44th Principle of Software Engineering

Theorem 12.2 The *predictability of new equilibrium* states that a *newly established equilibrium* on price P'_e is determined by the effect *P'* and feedback effect *P"* of the driving forces deviating from the current equilibrium, i.e.:

$$\begin{aligned} P'_e &= P'' + \frac{P'\text{-}P''}{2} \\ &= \frac{P' + P''}{2}, \ P' > P'_e \end{aligned} \tag{12.7}$$

and the increment of price caused by the shifting of equilibriums is:

$$\begin{aligned} \Delta P &= P'_e - P_e \\ &= \frac{P' + P''}{2} - P_e, \ P' > P'_e \end{aligned} \tag{12.8}$$

where ΔP may be positive or negative that represents a upward or downward shifting of the current equilibrium price, respectively.

Example 12.1 Assume that two shifted equilibriums of a market system are affected by demand increases resulting in the following effects (P_{e1}, P'_1, P''_1) = (20, 36, 10) and (P_{e2}, P'_2, P''_2) = (30, 40, 8). The newly established equilibriums can be predicated as:

$$\begin{aligned} P'_{e1} &= \frac{P'_1 + P''_1}{2} \\ &= \frac{36 + 10}{2} \\ &= \$23 \end{aligned}$$

and

$$\begin{aligned} P'_{e2} &= \frac{P'_2 \ + P''_2}{2} \\ &= \frac{40 + 8}{2} \\ &= \$24 \end{aligned}$$

The increments in the above situations are:

$$\begin{aligned} \Delta P_1 &= P'_{e1} - P_{e1} \\ &= 23 - 20 \\ &= \$3 \end{aligned}$$

and

$$\begin{aligned} \Delta P_2 &= P'_{e2} - P_{e2} \\ &= 24 - 30 \\ &= \text{-}\$6 \end{aligned}$$

It can be proven that Theorem 12.2 can be applied to any other mode of market equilibriums.

Mode 2. *Demand Decrease E(D-)*

The reactions of the equilibrium mechanism to an event of demand decrease, *E*(*D*-), are formally described as follows:

$$E(D\text{-}) = D \downarrow \ \rightarrow \left\langle \begin{array}{ll} \rightarrow P \downarrow & \qquad \rightarrow \\ \rightarrow S \downarrow & \rightarrow P \uparrow \rightarrow \end{array} \right\rangle \Rightarrow P'_e \tag{12.9}$$

The chain of feedback reactions as described in Eq. 12.9 can be illustrated as shown in Fig. 12.5.

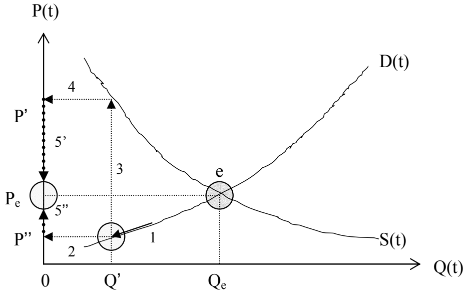


Figure 12.5 The equilibrium mechanism of Mode 1: *E*(*D*-)

Mode 3. *Supply Increase E(S+)*

The reactions of the equilibrium mechanism to an event of supply increase, *E*(*S*+), are formally described as follows:

$$E(S+) = S \uparrow \ \rightarrow \left\langle \begin{matrix} \rightarrow P \downarrow & & \rightarrow \\ \rightarrow S \uparrow & \rightarrow P \uparrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e \tag{12.10}$$

The chain of feedback reactions as described in Eq. 12.10 can be illustrated as shown in Fig. 12.6.

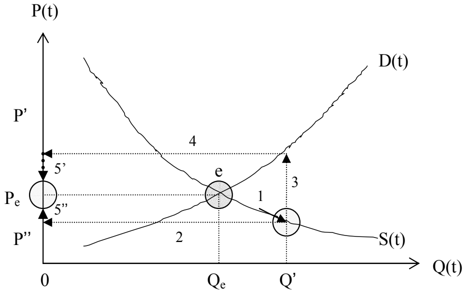


Figure 12.6 The equilibrium mechanism of Mode 1: *E*(*S*+)

Mode 4. *Supply Decrease E(S-)*

The reactions of the equilibrium mechanism to an event of supply decrease, *E*(*S*-), can be formally described as follows:

$$E(S\text{-}) = S \downarrow \ \rightarrow \left\langle \begin{matrix} \rightarrow P \uparrow & & \rightarrow \\ \rightarrow S \downarrow & \rightarrow P \downarrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e \tag{12.11}$$

The chain of feedback reactions as described in Eq. 12.11 can be illustrated as shown in Fig. 12.7.

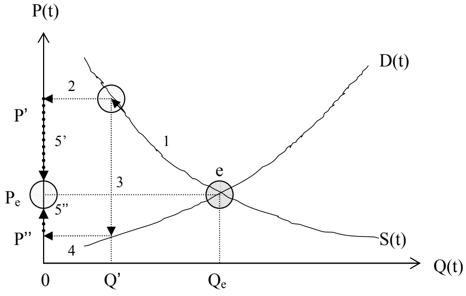


Figure 12.7 The equilibrium mechanism of Mode 1: *E*(*S*-)

12.2.3.2 Complex Modes of Economic Equilibriums

Observing the driving causes of market systems, there are four additional complex modes based on the above simple modes, which could not be accurately dealt with conventional economics theories. They are compound demand/supply increases *E*(*D*+, *S*+), compound demand increase and supply decrease *E*(*D*+, *S*-), compound demand decrease and supply increase *E*(*D*-, *S*+), and compound demand decrease and supply decrease *E*(*D*-, *S*-).

Theorem 12.1 can still be applied in each of the complex modes and situations as follows.

Mode 5. *Compound Demand/Supply Increases E*(*D*+, *S*+)

The reactions of the equilibrium mechanism to a compound event of demand/supply increases, *E*(*D*+, *S*+), are formally described by Eq. 12.12. The chain of feedback reactions described in Eq. 12.12 can be illustrated by combining Figs. 12.4 and 12.6.

$$E(D+, S+) = \left\langle \begin{array}{ll} D\uparrow & \rightarrow \left\langle \begin{array}{lll} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \\ S\uparrow & \rightarrow \left\langle \begin{array}{lll} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow D\uparrow & \rightarrow P\uparrow & \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e \qquad (12.12)$$

Mode 6. *Compound Demand Increase/Supply Decrease E*(*D*+, *S*-)

The reactions of the equilibrium mechanism to a compound event of demand increase/supply decrease, *E*(*D*+, *S*-), are formally described by Eq. 12.13. The chain of feedback reactions described in Eq. 12.13 can be illustrated by combining Figs. 12.4 and 12.7.

$$E(D+, S-) = \left\langle \begin{array}{ll} D\uparrow & \rightarrow \left\langle \begin{array}{lll} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \\ S\downarrow & \rightarrow \left\langle \begin{array}{lll} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow D\downarrow & \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e \qquad (12.13)$$

Mode 7. *Compound Demand Decrease/Supply Increases E*(*D*-, *S*+)

The reactions of the equilibrium mechanism to a compound event of demand decrease/supply increases, *E*(*D*-, *S*+), are formally described by Eq. 12.14. The chain of feedback reactions described in Eq. 12.14 can be illustrated by combining Figs. 12.5 and 12.6.

$$E(D\text{-}, S\text{+}) = \left\langle \begin{array}{l} D \downarrow \; \to \left\langle \begin{array}{lll} \to P \downarrow & & \to \\ \to S \downarrow & \to P \uparrow & \to \end{array} \right\rangle \\ S \uparrow \; \to \left\langle \begin{array}{lll} \to P \downarrow & & \to \\ \to D \uparrow & \to P \uparrow & \to \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e \tag{12.14}$$

Mode 8. *Compound Demand/Supply Decreases E*(*D*-, *S*-).

The reactions of the equilibrium mechanism to a compound event of demand/supply decreases, *E*(*D*-, *S*-), are formally described by Eq. 12.15. The chain of feedback reactions described in Eq. 12.15 can be illustrated by combining Figs. 12.5 and 12.7.

$$E(D\text{-}, S\text{-}) = \left\langle \begin{array}{l} D \downarrow \; \to \left\langle \begin{array}{lll} \to P \downarrow & & \to \\ \to S \downarrow & \to P \uparrow & \to \end{array} \right\rangle \\ S \downarrow \; \to \left\langle \begin{array}{lll} \to P \uparrow & & \to \\ \to D \downarrow & \to P \downarrow & \to \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e \tag{12.15}$$

12.2.3.3 The Adaptive Equilibrium Mechanisms of Market Systems

The eight modes of equilibrium mechanisms described so far are summarized in Table 12.1.

Theorems 12.1 and 12.2 as well as related mathematical models derived in this section are actually a formal proof of Adam Smith's hypothesis of *invisible hand* proposed in 1776 [Smith, 1776] with the enhancement in Theorem 12.2.

Corollary 12.3 The *adaptive equilibrium mechanism* of market systems described in Theorem 12.1 and Modes 1 through 8 is the *invisible hand*, which self-regulates and self-organizes the equilibrium of quantities and prices affected by the interactions between demands and supplies.

Corollary 12.4 Equilibrium market is a *conservative system*. Once an equilibrium is established in a market, the price may gradually wave around and slowly shifting from P_e, but may not be increased or decreased abruptly and dramatically.

Table 12.1
Adaptive Equilibrium Behaviors of Market Systems

Mode		Event		Chain of Feedback Reactions	Illustration
No.	Symbol	D	S		
1	*E*(*D*+)	↑		$D\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.4
2	*E*(*D*-)	↓		$D\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.5
3	*E*(*S*+)		↑	$S\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.6
4	*E*(*S*-)		↓	$S\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.7
5	*E*(*D*+, S+)	↑	↑	$\left\langle \begin{matrix} D\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \\ S\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.4 & Fig. 12.6
6	*E*(*D*+, S-)	↑	↓	$\left\langle \begin{matrix} D\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \\ S\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.4 & Fig. 12.7
7	*E*(*D*-, S+)	↓	↑	$\left\langle \begin{matrix} D\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \\ S\uparrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.5 & Fig. 12.6
8	*E*(*D*-, S-)	↓	↓	$\left\langle \begin{matrix} D\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\uparrow & \rightarrow \end{matrix} \right\rangle \\ S\downarrow \;\rightarrow \left\langle \begin{matrix} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\downarrow & \rightarrow P\downarrow & \rightarrow \end{matrix} \right\rangle \end{matrix} \right\rangle \Rightarrow P'_e$	Fig. 12.5 & Fig. 12.7

12.3 Economic Models

To explain the relations among a great variety of economic phenomena and their behaviors, a set of economic models such as the production model, cost model, and market model will be studied in this section.

12.3.1 PRODUCTION MODELS

Production models study the forms of production systems and their efficiencies between the input and output. The most important production model in economics is productivity.

Definition 12.8 The *productivity*, or the *average product*, $\overline{P}$, is a ratio between the total output O and the variable input or labor I_v, i.e.:

$$\overline{P} = \frac{O}{I_v} \tag{12.16}$$

Productivity is a macro measure of an economic system. In order to find the efficiency of economic input of a production system per unit of labor, the marginal product at the micro level is introduced below.

Definition 12.9 The *marginal product* $\overline{P}_\Delta$ is a ratio between the incremental output and the incremental input, i.e.:

$$\overline{P}_\Delta = \frac{\Delta O}{\Delta I_v} \tag{12.17}$$

Specialization efficiency was found in the industrial revolutions that productivity or marginal product can be increased by adding specialized labor in the incremental input in a certain range. However, overhead of specialization may diminish the return of variable input, when ΔI_v is large enough. This observation leads to the following lemma.

Lemma 12.6 *Law of diminishing returns* **s**tates that specialization efficiency is over turned by overhead of using more variable input.

12.3.2 COST MODELS

Costs of production systems can be classified as the *fixed* and *variable costs*. The former such as those of buildings and machines are invariant with the output or scale of production even though if there is no output the same fixed costs still exist. The latter are proportional to output in a production system such as those of materials and labor. Therefore, labor and its rational organization are the most important and value-adding essence in production.

Definition 12.10 *Total cost* of a production system C is the sum of fixed cost c_f and variable cost c_v, i.e.:

$$C = c_f + c_v \tag{12.18}$$

The total cost is an absolute value in production. In economic analyses, the relative cost against output, or the average cost, is more meaningful in cost analyses.

Definition 12.11 *The average cost* in production $\overline{C}$ is the unit cost per product, i.e.:

$$\begin{aligned}\overline{C} &= \frac{C}{O}\\ &= \frac{c_f + c_v}{O}\\ &= \frac{c_f + c_v}{\overline{P} \bullet I_v}\end{aligned} \tag{12.19}$$

Definition 12.12 *Marginal cost* $\overline{C}_\Delta$ is the ratio between the incremental total cost and incremental output, i.e.:

$$\begin{aligned}\overline{C}_\Delta &= \frac{\Delta C}{\Delta O}\\ &= \frac{c_f + \Delta c_v}{\overline{P}_\Delta \bullet \Delta I_v}\end{aligned} \tag{12.20}$$

Eqs. 12.19 and 12.20 indicate that the average cost or marginal cost may be reduced by increasing productivity or decreasing variable costs on labor or materials. Eq. 12.20 also indicates that the increase of ΔO will result in the increases of Δc_v. The tradeoffs between ΔO and Δc_v, therefore, form the economical scale problem.

Definition 12.13 The *economical scale* of production is the maximum output that yields the minimum average cost under a certain productivity, i.e.:

$$\overline{C}_{\min} = \frac{C}{O_{\max}} \tag{12.21}$$

A typical relationship between $O_{\max}$ and $\overline{C}_{\min}$ can be illustrated in Fig. 12.8.

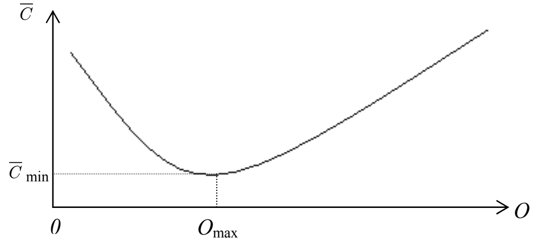


Figure 12.8 The economical scale of production

12.3.3 MARKET MODELS

Definition 12.14 The *market* is an economic domain in which buyers and sellers exchange commodity and services.

Markets in economics can be classified into a spectrum of categories such as those of perfect competitive, monopolistically competitive (with product differentiation), oligopoly, and monopoly. The perfect competition market and oligopoly are the two extreme forms of markets, where the former is the most healthy and efficient market.

Definition 12.15 A *perfect competitive market* is a free-entry market where many sellers supply identical products or services so that none of them may dominantly influence the market prices.

In the perfect competitive market, a supplier is forced by competitors to operate at maximum efficiency rather than to manipulate a higher price in order to make profits. Therefore, the perfect competitive market is the most consumer-friendly market.

Definition 12.16 A *monopolistic market* is a market where only a sole supplier provides a good or service without any close substitutes.

Although the demand is decided by buyers, a monopoly in the market may be formed based on one of the following conditions: a) A controller of an essential resource; b) A holder of a government franchise; c) A creator of a new market, or a pioneer getting a market first; d) A founder of de facto industrial standard; and e) An inventor and entrepreneur of a widely applicable product or service.

Real-world markets are operating in between the perfect competitive and monopolistic market modes. Usually, the large-scale and global industries are oligopolistic, local utility industries are monopolistic, and the remainder is perfect competitive or semi-competitive.

The markets may also be classified as surplus and shortage markets, which can be illustrated by the interactions of demands and supplies over time as shown in Fig. 12.9, where:

$$\begin{array}{lll} \text{a) A surplus market:} & t_{i'} < t_i \Rightarrow S'(t_i) > D(t_i) & \\ \text{b) A shortage market:} & t_{i''} > t_i \Rightarrow S''(t_i) < D(t_i) & (12.22) \end{array}$$

The former is also known as the *buyer-market* and the latter the *seller-market* dependent on whether the demand occurs after the supply (*S'*(*t*)) or before it (*S''*(*t*)).

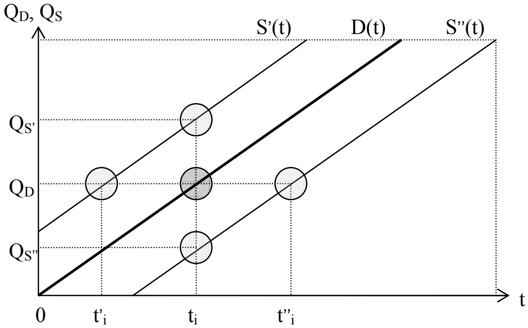


Figure 12.9 Surplus vs. shortage markets

12.4 Dynamic Values of Money and Assets

A basic concept of economics is that the values of physical assets and their denoted representation in terms of money are a relative quantity. Both of their values change over time, or more rigorously, their value is a function of time and the interest rate. This section describes the dynamics of money and assets. The techniques for determining the values of various cash flows in any given point of time are presented.

12.4.1 DYNAMICS OF MONEY

Since money has been recognized as a generic representation or measure of the value of any goods, services, and assets, the study of its dynamics is to analyze the factors and mechanisms that influence the value of money.

Definition 12.17 The *dynamic value of money* or an asset, $V(t)$, is its present worth P projected at a given point of time t for a given average or predicated interest rate i during $[0, t]$, i.e.:

$$V(t) = f(P, i, t) \tag{12.23}$$

For totally n interest calculation periods, the *simple interest* I that is earned only on principal P during each interest period is:

$$I(n) = (P \bullet i) \bullet n \tag{12.24}$$

where the dimension of time is simplified into a serial discrete interest periods n.

The value at end of the *nth* interest period, $V(n)$, or the future value $F(n)$, for a given average interest i can be determined as follows:

$$\begin{aligned} V(n) &= F(n) \\ &= f(P, i, n) \\ &= P + I(n) \\ &= P + (P \bullet i \bullet n) \\ &= P\,(1 + i \bullet n) \end{aligned} \tag{12.25}$$

A more advanced interest calculation method is known as the *compound interest* that is recursive accumulation of the interest based on each end of the n given periods as the future value $F(n)$, i.e.:

$$F(n) = P(1+i)^{n} \tag{12.26}$$

An inverse function of Eq. 12.26 determines the present value P of a future value $F(n)$ for given periods n and interest i, i.e.:

$$P = F(n)\frac{1}{(1+i)^n} \tag{12.27}$$

12.4.2 DYNAMICS OF ASSET'S VALUES

As that of money discussed in the preceding subsection, values of assets possess a dynamic property too.

Definition 12.18 For a given asset, the continuous decreasing of value over time is known as *depreciation*.

The depreciation of assets can be classified as: a) *Physical depreciation* that refers to the reduction in asset's capacity to perform its intended service due to physical impairment; b) *Functional depreciation* that refers to obsolescence; c) *Economic depreciation* that refers to the total values lost during the life span of an asset; and d) *Accounting depreciation* that refers to a systematic allocation of the initial cost of an asset in parts over time.

The physical, functional, and economic depreciations are equivalent to the concept of system dissimilation discussed in Section 10.5.7. The accounting depreciation may be used for booking investment costs or for tax purposes.

Definition 12.19 Assume an asset provides an equal amount of utility or service in each year of its life-span *n*, the *linear depreciation* of the asset in each year *D* is:

$$D = \frac{P - S}{n} \tag{12.28}$$

where *P* is the *initial value* of the asset, and *S* the *salvage value* by the year end of *n*.

Therefore, the real value of the asset in year *k*, $V_a(k)$, $0 \le k \le n$, can be determined as:

$$\begin{aligned} V_a(k) &= P - kD \\ &= P - k\frac{P - S}{n} \\ &= \frac{P(n-k) + kS}{n} \end{aligned} \tag{12.29}$$

where, particularly, $V_a(0) = P$ and $V_a(n) = S$.

There are a number of nonlinear depreciation methods that enables a faster early depreciation by inverse exponential models, so that the costs of initial investment can be distributed or recovered in the early phase of projects. Details may be referred to [Park et al., 2001; Sepulveda et al., 1984].

12.4.3 CUMULATIVE VALUES OF CASH FLOWS

Cumulated values of a series of cash flows can be derived as a sum of individual payments at the same point of time [Park et al., 2001; Sepulveda et al., 1984], such as at present or at the end of *n* period in the future .

12.4.3.1 The Uniform Payment Series

Definition 12.20 The *uniform payment series* is a series of identical payments *A* at the end of each period by a fixed frequency.

The *cumulated present value* $P_\Sigma(A)$ of a uniform series payments *A* is given by Eq. 12.30 below:

$$P_\Sigma(A) = A\,\frac{(1+i)^n\text{-}1}{i(1+i)^n} \qquad (12.30)$$

Inversely, the equivalent value of uniform payment *A* at present is:

$$A = P_\Sigma(A)\,\frac{i(1+i)^n}{(1+i)^n\text{-}1} \qquad (12.31)$$

where *A* is called the *capital recovery factor* for denoting the periodical return of an initial investment $P_\Sigma(A)$.

The *cumulated future value* $F_\Sigma(A)$ of a uniform series *A* is given by Eq. 12.32 below:

$$F_\Sigma(A) = A\,\frac{(1+i)^n\text{-}1}{i} \qquad (12.32)$$

Inversely, the equivalent value of uniform payment *A* in a given year in the future is:

$$A = F_\Sigma(A)\,\frac{i}{(1+i)^n\text{-}1} \qquad (12.33)$$

where A is called the *sinking fund factor* for denoting the periodical payment for a given accumulated fund in the future $F_\Sigma(A)$.

It is noteworthy that the cumulative present and future values are still obeying the relationship as described in Eq. 12.34, i.e.:

$$F_\Sigma\,(A) = P_\Sigma(A)\,(1+i)^n \qquad (12.34)$$

Eq. 12.34 can be proved by replacing $P_\Sigma(A)$ and $F_\Sigma(A)$ in the above equation by Eq. 12.30 and Eq. 12.32, respectively.

12.4.3.2 The Linear Gradient Payment Series

Definition 12.21 A *linear gradient payment series* is a series of linearly increased payments G by a fixed frequency.

The linear gradient series G is illustrated in Fig. 12.10.

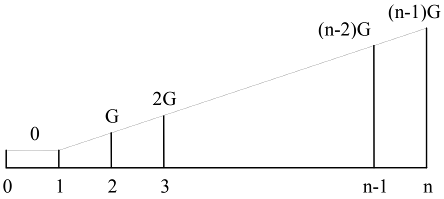


Figure 12.10 Linear gradient series

The cumulated present value $P_\Sigma(G)$ of a linear gradient series G is given below:

$$P_\Sigma(G) = G\,\frac{(1+i)^n - i \bullet n\ \text{-}1}{i^2\,(1+i)^n} \qquad (12.35)$$

where G is the increment factor and G can be positive or negative to implement a linear gradient increase or decrease in the series.

12.3.3.3 The Geometric Gradient Payment Series

Definition 12.22 A *geometric gradient payment series* is a series of nonlinearly increased payments g by a fixed frequency.

The geometric gradient series g is illustrated in Fig. 12.11.

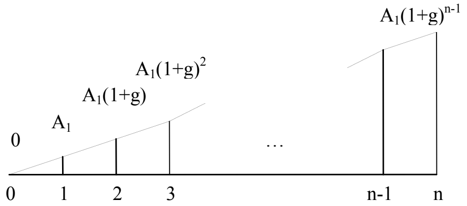


Figure 12.11 Geometric gradient series

The cumulated present value $P_\Sigma(g)$ of a geometric gradient series g is given below:

$$P_\Sigma(g) = A_1 \frac{1\text{-}(1+g)^n(1+i)^{\text{-}n}}{i\text{-}g} \qquad (12.36)$$

where g is the growth rate and g can be positive or negative to implement a geometric gradient increase or decrease in the series.

In a more generic situation, the cash flows may be a composite series formed by the combination of the primitive series as discussed in the previous subsections. In this case, the cumulated value can be calculated as the sum of individual component series. Further discussions may be referred to [Park et al., 2001; Sepulveda et al., 1984].

12.5 Economic Analyses

This section develops a set of algorithms for dynamic cost and investment analysis that can be used in economic analyses for software engineering projects. The algorithms provide numerical solutions for values of cost and investment in present (P) and future (F), cumulative present value (P_Σ) and future value (R_Σ), return-period (n) and return-rate (ρ). By applying these algorithms, complicated mathematical problems in dynamic cost and investment estimation in software engineering can be solved easily. The algorithms are useful not only for project managers to plan and analyze software development costs, but also for customers to estimate investment benefit and risk of software projects.

12.5.1 PROJECT COSTS ANALYSES

The concepts of the dynamic values of money and assets have been introduced in Section 12.4. Fundamental expressions for comparing the values of cost/investment over time are provided. This subsection demonstrates applications of the dynamic value theory in project economic analyses.

Example 12.2 An engineering project is predicted to yield different cash flows as given in Table 12.2. Assuming the initial investment *P* = \$100,000, and interest rate *i* = 10%, analyze the cumulated payback value P_Σ of each cash flow.

Table 12.2
Cash Flows of a Project

End of year	**0**	**1**	**2**	**3**
Cash flow 1 (Random, k\$)	-100	30	60	50
Cash flow 2 (Uniform, k\$)	-100	46	46	46
Cash flow 3 (Linear gradient, k\$)	-100	0	80	160
Cash flow 4 (Geometric gradient, k\$)	-100	25	25 • 1.5	$25 \bullet 1.5^2$

a) **Cash Flow 1:** According to Eq.12.27, P_Σ is determined as follows:

$$\begin{aligned} P_\Sigma &= P + F(1)\frac{1}{(1+i)^1} + F(2)\frac{1}{(1+i)^2} + F(3)\frac{1}{(1+i)^3} \\ &= -100 + 30\text{k} \bullet 0.9091 + 60\text{k} \bullet 0.8264 + 50\text{k} \bullet 0.7513 \\ &= -100 + 27{,}273 + 49{,}584 + 37{,}566 \\ &= \$14{,}423 \end{aligned}$$

b) **Cash Flow 2:** According to Eq.12.34, P_Σ(A) is determined as follows:

$$\begin{aligned} P_\Sigma(A) &= P + A\,\frac{(1+i)^n\text{-}1}{i(1+i)^n} \\ &= -100 + 46{,}000 \bullet \frac{1.1^3\text{-}1}{0.1 \bullet 1.1^3} \\ &= -100 + 46{,}000 \bullet 2.4869 \\ &= \$14{,}397 \end{aligned}$$

c) **Cash Flow 3:** According to Eq.12.35, P_Σ(G) is determined as follows:

$$\begin{aligned}P_\Sigma(G) &= P + G\frac{(1+i)^n \text{-} \ i \bullet n \text{ -}1}{i^2(1+i)^n}\\ &= -100 + 80{,}000 \bullet \frac{1.1^3 \text{-} \ 0.1 \bullet 3 \text{ -}1}{0.1^2 \ \bullet 1.1^3}\\ &= -100 + 186{,}320\\ &= \$86{,}320\end{aligned}$$

Another way to calculate P_Σ(G) for Cash Flow 3 is using Eq. 12.27 as that for Cash Flow 1, and the same result will be yielded as shown below. However, when the series is very long, the above solution is more efficient.

$$\begin{aligned}P_\Sigma &= P + F(1)\frac{1}{(1+i)^1} + F(2)\frac{1}{(1+i)^2} + F(3)\frac{1}{(1+i)^3}\\ &= -100 + 0 + 80{,}000 \bullet 0.8264 + 160{,}000 \bullet 0.7513\\ &= -100 + 186{,}320\\ &= \$86{,}320\end{aligned}$$

d) **Cash Flow 4:** According to Eq.12.36, P_Σ(g) is determined as follows:

$$\begin{aligned}P_\Sigma(g) &= P + A_1\frac{1\text{-}(1{+}g)^n(1+i)^{\text{-}n}}{i\text{-}g}\\ &= -\ 100 + 25{,}000 \bullet \frac{1\text{-}(1{+}1.5)^3 \ \bullet 1.1^{\text{-}3}}{0.1\text{-}1.5}\\ &= -100 + 191{,}773\\ &= \$91.773\end{aligned}$$

12.5.2 PROJECT BENEFIT-COST ANALYSES

Analyses of benefit-cost ratios are usually based on the preset values *P*. However, when project life spans are different, the analysis can be based on annual values *A*.

Definition 12.23 The *total benefit* of a project *B* is the sum of its benefits B_k of year *k*, $0 \le k \le n$, in the view of its present values, i.e.:

$$B = \sum_{k=0}^{n} \frac{B_k}{(1+i)^k} \tag{12.37}$$

where *i* is the sponsor's interest or discount rate.

Definition 12.24 The *total costs* of a project *C* is the sum of its costs C_k of year *k*, $0 \le k \le n$, in the view of its present values, i.e.:

$$\begin{aligned} \mathrm{C} &= \sum_{k=0}^{n} \frac{C_k}{(1+i)^k} \\ &= C_0 + C' \end{aligned} \tag{12.38}$$

where C_k includes all capital expenditures C_0 and annual operating costs C′.

When both the total benefit and cost are known, the benefit-cost ratio of a project can be determined as follows.

Definition 12.25 *Benefit-cost ratio BC* of a project is a ratio between the total benefit *B* and the total cost *C*, i.e.:

$$\begin{aligned} \mathrm{BC} &= \frac{B}{C} \\ &= \frac{B}{C_0 + C'} \end{aligned} \tag{12.39}$$

A variation of *BC* known as the *net benefit-cost ratio B′C* is defined as below:

$$\begin{aligned} \mathrm{B'C} &= \frac{B - C'}{C_0} \\ &= \frac{B'}{C_0} \end{aligned} \tag{12.40}$$

where $B' = B - C'$ is the net benefit.

Definition 12.26 The *economic evaluation criterion* to accept a project is that its benefit-cost ratio is larger than one, i.e.:

$$BC > 1 \tag{12.41}$$

or

$$B'C > 1 \tag{12.42}$$

Eqs. 12.41 and 12.42 imply B > C or $B' > C_0$, respectively.

According to Definition 12.26, an evaluation of BC or $B'C$ for a given project may result in three outcomes as follows:

$$BC \begin{cases} > 1, \text{ a desirable project} \\ = 1, \text{ a risky project} \\ < 1, \text{ an unacceptable project} \end{cases} \tag{12.43}$$

12.5.3 PROJECT PAYBACK PERIOD ANALYSES

Definition 12.27 The *payback period* ρ of a project is the expected point of time n at which the initial investment P will be recovered by the revenues of the project P_Σ for a given interest rate i, i.e.:

$$\rho = \{n \mid P = P_\Sigma(n)\} \tag{12.44}$$

The accurate solution of Eq. 12.44 can be derived by constructing a function $f(n)$ as follows:

$$\begin{aligned} f(n) &= P - P_\Sigma(n) \\ &= P - A\,\frac{(1+i)^n - 1}{i(1+i)^n} \\ &= 0 \end{aligned} \tag{12.45}$$

where P_Σ is assumed as a uniform series of annual payback. In general, P_Σ can be a cumulated present value of any kind of cash flows.

Because Eq. 12.45 needs to be solved by a numerical algorithm, a simple estimation of payback period in practical engineering economic analysis may be calculated by linear interpolation as follows:

$$\begin{aligned} \rho &= \{n \mid P = P_\Sigma(n)\} \\ &\approx \lfloor n \rfloor + \frac{P - P_\Sigma(\lfloor n \rfloor)}{P_\Sigma(\lceil n \rceil) - P_\Sigma(\lfloor n \rfloor)} \end{aligned} \tag{12.46}$$

where $\lfloor n \rfloor$ and $\lfloor n \rfloor$ are the floor and ceiling of the turning point in a year n, respectively, where the cumulated payback $P_\Sigma(\lfloor n \rfloor)$ before n is less than P, but the following year will yield a $P_\Sigma(\lceil n \rceil)$ greater than P.

Example 12.3 The payback periods of Cash Flows 1 and 3 as given in Example 12.2 and Table 12.2 can be calculated as follows:

(a) **Cash Flow 1**:

$$\begin{aligned}
\rho_1 &= \{n \mid P = P_\Sigma(n)\} \\
&\approx \lfloor n \rfloor + \frac{P - P_\Sigma(\lfloor n \rfloor)}{P_\Sigma(\lceil n \rceil) - P_\Sigma(\lfloor n \rfloor)} \\
&= 2 + \frac{100,000 - 76,875}{114,423 - 76,875} \\
&= 2 + \frac{23,125}{37,548} \\
&= 2 + 0.62 \\
&= 2.62 \text{ [year]}
\end{aligned}$$

(b) **Cash Flow 3**:

$$\begin{aligned}
\rho_2 &\approx \lfloor n \rfloor + \frac{P - P_\Sigma(\lfloor n \rfloor)}{P_\Sigma(\lceil n \rceil) - P_\Sigma(\lfloor n \rfloor)} \\
&= 2 + \frac{100,000 - 66,112}{186,320 - 66,112} \\
&= 2 + \frac{33,888}{120,208} \\
&= 2.28 \text{ [year]}
\end{aligned}$$

12.5.4 PROJECT RATE OF RETURN ANALYSES

In engineering economic analysis, γ is a useful indication of the speed of payback.

Definition 12.28 The *rate of return* γ of a project is the equivalent interest rate yield by a cash flow P_Σ for recovering the initial investment P for a given period n, i.e.:

$$\gamma = \{i \mid P = P_\Sigma\} \tag{12.47}$$

The accurate solution for Eq. 12.47 can be derived by constructing a function $f(i)$ as follows:

$$\begin{aligned}
f(i) &= P - P_\Sigma \\
&= P - A\frac{(1+i)^n - 1}{i(1+i)^n} \\
&= 0
\end{aligned} \tag{12.48}$$

where P_Σ is assumed as a uniform series of annual payback. In general, P_Σ can be a cumulated present value of any kind of cash flows.

Because Eq. 12.48 needs to be solved by a numerical algorithm, in practical engineering economic analysis, the rate of return γ can be approximately estimated as the inverse of ρ, i.e.:

$$\gamma = \frac{1}{\rho} \bullet 100\% \tag{12.49}$$

The rate of return is usually compared with the bank interest i in economic analyses. The evaluation criteria of γ can be set as given below:

$$\gamma \begin{cases} > i, \text{ a desirable project} \\ = i, \text{ a risky project} \\ < i, \text{ an unacceptable project} \end{cases} \tag{12.50}$$

Example 12.4 Using the same data as given and/or derived in Examples 12.2 and 12.3, determine the rates of return for Cash Flows 1 and 3.

(a) **Cash Flow 1**: As obtained in Example 12.3, ρ_1 = 2.62 year. According to Eq. 12.49 the rate of return γ_1 can be determined as follows:

$$\begin{aligned} \gamma_1 &= \frac{1}{\rho_1} \bullet 100\% \\ &= \frac{1}{2.62} \bullet 100\% \\ &= 38.17\% \end{aligned}$$

(b) **Cash Flow 3**: The rate of return γ_3 for ρ_3 = 2.28 year is as follows:

$$\begin{aligned} \gamma_3 &= \frac{1}{\rho_3} \bullet 100\% \\ &= \frac{1}{2.28} \bullet 100\% \\ &= 43.86\% \end{aligned}$$

A case study on economic analyses of a software engineering project will be provided in Section 12.6.4.

12.6 Software Engineering Economics

Software engineering economics is a branch of applied microeconomics that studies how resources are used to produce software systems and services and how optimistic decisions may be made for software engineering projects. This section describes elements of software costs, economic analyses, problem solving methods, and their applications in a variety of software engineering project decisions.

12.6.1 ELEMENTS OF SOFTWARE ENGINEERING COSTS

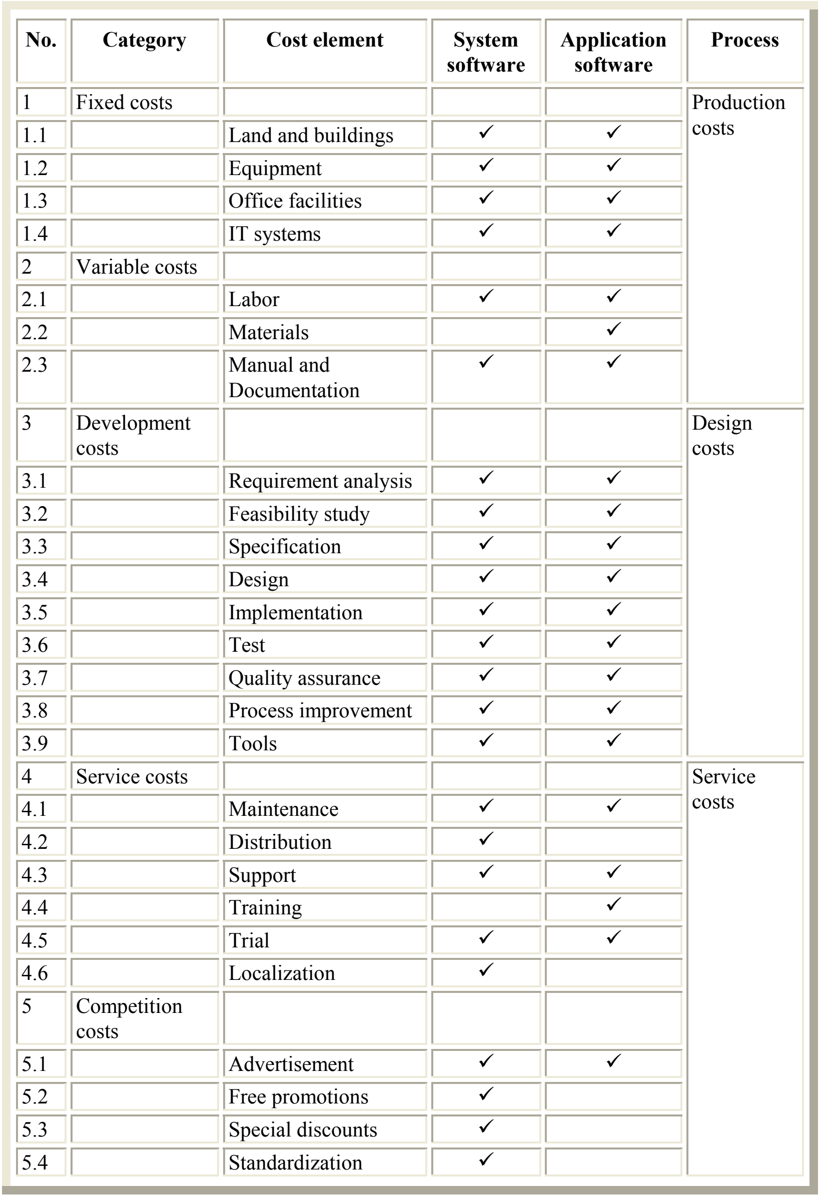
Prior to developing the cost and revenue models of software engineering, the taxonomy of software costs is presented. Then, the cost elements of system and application software are comparatively analyzed.

12.6.1.1 Analysis of Software Engineering Costs

The costs of software engineering projects can be classified into the categories of fix cost, variable cost, development cost, service cost, and competition cost as shown in Table 12.3. Mapping these cost elements into software engineering processes, they can be described as the design, production, and service costs.

The conventional cost models in economics consider only the *fixed costs* and *variable costs* as discussed in Section 12.3.2, which are oriented to the manufacturing industry characterized by mass production. For software engineering projects in the software industry, more cost categories need to be explored, such as the *development costs* and *service costs*. Also, the differences between the cost models of system and application software need to be distinguished, where system software are operating systems and fundamental system tools such as language compliers, database management systems, and network/communication software; while application software are those of user developed built on top of the system software.

Table 12.3
Elements of Software Costs

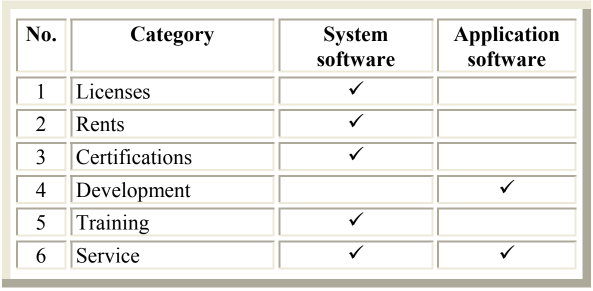


No.	Category	Cost element	System software	Application software	Process
1	Fixed costs				Production costs
1.1		Land and buildings	✓	✓	
1.2		Equipment	✓	✓	
1.3		Office facilities	✓	✓	
1.4		IT systems	✓	✓	
2	Variable costs				
2.1		Labor	✓	✓	
2.2		Materials		✓	
2.3		Manual and Documentation	✓	✓	
3	Development costs				Design costs
3.1		Requirement analysis	✓	✓	
3.2		Feasibility study	✓	✓	
3.3		Specification	✓	✓	
3.4		Design	✓	✓	
3.5		Implementation	✓	✓	
3.6		Test	✓	✓	
3.7		Quality assurance	✓	✓	
3.8		Process improvement	✓	✓	
3.9		Tools	✓	✓	
4	Service costs				Service costs
4.1		Maintenance	✓	✓	
4.2		Distribution	✓		
4.3		Support	✓	✓	
4.4		Training		✓	
4.5		Trial	✓	✓	
4.6		Localization	✓		
5	Competition costs				
5.1		Advertisement	✓	✓	
5.2		Free promotions	✓		
5.3		Special discounts	✓		
5.4		Standardization	✓		

12.6.1.2 Analysis of Software Engineering Revenues

The elements of software revenues in software engineering can be analyzed as shown in Table 12.4. It is noteworthy that, although the cost models of system and application software are quite similar, their revenue models are fairly different.

Table 12.4
Elements of Software Revenues



No.	Category	System software	Application software
1	Licenses	✓	
2	Rents	✓	
3	Certifications	✓	
4	Development		✓
5	Training	✓	
6	Service	✓	✓

According to the revenue models of software, it is obvious that system software may create much higher revenues than those of application software systems.

12.6.2 SOFTWARE ENGINEERING PROJECT COSTS ESTIMATION USING FEMSEC

The cost of a software engineering project is usually perceived as a linear function of the size of a given project. However, according to the coordinative work organization theory as developed in Section 8.5, software project cost is more directly related to the expected workload of projects, which is dominated by the property of interpersonal coordination rate required for the project. Further, the allocations of labor and time for a project cannot be carried out freely, but are constrained by certain laws as stated in Theorems 8.4 and 8.7. The workload-based approach to software engineering project cost determination will be formalized in this subsection by the Formal Economic Model of Software Engineering Cost (FEMSEC) [Wang, 2007d], and will be compared with the COCOMO approach in Section 12.6.3.

12.6.2.1 The FEMSEC Model of Software Engineering Costs

It recognized that the cost of a software engineering project is not simply a linear function of the size of the project rather than a complicated function related to the expected workload, the form of labor allocation, and whether the shortest project duration is achieved. A rational and rigorous treatment of the software engineering cost determination and estimation can be derived on the basis of the coordinative work organization theory as developed in Section 8.5, which results in the following formal economic model of software engineering costs.

The 45th Law of Software Engineering

Theorem 12.3 The *Formal Economic Model of Software Engineering Cost* (FEMSEC) states that, on the basis of the workload-driven project organization laws (Theorems 8.4 and 8.7), the expected project cost C can be rigorously determined with the optimal labor allocation L_0 and the shortest duration T_{min} by the following 6 steps:

1) Estimate the project size $\overline{S_p}$
2) Determine the ideal workload W_1
3) Allocate the optimal labor L_0
4) Determine the shortest duration T_{min}
5) Determine the expected workload W
6) Determine the expected project cost C

The procedure to determine an expected project cost according to the FEMSEC model given in Theorem 12.3 can be illustrated in Fig. 12.12.

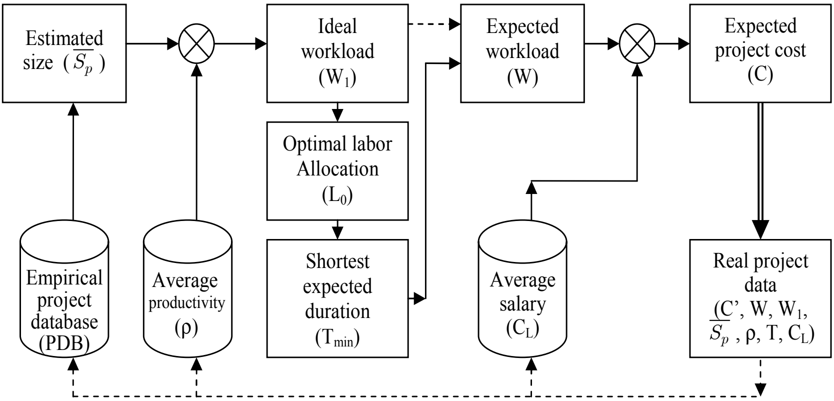

Estimated size ($\overline{S_p}$)
Ideal workload (W_1)
Expected workload (W)
Expected project cost (C)
Optimal labor Allocation (L_0)
Shortest expected duration (T_{min})
Empirical project database (PDB)
Average productivity (ρ)
Average salary (C_L)
Real project data (C', W, W_1, $\overline{S_p}$, ρ, T, C_L)

Figure 12.12 Illustration of the FEMSEC model

In the FEMSEC model, some of the steps require the availability of empirical and historical data of a specific organization, such as project size $\overline{S_p}$, average productivity ρ, and average salary C_L. The completion of each project will generate a new set of historical data, which will then be used to update the historical database. In case the historical data are not available in a certain organization, the sector's benchmarks may be used as an initial base.

12.6.2.2 The FEMSEC Method for Software Engineering Project Costs Determination

The following subsections formally describe the mathematical models for each of the six steps in the rational project costs determination process of FEMSEC.

12.6.2.2.1 Project Size Estimation

Knowing the size of a software engineering project is the starting point of cost estimation. Project sizes are usually represented by the symbolic size S_s of software systems in the unit of thousand lines of code (kLOC).

Definition 12.29 *Project size* can be estimated by a weighted average of its symbolic size, $\overline{S_p}$, as follows:

$$\overline{S_p} = \frac{1}{6}\ (S_{max} + 4S_{exp} + S_{min}) \quad [\text{kLOC}] \qquad (12.51)$$

where S_{exp} is the most likely expectation of the size of the project, S_{max} and S_{min} are the maximum or minimum expectation, respectively.

In Eq. 12.51, it can be seen that the weighted average size estimation give a higher weight to the most likely expectation. The size estimation model as defined in Definition 12.29 is a fairly accurate technique when empirical data are available on similar projects as references.

However, the empirical comparability is not always available at the whole project level in software engineering. Therefore, a more generic approach to size estimation is to use the strategy of division and conquer as described below.

Definition 12.30 Assuming a software system encompasses *n* subsystem, and each subsystem consists of *m* components, the size of this project can be estimated as a sum of the weighted average of estimated sizes of all components, $\overline{S_{ij}}$, i.e.:

$$\begin{aligned}\overline{S_p} &= \sum_{i=1}^{n}\sum_{j=1}^{m}\overline{S_{ij}} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{m}(\frac{1}{6}\ (S_{max}(i,j)+4S_{exp}(i,j)+S_{min}(i,j)) \quad [\text{kLOC}]\end{aligned} \tag{12.52}$$

Since smaller components are easier to be estimated and more similar references exist, the sum of component level estimations is much more accurate than the one-level estimations as described in Definition 12.29.

12.6.2.2.2 Ideal Workload Determination

Once the size of a given project is obtained as discussed in the previous subsection, the workload can be determined on the basis of software productivity benchmarks of a specific organization or the software industry.

Definition 12.31 The *ideal workload* of a software project is determined by the ratio of the estimated project size $\overline{S_p}$ and the software productivity ρ in terms of kLOC/PY, i.e.:

$$W_1 = \frac{\overline{S_p}}{\rho} \bullet 12 \quad [\text{PM}] \tag{12.53}$$

where a typical benchmark is ρ = 3,000 LOC/PY where management, quality assurance, and supporting activities are considered [Boehm, 1987; Dale and Zee, 1992; Jones, 1981/86; Livermore, 2005], and the units PY and PM stand for *person-year* or *person-month*, respectively.

12.6.2.2.3 Optimal Labor Allocation

According to Theorem 8.7 and the pigeon diagram as shown in Fig. 8.5, the optimal labor allocation, L_0, for a given ideal workload is solely determined by the rate of interpersonal coordination *r* of the project when multiple persons are working on it, i.e.:

$$L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \quad [P] \tag{12.54}$$

In software engineering, the coordination rate is within the scope $1\% \le r \le 90\%$. Applying Eq. 12.54, this results in the optimal labor allocation for a software engineering project is constrained by the scope of:

$$15 \geq L_0 \geq 1 \quad [P] \tag{12.55}$$

Therefore, the team organization forms for very large-scale projects have to adopt hierarchical multi-group structures, provided that each such group must still obey the same law for group size limitation and optimization. Detailed organizational strategies for multi-group large-scale projects are described in the system and social organization theories presented in Sections 10.3.5 (The system organization tree) and 13.4.2 (The formal model of social organization), respectively.

12.6.2.2.4 The Shortest Duration Determination

After the ideal workload W_1 and the optimal labor allocation L_0 are determined, the duration of a software engineering project or subproject (if multi-groups are needed) is ready to be derived.

According to Theorem 8.7, the shortest duration of a given project is determined by the following formula:

$$\begin{aligned} T_{\min} &= \{T \mid L = L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil\} \\ &= \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0}) \end{aligned} \tag{12.56}$$

The duration T_{min} can also be determined by using a set of the pigeon diagrams as show in Fig. 8.5.

12.6.2.2.5 Expected Workload Determination

When the optimal labor allocation and shortest duration of the project is determined via the methods provided in Sections 12.6.2.2.1 through 12.6.2.2.4, the expected effort or real workload of the project can be obtained.

According to Theorem 8.4, the expected workload is determined by the product of the optimal labor allocation L_0 and the shortest project duration T_{min}, i.e.:

$$\begin{aligned} W_{\exp} &= L_0 \bullet T_{\min} \\ &= L_0 \bullet \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0}) \\ &= \frac{1}{2} W_1 (rL^2{}_0 - rL_0 + 2) \quad [\text{PM}] \end{aligned} \tag{12.57}$$

Empirical workload estimations that are not based on the optimal labor allocation and the shortest project duration may result in a significant loss of effort, thus a much more expensive project, as shown in Example 8.5.

12.6.2.2.6 Expected Project Cost Determination

On the basis of the expected workload of software engineering project obtained in the preceding subsection, the cost of the given project can be determined as follows.

Definition 12.32 The *expected cost* of a software project *C* is a product of the expected workload *W* [PM] and the average cost of labor C_L [\$/PM], i.e.:

$$\begin{aligned} C_{\text{exp}} &= W_{\text{exp}} \bullet C_L \\ &= L_0 \bullet T_{min} \bullet C_L \quad [\$] \end{aligned} \tag{12.58}$$

Example 12.5 Given the estimated size of a software engineering project is $\overline{S_p}$ = 2,000LOC, determine the expected cost of this project, with the historical data or benchmarks r = 8.0%, ρ = 3.0kLOC/PY, and C_L = \$80,000/PY.

According to the FEMSEC model, the cost of the given software project can be analyzed as follows:

a) Ideal workload determination (Eq. 12.53)

$$\begin{aligned} W_1 &= \frac{\overline{S_p}}{\rho} \bullet 12 \\ &= \frac{2{,}000}{3{,}000} \bullet 12 \\ &= 8.0 \quad [\text{PM}] \end{aligned}$$

b) Optimal labor allocation (Eq. 12.54)

$$\begin{aligned} L_0 &= \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \\ &= \left\lceil \frac{1.414}{\sqrt{0.08}} \right\rceil = 5.0 \quad [P] \end{aligned}$$

c) The shortest duration determination (Eq. 12.56)

$$\begin{aligned} T_{\min} &= \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0}) \\ &= 0.5 \bullet 8.0(0.08 \bullet 5.0 - 0.08 + 2/5.0) \\ &= 2.9 \quad [M] \end{aligned}$$

d) Expected workload determination (Eq. 12.57)

$$\begin{aligned} W_{\exp} &= L_0 \bullet T_{\min} \\ &= 5.0 \bullet 2.9 \\ &= 14.5 \quad [\mathrm{PM}] \end{aligned}$$

e) Cost Determination (Eq. 12.58)

$$\begin{aligned} C_{\exp} &= W_{\exp} \bullet C_L \\ &= 14.5 \bullet 80,000/12 \\ &= \$96,666.66 \quad [\$] \end{aligned}$$

Example 12.6 For a large-scale software engineering project with $\overline{S_p}$ = 10,000LOC and r = 8.0%, determine the expected cost of this project according to the FEMSEC model with the same benchmarks as given in Example 12.5.

a) Ideal workload determination (Eq. 12.53)

$$\begin{aligned} W_1 &= \frac{\overline{S_p}}{\rho} \bullet 12 \\ &= \frac{10{,}000}{3{,}000} \bullet 12 \\ &= 40.0 \quad [\mathrm{PM}] \end{aligned}$$

b) Optimal labor allocation (Eq. 12.54)

$$\begin{aligned} L_0 &= \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \\ &= \left\lceil \frac{1.414}{\sqrt{0.08}} \right\rceil = 5.0 \quad [P] \end{aligned}$$

c) The shortest duration determination (Eq. 12.56)

$$\begin{aligned} T_{\min} &= \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0}) \\ &= 0.5 \bullet 40.0(0.08 \bullet 5.0 - 0.08 + 2/5.0) \\ &= 14.4 \quad [M] \end{aligned}$$

d) Expected workload determination (Eq. 12.57)

$$\begin{aligned} W_{\exp} &= L_0 \bullet T_{\min} \\ &= 5.0 \bullet 14.4 \\ &= 72.0 \quad [\text{PM}] \end{aligned}$$

e) Cost Determination (Eq. 12.58)

$$\begin{aligned} C_{\exp} &= W_{\exp} \bullet C_L \\ &= 72.0 \bullet 80{,}000/12 \\ &= \$480{,}000.00 \quad [\$] \end{aligned}$$

The above examples show that the key to reduce software project cost is to improve the productivity ρ. However, Theorem 1.6 states that ρ is constrained by the conservative human cognitive capability. Therefore, the only rational way is to improve automation rate in software development, i.e., to adopt automatic and intelligent software code generation systems [Wang, 2007a].

The 45th Principle of Software Engineering

Theorem 12.4 The *ultimate objective of software engineering* states that automatic code generation is the only silver bullet to overcome the natural constraints on conservative software development productivity, to reduce software development costs, and to improve software quality as a result of reduced human involvement and uncertainty.

Theorem 12.4 indicates that no matter how tough it is, intelligent and automatic software code generation systems should be developed and implemented in the future of software engineering, in order to release human labor, the low-tech means of programming, in the high-tech discipline of software engineering. A pilot automatic code generation system will be discussed in Chapter 15 on intelligent code generation [Tan, Wang, and Ngolah, 2006].

12.6.3 SOFTWARE ENGINEERING PROJECT COSTS ESTIMATION USING COCOMO

COCOMO, standing for Constructive Cost Model, is proposed by Barry Boehm in 1981 and revised in 2000 [Boehm, 1981/84; Boehm et al., 2000]. The COCOMO model is an empirical software cost model using multiple weighted nonlinear approximation techniques calibrated on the basis of 63 software projects.

12.6.3.1 The Conceptual Model of COCOMO

The basic concept of COCOMO is that the software project cost is determined by a number of factors and attributes. Therefore, empirical calibrations of these attributes based on historical data may be useful to predicate those of future projects.

Definition 12.33 The *cost factors* of software projects identified in COCOMO are software size, effort, duration, and multiple cost drivers. Their relationships are perceived as follows:

$$Cost = f(size, effort, duration, cost\ drivers) \qquad (12.59)$$

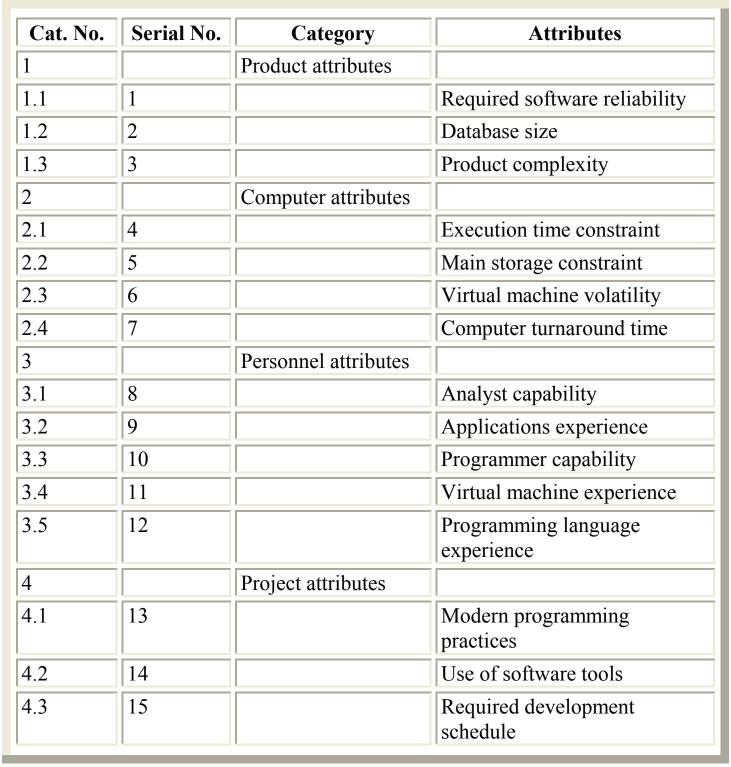
It is noteworthy that some of the cost factors in Eq. 12.59 may not independent of each other. In other words, some of the factors are derivatives of others. For example, both effort and duration are derived quantities of project size. Further, effort is determined by the duration for a given project.

The cost drives of software projects can be classified into four categories known as the *product, computer, personnel,* and *project* attributes. Fifteen cost attributes in the four categories have been identified [Boehm, 1981/84] as summarized in Table 12.5.

Boehm (1981/84) identified three different project types known as the *development modes*. They are the *organic, semidetached,* and *embedded* modes as described below:

- The *organic mode* is a type of project with a small team, experienced programmers, and familiar in-house environment in which the size of project is less than 50 kLOC.

Table 12.5
The Cost Driver Attributes of COCOMO



Cat. No.	Serial No.	Category	Attributes
1		Product attributes	
1.1	1		Required software reliability
1.2	2		Database size
1.3	3		Product complexity
2		Computer attributes	
2.1	4		Execution time constraint
2.2	5		Main storage constraint
2.3	6		Virtual machine volatility
2.4	7		Computer turnaround time
3		Personnel attributes	
3.1	8		Analyst capability
3.2	9		Applications experience
3.3	10		Programmer capability
3.4	11		Virtual machine experience
3.5	12		Programming language experience
4		Project attributes	
4.1	13		Modern programming practices
4.2	14		Use of software tools
4.3	15		Required development schedule

- The *embedded mode* is a type of project with tight constraints such as hardware, environment, timing, and performance.
- The *semi-detached mode* is a type of project that lies between the organic and embedded modes in which the size of project is between 50-300kLOC.

The estimations of software engineering project costs by COCOMO can be carried out at three levels known as the *basic* level, *intermediate* level, and *detailed* level using different approximate curves and models.

12.6.3.2 The Basic COCOMO Model

The basic COCOMO model provides a rough estimation of software project effort for small, simple, and repetitive projects [Boehm, 1981/84].

Definition 12.34 The software project *effort WM* in the basic COCOMO model is determined by the following empirical curve that is proportional to the size of the software *KDSI* and project type weights *k* and *C*, i.e.:

$$WM = C\ (KDSI)^k \tag{12.60}$$

where *WM* stands for the project effort in *work-month*, *k* and *C* are the project type or development mode constants, and *KDSI* is *thousands of delivered source instructions*.

12.6.3.3 The Intermediate COCOMO Model

The intermediate COCOMO model considers more effort factors in a software project affected by the 15 cost driver attributes as shown in Table 12.5 [Boehm, 1981/84].

Definition 12.35 The software project *effort WM* in the intermediate COCOMO model is determined by the following empirical curve that is proportional to the size of the software KDSI and project type weights e_i, *C*, and *EM*, i.e.:

$$\begin{aligned} WM &= C(WDSI) \\ &= C(KDSI)^{e_i} \prod_{j=1}^{15} EM_j\ , \quad i \in \{1, 2, 3\} \end{aligned} \tag{12.61}$$

where *WDSI* denotes the *weighted delivered source instructions*; *C* the project type or development *mode constants*; e_i exponent used for the *i*th *project type* where $i \in \{1, 2, 3\}$ represents the type of organic, embedded, or semi-detached, respectively; and EM_j are *effort multiplier* determined by the *j*th cost driver attribute within the range of 0.7 (very low complexity) to 1.66 (very high complicity).

12.6.3.4 The Detailed COCOMO Model

The detailed COCOMO model is similar to the intermediate one, but the life cycle of a project is divided into four phases known as the phases of *product design, detailed design, coding/unit test,* and *integration/test*. Each phase will be iteratively calculated by Eq.12.61 with different project type weights *C,* e_i, and *EM* [Boehm, 1981/84].

12.6.3.5 The COCOMO II Model

COCOMO II [Boehm et al., 2000] is a revision of the 1981 version of COCOMO. A software costs and effort analysis in COCOMO II still starts from the estimation of the size of the project in unit of thousand source lines of code (kSLOC). It is noteworthy that the counting methods for SLOC are greatly varying, and the result is highly dependent on programming languages.

COCOMO II adopts a set of five *scale drivers* to replace the development modes known as the project types. The scale drivers are *precedentedness, development flexibility, architecture/risk resolution, team cohesion,* and *process maturity*. The exponent used in the effort equation is determined by the scale drivers. COCOMO II extends the cost drivers from 15 to 17 to weight the effort required to complete a project.

Definition 12.36 The *effort E* of a software project in COCOMO II is estimated by the following empirical approximation, i.e.:

$$E = 2.94\ EAF \bullet (kSLOC)^{C} \quad [\mathrm{PM}] \qquad (12.62)$$

where *EAF* stands for *effort adjustment factor* derived from the 17 cost drivers, *C* is an *exponent* determined by the five scale drivers.

The unit of project effort *E* is supposed to be person-month (PM). However, it is not clear how a set of pure quantities in Eq. 12.62 may be transferred into a physical unit PM according to the convention of *dimension analysis*.

When the project effort is determined in terms of person-month (PM), the project duration *D* may be estimated as follows.

Definition 10.37 The *duration D* of a software project can be estimated by the following empirical approximation, i.e.:

$$D = 3.67 \bullet E^{SE} \quad [\mathrm{M}] \qquad (12.63)$$

where *SE* is the *schedule exponent* derived from the five scale drivers, and the unit of project duration is month (M).

Note that the effort *E* is a product of project duration *D* and number of persons *N* working in the project, i.e., *E* = *DN*. Then, *N*, the average staffing in COCOMO II, can be estimated as follows.

Definition 10.38 The *average staffing N* of a software project is the number of persons needed in the project, which can be determined by the following empirical approximation, i.e.:

$$N = E / D \quad [\text{P}] \tag{12.64}$$

where the unit of average staffing is number of persons (P).

It is noteworthy that the duration D is estimated first in the COCOMO approach and the basic assumption is that the simple product of duration and number of persons results in the effort of the project. Therefore, the duration of a project may be determined first, and then to seek how many programmers are needed.

However, theoretically, the effort product $E = DN$ is not a linear function according to Law 23 and Law 25 of software engineering proven in Theorems 8.4 and 8.7. In other words, a person is not simply equivalent to a month in the hybrid product of person-month according to Theorems 8.8 and 8.9. A rule of thumb is that the more the persons are involved in a project, the less the contribution per person to the collective person-months. A more rigorous treatment of project duration and the equivalency between labor and time in the mythical man-month is provided in Sections 12.6.2, 8.5, and 13.5.2.

12.6.4 ECONOMIC ANALYSES OF SOFTWARE PROJECTS

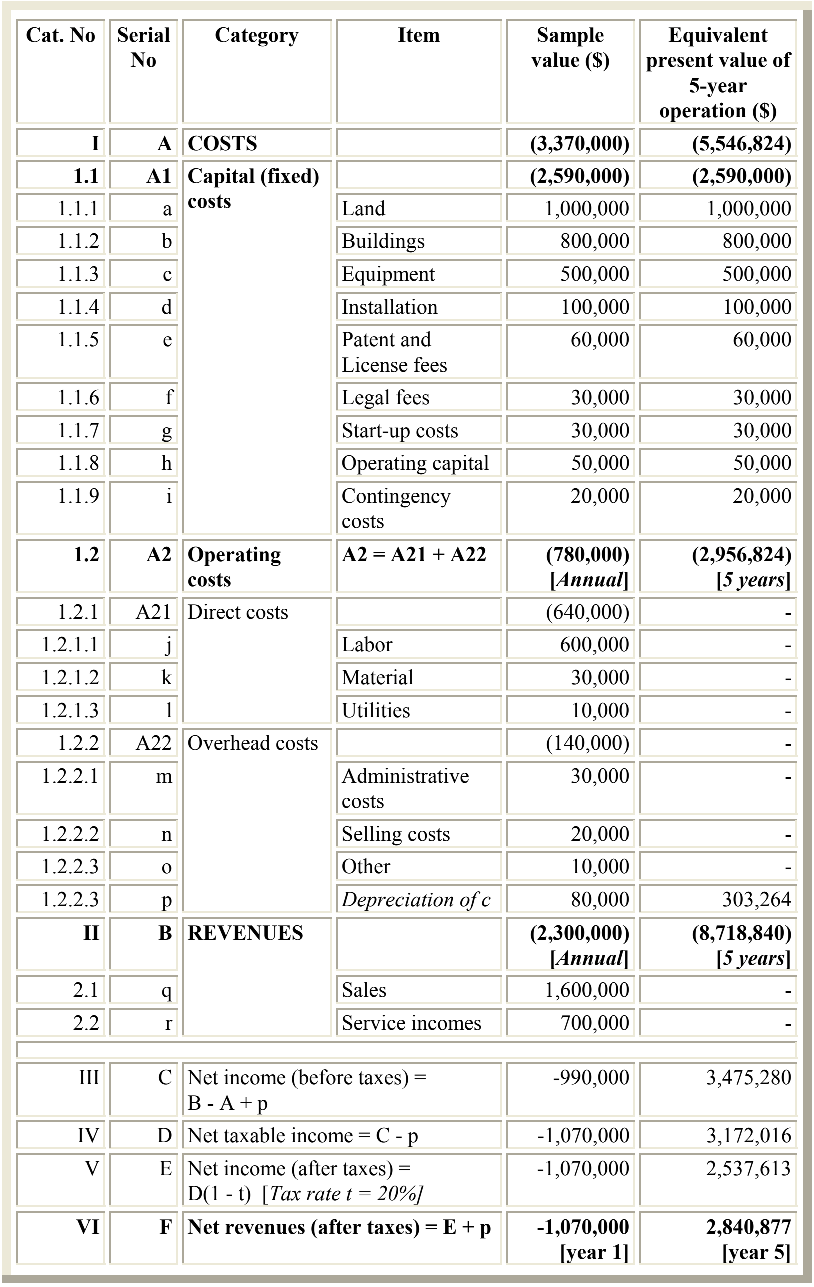
The software project costs determined by the FEMSEC and COCOMO models, presented in Sections 12.6.2 and 12.6.3, respectively, focused on the *operational cost* in economics. There are additional costs such as office, facilities, and developing environment. A complete economic analysis of software engineering project taking into account all of the categories of *developing costs* is provided in this subsection.

12.6.4.1 Estimations of Costs and Revenues of Software Projects

The economic data of an engineering project can be classified into categories of costs, revenues, and other derived cash flows. With a set of sample data on a project of a new software development organization as given in Table 12.6, economical analyses can be carried out based on the theories of engineering economics developed in Section 12.5 for this 5-year software engineering project.

On the basis of the raw figures of the software project as described in Table 12.6, the derived costs, benefit-cost ratio, return period, and return rate of this project are calculated in the following subsections, which also explain how the derived data of Table 12.6 are obtained.

Table 12.6
Estimations of Costs and Revenues of a Software Project



Cat. No	Serial No	Category	Item	Sample value ($)	Equivalent present value of 5-year operation ($)
I	**A**	**COSTS**		**(3,370,000)**	**(5,546,824)**
1.1	**A1**	**Capital (fixed) costs**		**(2,590,000)**	**(2,590,000)**
1.1.1	a		Land	1,000,000	1,000,000
1.1.2	b		Buildings	800,000	800,000
1.1.3	c		Equipment	500,000	500,000
1.1.4	d		Installation	100,000	100,000
1.1.5	e		Patent and License fees	60,000	60,000
1.1.6	f		Legal fees	30,000	30,000
1.1.7	g		Start-up costs	30,000	30,000
1.1.8	h		Operating capital	50,000	50,000
1.1.9	i		Contingency costs	20,000	20,000
1.2	**A2**	**Operating costs**	**A2 = A21 + A22**	**(780,000)** **[*Annual*]**	**(2,956,824)** **[*5 years*]**
1.2.1	A21	Direct costs		(640,000)	-
1.2.1.1	j		Labor	600,000	-
1.2.1.2	k		Material	30,000	-
1.2.1.3	l		Utilities	10,000	-
1.2.2	A22	Overhead costs		(140,000)	-
1.2.2.1	m		Administrative costs	30,000	-
1.2.2.2	n		Selling costs	20,000	-
1.2.2.3	o		Other	10,000	-
1.2.2.3	p		*Depreciation of c*	80,000	303,264
II	**B**	**REVENUES**		**(2,300,000)** **[*Annual*]**	**(8,718,840)** **[*5 years*]**
2.1	q		Sales	1,600,000	-
2.2	r		Service incomes	700,000	-
III	C	Net income (before taxes) = B - A + p		-990,000	3,475,280
IV	D	Net taxable income = C - p		-1,070,000	3,172,016
V	E	Net income (after taxes) = D(1 - t) [*Tax rate t = 20%]*		-1,070,000	2,537,613
VI	**F**	**Net revenues (after taxes) = E + p**		**-1,070,000** **[year 1]**	**2,840,877** **[year 5]**

12.6.4.2 Cumulated Value of Operating Costs

According to Definition 12.30, the cumulated present value P_Σ for the first five-year revenues of the project can be determined as follows:

$$\begin{aligned} P_\Sigma(C_{OP}) &= A\,\frac{(1+i)^n\text{-}1}{i(1+i)^n} \\ &= \$780,000 \bullet \frac{(1+0.1)^5\text{-}1}{0.1(1+0.1)^5} \\ &= \$780,000 \bullet 3.7908 \\ &= \$2,956,824 \end{aligned}$$

where the interest rate *i* is assumed at 10%.

This result is shown in Line *A2* of Table 12.6. Note that the capital costs as shown in Line *A1* of Table 12.6 are a set of one-off investment committed in year 1 of the project.

12.6.4.3 Cumulated Present Value of Revenues

The cumulated value of revenues of this project in the first five years can be conducted similarly as that of the cumulated operating costs below.

$$\begin{aligned} P_\Sigma(R) &= A\,\frac{(1+i)^n\text{-}1}{i(1+i)^n} \\ &= \$2,300,000 \bullet 3.7908 \\ &= \$8,718,840 \end{aligned}$$

This result is shown in Line *B* of Table 12.6.

12.6.4.4 Annual and Cumulated Depreciations of Equipment

The depreciation of the fixed capitals, particularly equipment, buildings, and land, can be quantitatively analyzed. There are special regulations for land and building depreciations [Park et al., 2001]. Assuming only equipment is depreciated in this project and the salvage value of all equipment of this project is $100,000, the annual depreciation of this project can be determined according to Definition 12.19, i.e.:

$$\begin{aligned} D &= \frac{P - S}{n} \\ &= \frac{\$500{,}000 - \$100{,}000}{5} \\ &= \$80{,}000 \end{aligned}$$

The cumulated value of annual equipment depreciations in five years can be derived according to Definition 12.30:

$$\begin{aligned} P_\Sigma(D) &= A\,\frac{(1+i)^n - 1}{i(1+i)^n} \\ &= \$80{,}000 \bullet \frac{(1+0.1)^5 - 1}{0.1(1+0.1)^5} \\ &= \$80{,}000 \bullet 3.7908 \\ &= \$303{,}264 \end{aligned}$$

D and $P_\Sigma(D)$ are shown in Line *p* of Table 12.6, respectively.

12.6.4.5 Project Benefit-Cost Ratios

The benefit-cost ratio *BC* is a useful indicator of the economical feasibility of a software engineering project. The total benefit (revenue) and cost of the given software engineering project for the first five years are summarized in Table 12.6. According to Definition 12.25, the benefit-cost ratio of this project can be determined as follows:

$$\begin{aligned} BC &= \frac{B}{C} \\ &= \frac{8{,}718{,}840}{5{,}546{,}824} \\ &= 1.57 \end{aligned} \tag{12.65}$$

Because the project yields a benefit-cost ratio *BC* = 1.57 > 1, it fulfills the *economical criterion* for an acceptable project as provided in Definition 12.26. In other words, this project is a profit-making project.

12.6.4.6 Project Payback Periods

Observing Table 12.6 it can be seen that this project is a uniform series of cash flow as illustrated below, where *A'* denotes the net revenue of each year that is yielded by $A' = A - C_{op}$.

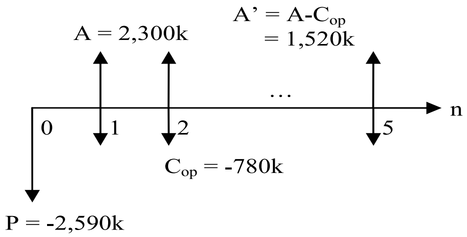


Figure 12.13 The cash flow of the software engineering project

According to Eq.12.30, $P_\Sigma(A')$ of each year in the project life span can be determined as follows:

$$\begin{aligned} P_\Sigma(A_1') &= \frac{A_1'}{1+i} \\ &= 1{,}520\text{k} \ / \ 1.1 \\ &= \$1{,}381{,}818 \end{aligned}$$

$$\begin{aligned} P_\Sigma(A_2') &= A_2' \frac{(1+i)^n\text{-}1}{i(1+i)^n} \\ &= 1{,}520\text{k} \bullet \frac{1.1^2\text{-}1}{0.1 \bullet 1.1^2} \\ &= 1{,}520\text{k} \bullet 1.7355 \\ &= \$2{,}637{,}960 \end{aligned}$$

Similarly, $P_\Sigma(A_3') = \$3{,}780{,}088$, $P_\Sigma(A_4') = \$4{,}818{,}248$, and $P_\Sigma(A_5') =$ $\$5{,}762{,}016$ are obtained.

It is apparent that the floor of payback year $\lfloor n \rfloor = 1$ for this project. Therefore, the payback period ρ can be determined below according to Eq. 12.46:

$$\begin{aligned} \rho &= \{n \mid P = P_\Sigma(n)\} \\ &\approx \lfloor n \rfloor + \frac{P(n)\text{-}P_\Sigma(\lfloor n \rfloor)}{P_\Sigma(\lceil n \rceil)\text{-}P_\Sigma(\lfloor n \rfloor)} \\ &= 1 + \frac{2{,}590{,}000 \text{ - } 1{,}381{,}818}{2{,}637{,}960 \text{ - } 1{,}381{,}818} \\ &= 1 + \frac{1{,}208{,}182}{1{,}256{,}142} \\ &= 1.96 \text{ [year]} \end{aligned}$$

12.6.4.7 Project Rate of Return

Based on the analysis result of the previous subsection, the rate of return of this project can be derived according to Eq.12.49 as follows:

$$\begin{aligned}\gamma &= \frac{1}{\rho} \bullet 100\% \\ &= \frac{1}{1.96} \bullet 100\% \\ &= 51.03\%\end{aligned}$$

It indicates that this project's rate of return is much higher than given interest rate at 10%. Therefore, this is a profitable software project that the initial investment may be recovered less than two years (ρ = 1.96 years).

12.6.5 THE SOFTWARE LEGACY COST MODEL

A special phenomenon in software engineering economics is known as the software legacy maintenance costs. A software legacy maintenance cost model can be quantitatively described based on the relationship between the development cost and maintenance cost in a software development organization.

12.6.5.1 Development Costs vs. Maintenance Costs

Definition 12.39 The *development cost* C_d is the marginal cost determined as follows:

$$C_d = k \bullet n_p \bullet n_d \tag{12.66}$$

where n_p is the average number of projects completed per year, n_d is the average number of developers per project, and k is the average cost per person.

Definition 12.40 The *maintenance cost* C_m is a cumulated cost over time t as follows:

$$\begin{aligned}C_m(t) &= k \bullet n_m \bullet N_L \\ &= k \bullet n_m \bullet t \bullet n_p\end{aligned} \tag{12.67}$$

where N_L is the number of existing legacy systems, and n_m is the average number of maintainers per legacy project, and t is time in year.

Definition 12.41 The *total costs C* of software engineering is the sum of development cost C_d and maintenance cost C_m, i.e.:

$$C = C_d + C_m \tag{12.68}$$

12.6.5.2 The Software Legacy Maintenance Cost Model

Definition 12.42 The *ratio* of the maintenance cost C_m in the total costs C, r_m, is the rate of percentage as follows:

$$r_m = \frac{C_m}{C_d + C_m} \bullet 100\% \tag{12.69}$$

Example 12.7 Assume a software development organization develops and completes three new systems each year (n_p), the number of development persons needed per system is ten (n_d), and the number of maintenance persons needed per delivered (legacy) system is three (n_m). Let all the delivered (legacy) systems have a lifespan of 20 years.

The development cost C_d, the maintenance cost C_m, the total cost C, and the ratio of maintenance cost r_m% for the legacy systems produced by this organization over 20 years can be determined according to Eqs. 12.66 through 12.69 as shown in Table 12.7.

Using the data obtained in Table 12.7, the curves of relative costs and the ratios of legacy maintenance cost can be plotted as shown in Fig. 12.14.

Table 12.7
Ratio of Maintenance Costs in a Software Development Organization

n (year)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**15**	**20**
C_d	30	30	30	30	30	30	30	30	30	30	30	30	30
C_m	0	9	18	27	36	45	54	63	72	81	90	135	180
C_{d+m}	30	39	48	57	66	75	84	93	102	111	120	165	210
r_m (%)	0	23.1	37.5	47.4	54.5	60	64.3	67.7	70.6	73	75	81.8	85.7

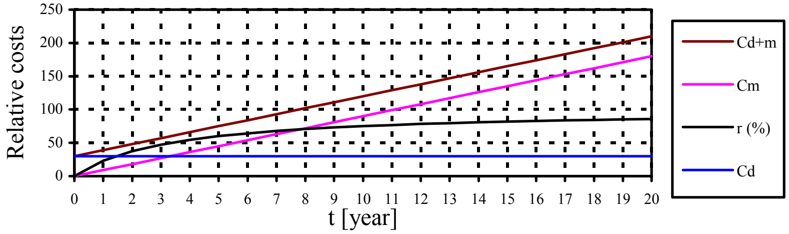


Figure 12.14 The Software Legacy Maintenance Cost (SLMC) model

It is noteworthy that the SLMC curves as shown in Fig. 12.14 are based on application software projects. For system software developers, the curves C_m and r_m% can be increased much faster than those of the trends in Fig. 12.14, because the maintenance effort would be multiple times higher when a large number of users cumulated for a system software such as operating systems and database management systems.

The 46th Principle of Software Engineering

Theorem 12.5 The *exponential Software Legacy Maintenance Costs* (SLMC) states that the ratio of maintenance cost C_m in a software development organization, r_m%, tends to exponentially increase over time t, and it is proportional to the total number of legacy systems N_L that the organization produced.

It can be observed in Fig. 12.14 that the year t_0 in which the maintenance cost overtakes the development cost in the given organization is at $t_0 \approx 3.4$ year. More formally, it can be determined as follows.

Corollary 12.5 The *overtaken time* t_o in which the maintenance cost exceeds the development cost in a software development organization can be determined using the following expression, i.e.:

$$\begin{aligned} t_0 &= \{t \mid C_m = C_d\} \\ &\approx \lfloor t \rfloor + \frac{C_m(t) - C_m(\lfloor t \rfloor)}{C_m(\lceil t \rceil) - C_m(\lfloor t \rfloor)} \quad \text{[year]} \end{aligned} \qquad (12.70)$$

Example 12.8 Using the data provided in Table 12.7, the overtaken time n_y for the software development organization as given in Example 12.7 can be estimated below:

$$\begin{aligned} t_o &= \{t \mid C_m = C_d\} \\ &\approx \lfloor t \rfloor + \frac{C_m(t) - C_m(\lfloor t \rfloor)}{C_m(\lceil t \rceil) - C_m(\lfloor t \rfloor)} \\ &= 3 + \frac{30\text{-}27}{36\text{-}27} \\ &= 3 + 0.33 \\ &= 3.33 \text{ [year]} \end{aligned}$$

The fairly short overtaken time due to maintenance costs domination in the software industry indicates a real crisis on legacy maintenance in software engineering and for the modern information-based society.

Based on the SLMC model, a corollary on an important phenomenon called *Software Maintenance Crisis* can be derived, which will be further analyzed in Section 14.3.3.

12.7 Summary

Economics is the study of how resources are used to produce and distribute commodities and how services are provided in society. **Engineering economics** is a branch of microeconomics dealing with engineering related economic decisions. Fundamental economic structures are the underlying forces of socialization and social organization. In turn, the fundamental economic structures are determined by the current and predominantly highest level of unsatisfied fundamental human needs.

Software engineering economics is a branch of applied microeconomics that studies how resources are used to produce software systems and services and how optimal decisions may be made for software engineering projects. Therefore, a successful software engineer requires certain knowledge of economics in addition to science and engineering.

This chapter has introduced fundamental principles and methodologies utilized in engineering economics and their applications in software engineering. It has also applied formal methodology into economic analysis and modeling. The first part of this chapter has reviewed classic thought and principles of economics, and a number of empirical economic models have been formalized with rigorous mathematical models. The second part of this chapter has been focused on the theories and principles of *software engineering economics*. Formal economic models for software engineering have bee developed such as the cost models and the FEMSEC model of software engineering. Applications of economic analysis and problem solving methodologies in a variety of contexts of software project decision making have been discussed. This has led to the development of the law of software legacy maintenance costs, and the finding of a hidden but significant phenomenon in software engineering known as the Software Maintenance Crisis (SMC). As a result, the **economics foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Economics Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 12. Economics Foundations of SE

■ Fundamental Principles of Economics

- Basic axioms of economics
  - Demand vs. supply
  - The principle of resource scarcity
  - The ultimate objective of markets
  - The law of maximizing profit
  - The unlimited demanding behaviors of consumers
  - The profit-driven behaviors of producers
  - The law of market conservation
- Economic equilibrium between demands and supplies
- The behaviors of market systems
  - Simple modes of economic equilibriums
  - Complex modes of economic equilibriums
  - The adaptive equilibrium mechanisms of market systems
  - The formal model of the invisible hand

■ Economic Models

- Production models
- Cost models
- Market models

■ Dynamic Values of Money and Assets

- Dynamics of money
- Dynamics of asset's values
- Cumulative values of cash flows
  - The uniform payment series
  - The linear gradient payment series
  - The geometric gradient payment series

■ Economic Analyses

- Project cost analyses
- Project benefit-cost analyses
- Project payback period analyses
- Project rate of return analyses

■ Software Engineering Economics
- Elements of software engineering costs
  - Analysis of software engineering costs
  - Analysis of software engineering revenues
- Software engineering project costs estimation using FEMSEC
  - The FEMSEC model of software engineering costs
  - The FEMSEC method for software engineering project costs determination
- SE project costs estimation using COCOMO
  - The conceptual model of COCOMO
  - The basic COCOMO model
  - The intermediate COCOMO model
  - The detailed COCOMO model
  - The COCOMO II model
- Economic analyses of software projects
  - Estimations of costs and revenues of software projects
  - Cumulated value of operating costs
  - Cumulated present value of revenues
  - Annual and cumulated depreciations of equipment
  - Project benefit-cost ratios
  - Project payback periods
  - Project rate of return
- The software legacy maintenance cost model
  - Development costs vs. maintenance costs
  - The software legacy maintenance cost model

SIGNIFICANT FINDINGS OF THIS CHAPTER

- The **theoretical framework** of **economics** is based on a number of basic axioms, which form the fundamental models of economics, such as *generic constraints of resource scarcity, the ultimate objective of markets, unlimited demanding behaviors of consumers, profit-driven behaviors of producers,* and *the law of market conservation.*

- The most basic yet important principle of economics is the recognition of a pair of contradictive phenomena, **resource scarcity vs. unlimited human demands,** in human activities and the society.

- The **equilibrium model** of market systems is a negative feedback system, in which the increase or decrease of price in the market will result in a negated feedback, and so do the changes of quantities of demands and supplies on prices, both which intends to resist the tendency of deviating from the current equilibrium.

- The **adaptive economic equilibrium** states that a market with autonomic interactions between demands and supplies is a self-regulated and self-organized system, where any change in demand, supply, or both will be autonomously adjusted to an equilibrium (Theorem 12.1).

    - The **result of interactions** between dynamic demands and supplies, through prices result in an automatic stabilization of the price at a new equilibrium that is close to the current equilibrium.

    - There are four **simple modes** that may drive a market away from an equilibrium considered in conventional economics. They are demand increase *E*(*D*+), demand decrease *E*(*D*-), supply increase *E*(*S*+), and supply decrease *E*(*D*-).

    - There are four **complex modes** based on the above simple modes, which could not formally modeled in conventional economics textbooks. They are compound demand/supply increases *E*(*D*+, *S*+), compound demand increase and supply decrease *E*(*D*+, *S*-), compound demand decrease and supply increase *E*(*D*-, *S*+), and compound demand decrease and supply decrease *E*(*D*-, *S*-).

    - The **adaptive equilibrium mechanism** is applicable to both simple and compound modes and situations as described above.

- The **adaptive equilibrium mechanism** of market systems as described in Theorem 12.1 is the **invisible hand,** which self-regulates and self-organizes the equilibrium of quantities and prices affected by the interactions between demands and supplies.

- Equilibrium market is a **conservative system**. Once an equilibrium is established in a market, the price may gradually waving around and slowly shift from P_e, but may not be increased or decreased abruptly and dramatically.

- A set of **algorithms** is provided for **numerical solutions** of cost and investment in *present* (*P*), *future* (*F*), *cumulative present value* (P_Σ), *future value* (R_Σ), *return-period* (*n*), and *return-rate* (ρ). By applying these algorithms, complicated mathematical problems in dynamic cost and investment estimation in software engineering can be solved easily. They are useful not only for project managers to plan and analyze software development costs, but also for customers to estimate investment benefit and risk of software projects.

- The differences between the **cost models of system and application software** need to be distinguished, where system software are operating systems and fundamental system tools such as language compliers, database management systems, and network/communication software; while application software are those of user developed built on top of the system software.

- The **software revenue models** between system and application software are quite different, although their cost models are fairly similar. That is, system software may create much higher revenues than those of application software systems.

- The **cost of a software engineering project** is not a linear function of the size of the project. It is more directly related to the given workload in terms of person-month.

- The **Formal Economic Model of Software Engineering Cost (FEMSEC)** reveals that the cost of a software engineering project is not simply determined by the size of the project, but is a complicated function related to the *expected workload*, form of *labor* allocation, and if the *shortest project duration* is achieved.

  - The **optimal labor allocation**, L_0, for a given project is solely determined by the interpersonal coordination rate *r*, i.e., $L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \ [P]$.

  - The **shortest duration** of a given project is determined by: $T_{\min} = \{T \mid L = L_0\} = \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0})$.

  - The **expected workload** is determined by the product of the optimal labor allocation L_0 and the shortest project duration T_{min}, i.e.: $W_{\exp} = L_0 \bullet T_{\min} = \frac{1}{2} W_1 (rL^2{}_0 - rL_0 + 2)$ [PM].

  - The **expected cost of a software project** *C* is a product of the expected workload W_{exp} [PM] and the average cost of labor C_L [\$/PM], i.e.: $C_{\exp} = W_{\exp} \bullet C_L = L_0 \bullet T_{min} \bullet C_L[\$]$.

- The **ultimate objective of software engineering** states that **automatic code generation** is the only silver bullet to overcome the natural obstacles of the conservative software development productivity, in order to reduce software development costs and to improve software quality as a result of reduced human involvement and uncertainty.

- The ***Software Legacy Maintenance Costs*** **(SLMC) model** states that the fairly short **overtaken time** due to maintenance costs domination in the software industry indicates a real crisis on legacy maintenance in software engineering and for the modern information-based society. According to the SLMC model, a corollary on an important phenomenon called *Software Maintenance Crisis* will be derived in Chapter 14.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Fundamental principles of economics

- **Demands and supplies** are the fundamental behaviors of dynamic market systems, which form the context of economics.
  - **Demand** is the required quantities for a product or service that consumers are willing and able to buy at a given range of prices.
  - **Supply** is the required quantities for a product or service that producers are willing and able to sell at a given range of prices.

- **Resource scarcity** states that the total resources at a given time or the means of production represented by their values, such as land, building, materials, labor, and capital, are constrained by an invariable nature, which is always inadequate to meet the ever growing total demands.

- The **law of market conservation** states that the prices of goods or services in a market system behave conservatively and complementally to the quantities of demands and supplies.

- The **ultimate objective of markets,** and of the producers and consumers in them, are to pursue the maximum profit P_{max}, or in other words, to maximize the revenues R_{max} and to minimize the costs C_{min} at the same time.

- The **law of maximizing profit** states that the demands and supplies of goods or services in a market system are driven by the tendency to maximize profits leveraged by the changes of prices.

- **Equilibrium** of demand and supply e is a point of quantity $Q_e(t)$ where the demand $D(t)$ equals to the supply $S(t)$, i.e., $e = \{Q_e(t) \mid D(t) = S(t)\}$, where the price at e, $P_e(t)$, is called the *equilibrium price*.

- The **equilibrium mechanism** interacting between the quantities of demands, supplies, and the prices of them in a market system is the **invisible hand.**

- The **predictability of new equilibrium** states that a *newly established equilibrium* on price P'_e is determined by the effect P' and feedback effect P'' of the driving forces deviating from the current equilibrium.

Economic models

- A set of **economic models**, such as the *production model, cost model,* and *market model*, is derived to explain the relations among a great variety of economic phenomena and their behaviors.

- **Product models: Productivity,** or the *average product*, $\overline{P}$, is a ratio between the total output O and the variable input or labor I_v.

  - The **marginal product** $\overline{P}_\Delta$ is a ratio between the incremental output and the incremental input.

  - **Law of diminishing returns s**tates that specialization efficiency is over turned by overhead of using more variable input.

- **Cost models: Total cost** of a production system C is the sum of *fixed cost* c_f and *variable cost* c_v.

  - *The* **average cost** in production $\overline{C}$ is the unit cost per product.

  - **Marginal cost** $\overline{C}_\Delta$ is the ratio between the incremental total cost and incremental output.

  - The **economical scale** of production is the maximum output that yields the minimum average cost under a certain productivity.

- **The market models:** The *market* is an economic domain in which buyers and sellers exchange commodity and services.

  - A **perfect competitive market** is a free-entry market where many sellers supply identical products or services, so that none of them may dominatingly influence the market prices.

- A **monopolistic market** is a market where only a sole supplier provides a good or service without any close substitutes.

- **Real-world markets** are operating in between the perfect competitive and monopolistic market modes. Usually, the large-scale and global industries are oligopolistic, local utility industries are monopolistic, and the remainder is perfect competitive or semi-competitive.

Dynamic values of money and assets

- A basic concept of economics is that the values of physical assets and their denoted representation, money, are a relative quantity. Both of their values change over time, or more rigorously, their value is a function of time and the interest rate.

- The **dynamic value of money,** $V(t)$, is its present worth P projected at a given point of time t for a given average or predicated interest rate i during $[0, t]$, i.e., $V(t) = f(P, i, t)$.

  - The **value with simple interest** at end of nth interest payment period, $V(n)$, or the future value $F(n)$, for a given average interest i can be determined by $V(n) = F(n) = P\ (1 + i \bullet n)$.

  - The **value with compound interest** that considers the interest based at each given period of n periods is the future value $F(n)$ at the end of each period, i.e., $F(n) = P(1+i)^n$.

- The **dynamic value of assets:** For a given asset, the continuous decreasing of value over time is known as *depreciation*.

  - The **depreciation** of assets can be classified as: a) *Physical depreciation* that refers to the reduction in asset's capacity to perform its intended service due to physical impairment; b) *Functional depreciation* that refers to obsolescence; c) *Economic depreciation* that refers to the total values lost during the life span of an asset; and d) *Accounting depreciation* that refers to a systematic allocation of the initial cost of an asset in parts over time.

- **Cumulated values of a series of cash flows** can be derived as a sum of individual payments at the same point of time, such as at present or at the end of n period in the future.

  - The **uniform payment series** is a series of identical payments A at the end of each period by a fixed frequency.

- A **linear gradient payment series** is a series of linearly increased payments *G* by a fixed frequency.

- A **geometric gradient payment series** is a series of nonlinearly increased payments *g* by a fixed frequency.

- In a generic situation, the cash flows may be a **composite series** formed by the combination of the primitive series as discussed in the previous subsections. In this case, the cumulated value can be calculated as the sum of individual component series.

Economic analysis

- **Economic analyses** cover cost and investment in *present* (P), *future* (F), *cumulative present value* (P_Σ), *future value* (R_Σ), *return-period* (n), and *return-rate* (ρ).

- **Benefit-cost ratio** *BC* of a project is a ratio between the total benefit *B* and the total cost *C*, i.e., BC = B/C.

- The **economic evaluation criterion** to accept a project is that its benefit-cost ratio is larger than one, i.e., $BC > 1$, where $BC = 1$ or $BC < 1$ represents a risky or unacceptable project, respectively.

- The **payback period** ρ of a project is the expected point of time n at which the initial investment P will be recovered by the revenues of the project P_Σ for a given interest rate i, i.e., $\rho = \{n \mid P = P_\Sigma(n)\}$.

- The **rate of return** γ of a project is the equivalent interest rate yield by a cash flow P_Σ for recovering the initial investment P for a given period n, i.e., $\gamma = \{i \mid P = P_\Sigma\}$.

Software engineering economics

- **Software Engineering Costs Analyses:** The conventional cost models in economics consider only the *fixed costs* and *variable costs*, which are oriented to the manufacturing industry characterized by mass production. For software engineering projects, more cost categories need to be studied, such as the *development costs* and *service costs*.

- The **Formal Economic Model of Software Engineering Cost (FEMSEC)** states that, on the basis of the workload-driven project organization laws (Theorems 8.4 and 8.7), the expected project cost *C* can be determined rigorously with the optimal labor allocation L_0 and the shortest duration T_{min} in the following 6 steps:

1) Estimate the project size $\overline{S_p}$
2) Determine the ideal workload W_1
3) Allocate the optimal labor L_0
4) Determine the shortest duration T_{min}
5) Determine the expected workload W
6) Determine the expected project cost C

• **The COCOMO Model:** The *cost factors* of software projects identified in COCOMO are software size, effort, duration, and multiple cost drivers. Their relationships are perceived as *Cost* = *f* (*size*, *effort*, *duration*, *cost drivers*).

• The **basic COCOMO model:** The software project *effort WM* in the basic COCOMO model is determined by the following empirical curve that is proportional to the size of the software *KDSI* and project type weights *k* and *C*, i.e., $WM = C\ (KDSI)^k$, *w*here *WM* stands for the project effort in work-month, *k* and *C* are the project type or development mode constants, and *KDSI* is thousands of delivered source instructions.

• The **Intermediate COCOMO Model:** The software project *effort WM* in the intermediate COCOMO model is determined by the following empirical curve that is proportional to the size of the software KDSI and project type weights e_i, *C* and *EM*, i.e., $WM = C(WDSI) = C(KDSI)^{e_i} \prod_{j=1}^{15} EM_j$, $i \in \{1, 2, 3\}$, where *WDSI* denotes the weighted delivered source instructions; *C* the project type or development *mode constants*; e_i exponent used for the *i*th *project type* where $i \in \{1, 2, 3\}$ represents the type of organic, embedded, or semi-detached, respectively; and EM_j are *effort multiplier* determined by the *j*th cost driver attribute with the range of 0.7 (very low complexity) to 1.66 (very high complicity).

• The **Detailed COCOMO Model:** The detailed COCOMO model is similar to the intermediate one, but the life cycle of a project is divided into four phases known as the phases of *product design, detailed design, coding/unit test,* and *integration/test.* Each phase will be iteratively calculated as those in the intermediate model with different project type weights *C, e_i*, and *EM.*

- The **COCOMO II Model:** COCOMO II adopts a set of five scale drivers to replace the development modes or known as the project types. The scale drivers are *precedentedness, development flexibility, architecture/risk resolution, team cohesion,* and *process maturity*.

- The **effort** E of a software project is estimated by the following empirical approximation, i.e., $E = 2.94\ EAF \bullet (kSLOC)^E$ [PM], where EAF stands for *effort adjustment factor* derived from the 17 cost drivers, E is an *exponent* determined by the five scale drivers, and the unit of project effort is person-month (PM).

- The **duration** D of a software project can be estimated by the following empirical approximation, i.e., $D = 3.67 \bullet E^{SE}$ [M], where SE is the schedule exponent derived from the five scale drivers, and the unit of project duration is month (M).

- The **average staffing** N of a software project is the number of persons needed in the project, which can be determined by the following empirical approximation, i.e., $N = E / D$ [P], where the unit of average staffing is number of persons (P).

- It is noteworthy that the duration D is estimated first in the COCOMO approach and the axiom is that the simple product of duration and number of persons results in the effort of the project.

- The software project costs determined by the FEMSEC and COCOMO models focused on the **operational cost** in economics. There are additional costs such as office, facilities, and developing environment. A **complete economic analysis** of software engineering project that takes into account of all the categories of developing costs is provided in Section 12.6.4.

- The **software legacy maintenance cost model** can be quantitatively described by the relation between the development cost and maintenance cost in a software development organization.

- The ***exponential Software Legacy Maintenance Costs*** (SLMC) states that the ratio of maintenance cost C_m in a software development organization, r_m%, tends to exponentially increase over time t, and it is proportional to the total number of legacy systems N_L that the organization produced.

Questions and Research Opportunities

12.1 Explain the axioms of economics on: a) The principle of resource scarcity; b) The law of market conservation; c) The ultimate objective of markets; and d) The law of maximizing profit.

12.2 What is an economic equilibrium? Try to use system theory (Theorems 10.9 and 10.10) to explain that the economic equilibrium or the market behavior is conservative.

12.3 What is the mathematical model of Adam Smith's hypothesis of the *invisible hand*?

12.4 Given a shifted equilibrium of a software market that is affected by demand increases results in the following effects (P_e, *P'*, *P''*) = ($100, $80, $160).

(a) Predicate what is the *newly established equilibrium* P'_e.

(b) Analyze what the *increment of price* ΔP is caused by the shifts of equilibriums.

12.5 Draw a diagram to show the chain of reactions of the economic equilibrium mechanism in Mode 5 – *Compound Demand Increase/Supply Increase E*(*D*+, *S*+) as formally described in the following formula:

$$E(D+, S+) = \left\langle \begin{array}{l} D\uparrow \; \rightarrow \left\langle \begin{array}{lll} \rightarrow P\uparrow & & \rightarrow \\ \rightarrow S\uparrow & \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \\ S\uparrow \; \rightarrow \left\langle \begin{array}{lll} \rightarrow P\downarrow & & \rightarrow \\ \rightarrow D\uparrow & \rightarrow P\uparrow & \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$$

12.6 Draw a diagram to show the chain of reactions of the economic equilibrium mechanism in Mode 6 – *Compound Demand Increase/Supply Decrease E*(*D*+, *S*-) as formally described in the following formula:

$$E(D+, S-) = \left\langle \begin{array}{l} D\uparrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\uparrow \qquad\qquad \rightarrow \\ \rightarrow S\uparrow \;\; \rightarrow P\downarrow \rightarrow \end{array} \right\rangle \\ S\downarrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\uparrow \qquad\qquad \rightarrow \\ \rightarrow D\downarrow \;\; \rightarrow P\downarrow \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$$

12.7 Draw a diagram to show the chain of reactions of the economic equilibrium mechanism in Mode 7 – *Compound Demand Decrease/Supply Increase* $E(D-, S+)$ as formally described in the following formula:

$$E(D-, S+) = \left\langle \begin{array}{l} D\downarrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\downarrow \qquad\qquad \rightarrow \\ \rightarrow S\downarrow \;\; \rightarrow P\uparrow \rightarrow \end{array} \right\rangle \\ S\uparrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\downarrow \qquad\qquad \rightarrow \\ \rightarrow D\uparrow \;\; \rightarrow P\uparrow \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$$

12.8 Draw a diagram to show the chain of reactions of the economic equilibrium mechanism in Mode 8 – *Compound Demand/Supply Decreases* $E(D-, S-)$ as formally described in the following formula:

$$E(D-, S-) = \left\langle \begin{array}{l} D\downarrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\downarrow \qquad\qquad \rightarrow \\ \rightarrow S\downarrow \;\; \rightarrow P\uparrow \rightarrow \end{array} \right\rangle \\ S\downarrow \;\rightarrow \left\langle \begin{array}{l} \rightarrow P\uparrow \qquad\qquad \rightarrow \\ \rightarrow D\downarrow \;\; \rightarrow P\downarrow \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$$

12.9 What are the three basic factors that uniquely determine the dynamic value of a given amount of money for a given time?

12.10 How may depreciation be used to represent and predict the dynamic value of assets?

12.11 Calculate the cumulated present value of Cash Flow 4 as given in Table 12.2 using the simple sum of present equivalent values of individual future paybacks according to Eq. 12.27.

12.12 Determine the payback period ρ of Cash Flow 4 as given in Table 12.2.

12.13 Determine the rate of return γ of Cash Flow 4 as given in Table 12.2.

12.14 Calculate the benefit-cost ratio BC of Cash Flow 4 as given in Table 12.2 assuming where the costs of the project is the initial investment plus $20,000 operating cost per year. Then, assess if the project is economically acceptable.

12.15 What are the differences of the cost models of application and system software in the categories of design, production, and services?

12.16 For a software engineering project, given the estimated size $\overline{S_p}$ = 2,000LOC, interpersonal coordination rate r = 20.0%, productivity ρ = 3.0kLOC/PY, and average salary C_L = $80,000/PY, analyze and determine the expected cost of this project according to the FEMSEC model (Theorem 12.3 and Fig. 12.12).

12.17 Compare your programming experience and the theoretical result using FEMSEC as derived in Ex. 12.16, and explain the impact of the interpersonal coordination rate r on software engineering project efforts and costs.

12.18 Re-analyze Example 12.6 assuming that the whole project is divided into three lightly-coupled parallel subprojects, therefore each subproject can be conducted independently by an individual subgroup. Then, discuss the impact of different organizational forms on project duration and costs.

12.19 Try to draw a block diagram for the COCOMO II model, and compare it with the Formal Economic Model of Software Engineering Cost (FEMSEC) as given in Fig. 12.12.

12.20 On the basis of Exs. 12.16 and 12.19, analyze how software effort and costs are derived from FEMSEC and COCOMO II, respectively, and what their advantages and disadvantages are.

12.21 Recalling the 45th law as stated in Theorem 12.3, analyze if the COCOMO methodology is in line with the theoretical constraints. Why?

12.22 How may the FEMSEC model be used to economic optimization for software engineering projects?

12.23 Why should the *ultimate objective of software engineering* be put on automatic code generation tools rather than programmer-based development?

12.24 Refers to Section 12.6.4, what is the whole framework and major categories in software engineering economic analyses according to engineering economics? How is conventional software costs estimation classified in the framework?

12.25 Try to conduct a complete economic analysis for a software engineering project as described in Section 12.6.4, where all items of raw data are doubled as given in Table 12.6.

12.26 Based on the *Software Legacy Maintenance Costs* (SLMC) model as stated in Theorem 12.5, explain what a software maintenance crisis in the software industry is.

12.27 Read the following classic article in software engineering:

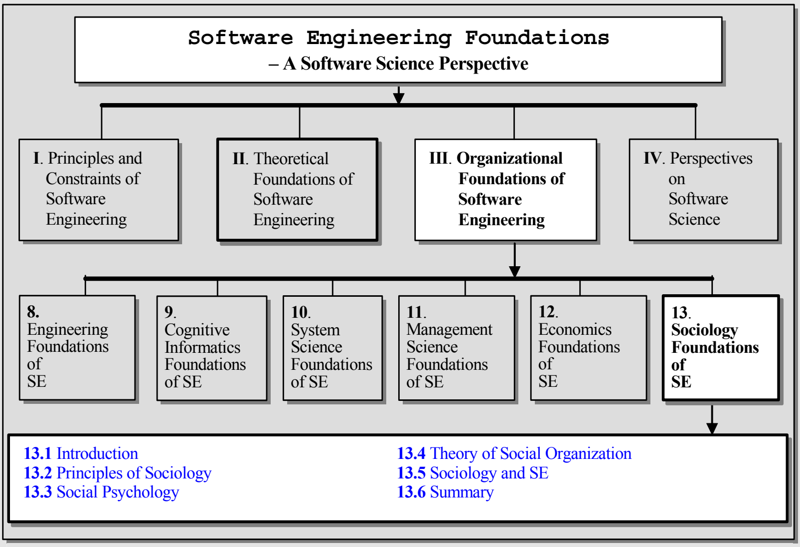
> Barry Boehm (1984), Software Engineering Economics, *IEEE Trans. on Software Engineering*, 10(1), pp. 4-12.

Discuss the following topics in a group or individually:

- About the author.
- What is the architecture of software engineering economic according to the author in the 1980s?
- What is the software cost model proposed in this article?
- What are the differences between software engineering economics and the generic engineering economics? What makes software engineering economics unique?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 13

SOCIOLOGY FOUNDATIONS OF SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
8. Engineering Foundations of SE
9. Cognitive Informatics Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE

13. Sociology Foundations of SE

Knowledge Structure

❍ Principles of Sociology

- Social Structures
- Social Behaviors
- Social Norms

❍ Social Psychology

- The Fundamental Human Traits
- Human Perceptions and Behaviors
- Collective Behaviors

❍ Theory of Social Organization

- Classic Thought of Social Organization
- The Formal Model of Social Organization
- The Formal Model of Socialization

❍ Sociology and Software Engineering

- Social Organization of SE
- Theory for Large-Scale SE Project Organization
- Human Factors in SE

Learning Objectives

- To gain knowledge on fundamental principles of sociology in terms of social structures, behaviors, and norms.
- To understand the fundamental human traits in social contexts and engineering.
- To know human collective behaviors and the driving force of motivation and attitudes.
- To understand the model of social organization and its applications in software engineering.
- To understand the formal model of socialization and interactions between sociology, economics, and human basic needs.
- To understand the social environment and principle of diversity for software engineering.
- To be familiar with the theory and laws of large-scale software engineering project organization.
- To be aware of human factors (strengths, weakness, and uncertainty) and ergonomics of software engineering.

"What I myself do not wish will never be imposed to others."

Confucius (551 – 479BC)

"From a long-life-span system perspective, the current generation who enters a society of the n*th generation, which was designed by the (*n*-1)th generation, will be responsible for the design of such a system for the (*n*+1)th generation ."*

Yingxu Wang (2003)

13.1 Introduction

Sociology studies how a human society may be organized efficiently and effectively on certain constraints of resources and environments. The objects of study in sociology are human societies. Therefore, to some extent, it may be perceived that management science is the microsociology while sociology is the macro management science. In both fields, the theories of system science and methodologies of system organizations play important roles in formalization of the theoretical frameworks of them.

Definition 13.1 *Sociology* is a branch of science that studies the structure, organization, operation, and development of human societies.

A human society is constructed by *individuals, groups, organizations,* and *sectors* from the bottom up [Wiggins et al., 1994; Macionis et al., 1997]. A group is the basic social unit formed by two or more persons working towards a particular purpose. The group is needed because of the *interdependency* among members when a given work cannot be carried out by a single individual limited by the scarcity of either resources or functions.

Various types of social organizations have been formed as results of historical, political, and/or economical processes. However, few natural laws have been sought in sociology. This chapter presents a rigorous treatment of social organization in the engineering context. The coordinative work organization theory developed in Chapter 8 can be directly applied in sociology to explain group mechanisms and behaviors. Organizational psychology and collective social behaviors within groups and organizations will be explored, which helps to explain how structures of groups and organizations may impact people's behaviors, productivity, and performance in software engineering.

Theories and methodologies of work organization [Wang, 2007d] have been one of the main thread across Chapters 8, 10, 11, 12, and 13 from engineering science, system science, management science, and economics foundations to sociology foundations. The final piece of the puzzle of the systematic theory on *coordinative work organization* will be completed in this chapter at the highest level of scopes in work organization – the society level – towards large-scale software engineering project organization. The abstract work organization theory will provide a systematic methodology for optimal allocation of labor, resources, and schedules for a given workload in a society in general, and in a software engineering context in particular.

This chapter presents a formal treatment of the sociological theories, models, and their applications in software engineering. In the remainder of this chapter, the sociology foundations of software engineering will be presented in four sections. Section 13.2 reviews fundamental principles of sociology, which covers social structures, social behaviors, and social norms. Section 13.3 explores social psychology such as the fundamental human traits, collective behaviors, and the perceptual influence on them, which form the underlying theory for explaining the human factor in engineering systems and societies. Section 13.4 develops theories of social organization that provide an essential understanding on coordinative work organization at various levels of societies. Based on the sociological models and theories, Section 13.5 extends sociology into the domain of software engineering. It explores the social organization of software engineering and ergonomics for software engineering, and explains how human strengths, weaknesses, and uncertainty may be dealt with in the context of software engineering. The coordinative work organization theory at the system level will be completed towards large-scale software engineering project organization. Then, the theoretical foundation of quality assurance in creative work such as programming and software engineering is developed.

13.2 Principles of Sociology

Sociology studies structures and behaviors of human societies. Sociology may be perceived as system science at the most complicated level of human societies and their organization. This section describes social structures, behaviors, and norms of human societies, and their basic principles.

13.2.1 SOCIAL STRUCTURES

A society is a huge organized human system in which people are grouped, coordinated, interconnected, and interacted by a variety of organizations. A society as a whole is constructed by individuals, groups, organizations, and sectors from the bottom up as shown in Fig. 13.1.

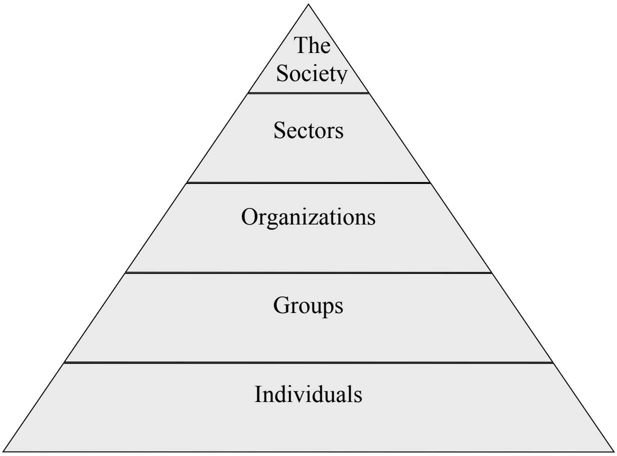


Figure 13.1 The hierarchical structure of a society

Social structures study the hierarchical architectures of societies at different levels and their social characteristics and interactions. This section focuses on the taxonomy and static structures. More formal treatment of groups and organizations will be discussed in Section 13.4 on theories of social organization.

13.2.1.1 Individuals

An individual is the bottom level and basic social unit of a society. The individuals are the most dynamic factor and the underlying driving force of a society.

Definition 13.2 An *individual* is a single human being that forms the basic social unit of a society.

When the behavior of individuals is studied, sociology puts emphases on relationships and interactions of individuals and related social structures; while social psychology focuses on human traits, characteristics, and behaviors. This section describes the former. The latter will be discussed in Section 13.3.

13.2.1.2 Groups

When multiple individuals work together towards a particular goal or interact intensively, a permanent or temporary group is formed.

Definition 13.3 A *group* is a formal or informal social unit formed by two or more persons working towards a particular purpose.

A group is the second level unit of a society, or a subsystem of the society according to system theory as discussed in Chapter 10. The individuals of a group are interdependent and they are identified with a single identity of that of the group.

When a number of individuals are treated as a whole, both internal relationships and external interactions of the group with the environment need to be studied.

The importance of studies on groups is well explained by Kurt Lewin in 1948 [Lewin, 1948; Zander, 1979].

> "Although the scientific investigations of group work are but a few years old, I don't hesitate to predict that group work – that is, the handling of human beings not as isolated individuals, but in the social setting of groups – will soon be one of the most important theoretical and practical fields."

Lemma 13.1 A group is needed because of the *interdependency* among members when a given work cannot be carried out by an individual limited by either resource dependency or functional dependency.

The *basic architectures* of groups in term of interrelationships among members can be classified into the forms of *serial, parallel, star, network, hierarchical,* and their combinations.

Work groups strongly influence the overall behaviors and performance of members. The cohesive bounds that keep members of a group together are identified as the bounds of *membership*, *goals*, *norms*, and *external oppressions* [Wiggins et al., 1994]. Lemma 13.1 explains that the *interdependency* is the essential natural force that keeps a group together.

13.2.1.3 Organizations

An organization is a superset of groups and the third-level subsystem of the society from the bottom-up.

Definition 13.4 An *organization* is a formal and stable social unit formed by one or more groups of people working towards a particular purpose.

A group is formed because of the extended needs for either the resource dependency or the functional dependency. When the scale of a group is increasing to a certain extent, internal coordination and synchronization between members in the group will be the dominant problem. This problem forces a large group to adopt more structured forms of organization.

More formal treatment of groups and organizations will be discussed in Section 13.4 on theories of social organization.

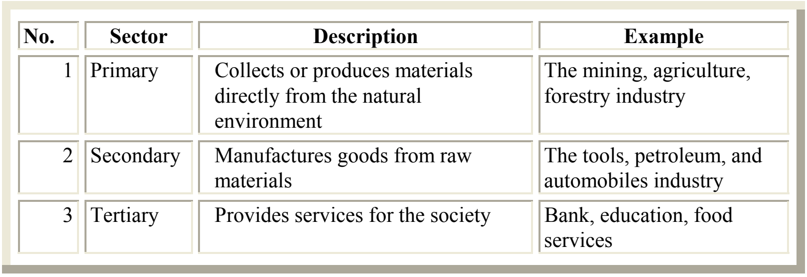
13.2.1.4 Sectors

A sector is a functional level of the hierarchical societies, where multiple organizations associate and interact due to their dependency and/or similarity.

Definition 13.5 A *sector* is a distinct branch of a society with multiple organizations that produce the same category of products or provide the same category of services.

Sectors are a macro categorization of organizations. A sector may geographically cross large areas in the scope of a country even of the world. The top-level sectors of an economy can be classified into the *primary*, the *secondary*, and the *tertiary sectors* as described in Table 13.1.

Table 13.1
Main Sectors of the Economy



No.	Sector	Description	Example
1	Primary	Collects or produces materials directly from the natural environment	The mining, agriculture, forestry industry
2	Secondary	Manufactures goods from raw materials	The tools, petroleum, and automobiles industry
3	Tertiary	Provides services for the society	Bank, education, food services

It is noteworthy that, in the postindustrial society, a fourth major sector emerging in the underpinning economy is the information sector. This trend will be analyzed in Section 13.4.

13.2.1.5 Societies

A society is the top level structure in the hierarchy of human organizations. Societies at the top level may be studied by the mechanisms and behaviors of lower level structures.

Definition 13.6 A *society* is the community of people in which members of it are geographically connected and socially integrated with common customs, organizations, and values.

There are different societies constrained by the economic structures and their levels of development. The economic structures in turn are driven by the fundamental human needs and demands. The relationships and interactions between the human needs, economic structures, and social types will be discussed in Section 13.4.2 on the formal socialization model of human societies.

13.2.2 SOCIAL BEHAVIORS

The dynamic aspect of human societies is their social behaviors. The study of social behaviors can be carried out hierarchically via social functions, relations, roles, and systems from the bottom up in a society.

13.2.2.1 Social Functions and Relations

Social functions are the minimum functional components of a society. The behaviors of a dynamic society can be modeled by a huge set of interacting functions.

Definition 13.7 A *social function* $\mathfrak{F}$ is a set of tasks and/or actions within a society that can be carried out by individuals.

High-level social functions can be divided into two categories: *public* functions and *private* functions. The former can be any job function in an organization or company. The latter can be such as family members and friends.

Definition 13.8 A *social relation R* is a function between two or more persons, *p*, in a society, i.e.:

$$R(p) = r : \ p \to P \qquad (13.1)$$

where *P* is all the individuals, $p \in P$, in the given society.

Social relations provide a constructive force for the building of a society. A person's membership in a society is highly indicated by the person's internal relations with other members in the society.

13.2.2.2 Social Roles

The conception of oneself is dependent on the roles that one performs in a society. In his work on *The Study of Man* [Linton, 1936], Robert Linton proposed the role theory. The *role theory* analogizes a person as an actor who plays an assigned role in accordance with a script specified by culture and the society. According to Linton, a role is a set of social expectations that apply to the behavior of specific categories of people in particular contexts. With the understanding of the roles, we can predicate who does what, when, and where in a given society. Glen Elder extended the role theory to a life-course framework in 1975 that explains the roles of members of a society according their ages and life stages [Elder, 1975].

Informally, a social role can be defined as follows.

Definition 13.9 *A social role* is a set of coherent social functions that is represented by a title of a category and is expected to be conformed in the society.

More formally, the social roles of a person are given below.

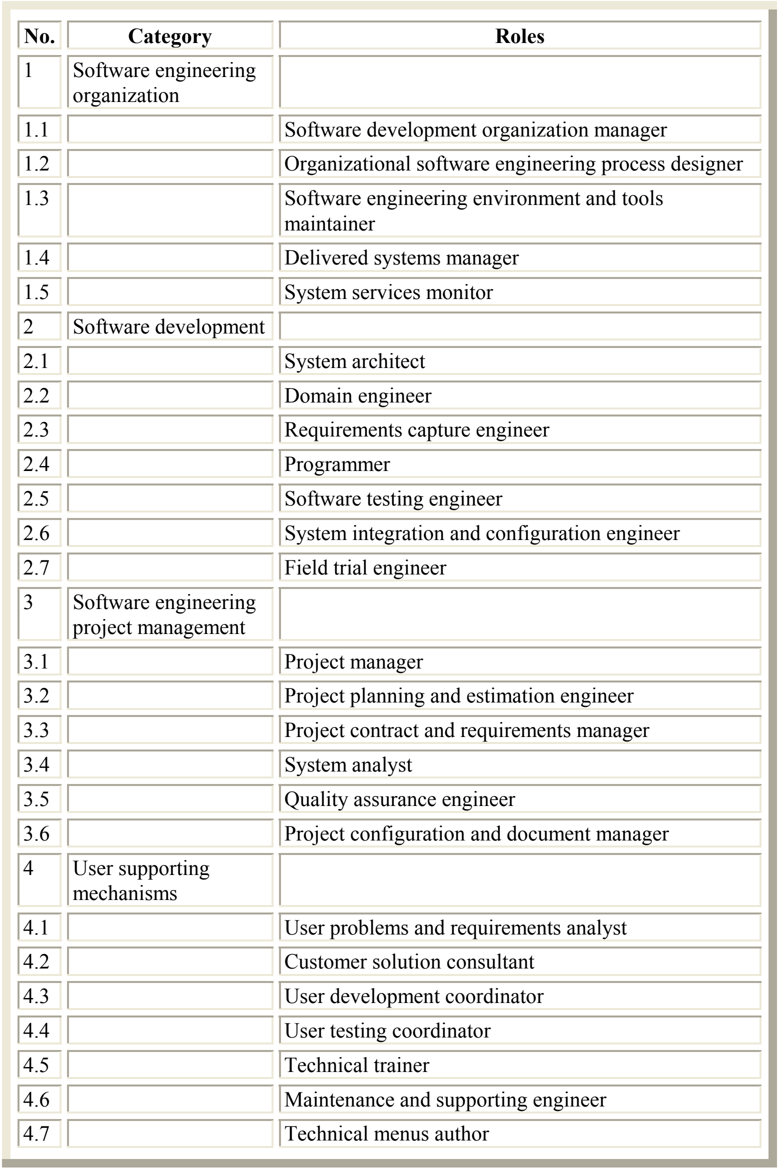
Definition 13.10 The *social roles SR* of a person *p* is a relation between the person *p* and a set of social functions *F*, $F \subseteq \mathfrak{F}$, i.e.:

$$SR(p) = f : p \rightarrow F \tag{13.2}$$

where *F* is a subset of all defined social functions $\mathfrak{F}$.

In 1922, Robert Park pointed out that it is in roles that we come to know ourselves as sociological man [Park, 1922]. A social role can be public or private such as an engineer and a father. The functions, tasks, and expectations for a given role may be well defined. For example, a software engineer is a professional whose roles and skills are regulated by the software engineering discipline and processes. Examining the requirements for functions of software engineers in software engineering at the technical, managerial, and organizational levels, a variety of roles can be identified as shown in Table 13.2 [Wang and King, 2000a].

Table 13.2
Roles of Software Engineers in Software Engineering



No.	Category	Roles
1	Software engineering organization	
1.1		Software development organization manager
1.2		Organizational software engineering process designer
1.3		Software engineering environment and tools maintainer
1.4		Delivered systems manager
1.5		System services monitor
2	Software development	
2.1		System architect
2.2		Domain engineer
2.3		Requirements capture engineer
2.4		Programmer
2.5		Software testing engineer
2.6		System integration and configuration engineer
2.7		Field trial engineer
3	Software engineering project management	
3.1		Project manager
3.2		Project planning and estimation engineer
3.3		Project contract and requirements manager
3.4		System analyst
3.5		Quality assurance engineer
3.6		Project configuration and document manager
4	User supporting mechanisms	
4.1		User problems and requirements analyst
4.2		Customer solution consultant
4.3		User development coordinator
4.4		User testing coordinator
4.5		Technical trainer
4.6		Maintenance and supporting engineer
4.7		Technical menus author

A significant finding in observing Table 13.2 is that a software engineer may be responsible for only one or limited role(s) rather than a master of all the skills in software engineering processes. This is one of the fundamental principles of engineering that is so obvious and so often to be ignored in practice. This is what we learnt from the universal principles of industrial engineering methodologies.

In a modern society, a person usually takes multiple roles from family, groups, organizations, and the society. It is interesting to observe the switching among the roles of individuals, as well as the influence and interference between them, in organizational sociology of engineering and science.

13.2.2.3 Social Systems

Contemporary social theories view human societies as a system. Therefore, system theories developed in Chapter 10 may be applied in the rigorous treatment and quantitative analyses of human societies.

Definition 13.11 A *society* is a dynamic human system that is interacting not only among members of the society via social relations, but also between the society, other societies, and the natural environment.

There are various types of societies characterized by the structures of economies of the societies. From a historical point of view, according to the economic structures underpinning the societies, human societies have evolved through five phases known as the *hunting/gathering, horticultural/pastoral, agrarian, industrial,* and *postindustrial societies*.

As social relations adhere people to people in a society, social roles adhere people to social functions. Therefore, social relations and social roles are the fundamental mechanisms in the construction of society. Because both social relations and social roles can be 1-to-1, 1-to-*n*, *n*-to-1, and *n*-to-*m*, the natural structures of human societies are hierarchical trees and networks.

13.2.3 SOCIAL NORMS

Norms are the *shoulds* of a society for regulating social behaviors that members of the society share and are expected to conform. Social norms can be considered from the aspects of cultures and values.

13.2.3.1 Cultures

A culture refers to a shared way of life [Macionis et al., 1997]. The *custom* of a social unit is a set of traditional and widely accepted habits of social behaviors shared by the members of the unit through long-term interactions.

Definition 13.12 The *culture* of a society is the collected ideas, customs, behaviors, and values shared by members of the unit.

Culture shapes what individuals do and influences individuals' behaviors and personality. The basic components shared by different cultures are *symbols, language, value, norms,* and *material objects*.

Languages play an important role in cultures because they are the media of cultures and the means of transmission from person to person and from generation to generation. Two anthropologists and linguistics, Edward Sapir and Benjamin Whorf, observed that people perceive the world through the cultural lens of language known as the *Sapir-Whorf hypothesis* [Sapir, 1929; Whorf, 1941].

13.2.3.2 Values

Values are guidelines of a culture shared by people for social judgment and behavioral normalization.

Definition 13.13 *Values* of a social unit are a set of ethical principles or standards shared by the unit that are used to judge and normalize social behaviors.

In 1970, Robin Williams identified the top nine central values of typical North America people [Williams, 1970] as follows:

- Equal opportunity
- Achievement/success
- Activity/work
- Material comfort
- Practicality/efficiency
- Progress
- Science
- Democracy/free enterprise
- Freedom

It is noteworthy that cultures are a dynamic entity undergoing continuous changes via cultural invention, discovery, and diffusion. Due to technological and economical advances, such as new communication techniques, travel, and migration, a global culture is emerging that is formed from a conjunction of traditionally different cultures [Macionis et al., 1997].

13.2.3.3 Socialization

Although each individual experiences a different life, statistically, the life courses of all individuals are similar in a society. That is because the current generation of individuals enters a given society predesigned and normalized by the earlier generations. Although they have no chance to shape the existing society, they are able to transfer it for the general welfare of the next generation. This is the historical view of socialization [Wang, 2005k].

Definition 13.14 *Socialization* is a conforming process that a person is integrated into a society at various levels of its hierarchy by adopting certain roles, relations, cultures, customs, value systems, and norm behaviors.

A society is an invisible network with unwritten rules, norms, and standards established well before a young person's entry. No matter observed or not, people feel the socialization stress and synchronization pressure at work. Therefore, social psychologists believe that travel is one of the best releases for people because travel enables one to temporarily escape from the invisible social networks.

13.2.3.4 The Social Philosophy of Confucianism

Confucianism, created by Confucius (551 – 479BC), a Chinese philosopher, educationist, and sociologist, is a crystallization of the Chinese social philosophy and ethical values during a five thousand year civilization. The essences of Confucianism may be summarized by the nine key words: humaneness, integrity, ritual, righteousness, loyalty, piety, tolerance, introspectiveness, and gentlemanliness. The essential values of Confucianism establish a set of unified and stable social norms as explained below:

- *Humaneness* (*ren,* in Chinese) is the norm of social attitude and justice. A best interpretation of humaneness is by the words of Confucius: "What I myself do not wish will never be imposed to others."
- *Integrity* or *honesty* (*xin*) is the norm of social values.
- *Ritual* or *politeness* (*li*) is the norm of personal behavior.
- *Righteousness* (*yi*) is the norm of ethical values.
- *Loyalty* (*zhong*) is the norm of socialization.
- *Piety* (*xiao*) is the norm of family relationship, particularly towards seniors.
- *Tolerance* (*ren rang*) is the norm of interpersonal relationships.
- *Introspectiveness* (*zi xing*) is the norm of inner purity.
- *Gentlemanliness* (*junzi*) is the norm of morals.

It is not a surprise at all that accurate counterparts of concepts and values of North America values and Confucianism may be found in other civilizations, cultures, or languages. This observation leads to the following lemma.

Lemma 13.2 The *union* of all proven social norms from different societies, or at least their *intersection*, represents a set of univeral values of humanity.

The identification of a common set of values may be helpful to normalize individual and collective behaviors in an organization, especially a software development organization in software engineering that produces information products for a global market.

13.3 Social Psychology

Social psychology is a branch of psychology that studies social interactions and their effects on human behaviors [Wiggins et al., 1994]. Because the basic objects under study in sociology are individual human beings and their interactions, social psychology is the key to understand a wide range of complicated social phenomena and the driving forces underpinning them.

This section explores the fundamental human traits and the basic needs of individuals in a society. Motivations and attitudes are studied in order to understand the natural drives and constraints of human social behaviors. The characteristics of collective behaviors of individuals are analyzed in the social context.

The study on human traits forms the foundation of sociology, because every individual's social behavior is driven and constrained by those axiomatic human traits and characteristics and the derived needs based on them. The study on human traits also forms the foundation for engineering organization.

13.3.1 THE FUNDAMENTAL HUMAN TRAITS

Human traits and needs are the fundamental force underlying almost all phenomena in human task performances, engineering organizations, and societies. This subsection explores the cognitive foundations of human traits and cognitive properties of human factors in engineering. The fundamental traits of human beings are identified, and the hierarchical model of basic human needs is formally described. The characteristics of human factors and their influences in engineering organizations and socialization are explored. Based on the models of basic human traits, needs, and their influences, the driving forces behind the human factors in engineering and society are revealed. A formal model of human errors in task performing is derived, and case studies of the error model in software engineering are presented.

13.3.1.1 Axiomatic Human Traits

The basic evolutional need of humans is the tendency to maximize the inclusive fitness of individuals and the whole mankind.

Definition 13.15 *Egoism* is a social behavior of human beings in which individuals put their own interests first in decision makings.

Both sociologists and economists believe that egoism drives most of the behaviors of individuals. However, statistically, all individual behaviors as a whole form the natural force towards the development and welfare of the entire society.

The basic forms of egoism of individuals are to *maximize* personal *lifespan, profit, pleasure, esteem, power,* and *information*, and to *minimize costs, energy consumption,* and *inconvenience*. It is noteworthy that most forms of egoism are dependent on the cooperation or recognition of others or the society. This basic constraint is the sociological foundation of altruism.

Definition 13.16 *Altruism* is a social behavior in which individuals sacrifice their own interests for the welfare of a group or society.

Altruism can be explained by the term of inclusive fitness as defined below.

Definition 13.17 The *inclusive fitness* of human beings is their own reproductive success and those of generically related individuals [Fried and Hademenos, 1999].

Lemma 13.3 Egoism is constrained by altruism; and the implementation of altruism is dependent on the natural egoism.

Lemma 13.3 provides an explanation of the relationship between egoism and altruism. Based on Lemma 13.3, the following theorem can be derived.

The 46th Law of Software Engineering

Theorem 13.1 The *basic essences for evolution* state that the *basic evolutional needs* of mankind are to preserve both the species' biological traits via *gene pools*, and the cumulated knowledge via various *information systems*.

The history indicates that evolution favors species like human beings and other organisms that are able to seek the maximum inclusive fitness.

13.3.1.2 The Hierarchical Model of Basic Human Needs

As an individual, the basic biological need of humans is a stable inner environment regulated by a mechanism known as homeostasis.

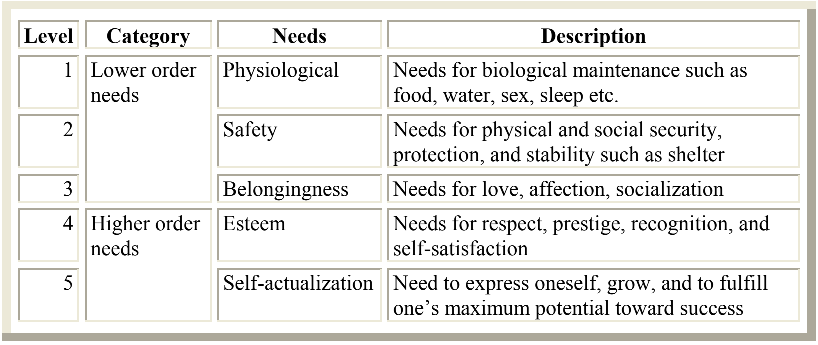
Definition 13.18 *Homeostasis* is an adaptive biological mechanism of the human body that maintains a relatively constant state in order to live and function.

At the psychological level, Sigmund Freud perceived that humans are motivated by internal tension states known as *drives* that build up until they are released. The basic drives that Freud identified are self-preservation, sex, and aggression. However he focused only on the last two drives later in his theory [Freud, 1895; Leahey, 1980].

Clark Hull proposed the drive-reduction theory that states motivation stems from a combination of drive and reinforcement of unfulfilled needs [Hull, 1943]. The *primary drives* are innate drives such as hunger, thirst, and sex; the *secondary drives* are acquired drives such as studying, socializing, and earning money.

The hierarchy of human needs is identified by Abraham Maslow at five levels known as the needs of *physiological, safety, social, esteem,* and *self-actualization* from the bottom up [Maslow, 1962/70]. The five basic levels of human needs are described in Table 13.3. Except those at Level 5, most needs identified by Maslow as shown in Table 13.3 are *deficiency* needs, which are a need generated by a lack of something. The Level 5 needs for self-actualization can be perceived as a *growth* needs.

Table 13.3
Maslow's Hierarchy of Needs



Level	Category	Needs	Description
1	Lower order needs	Physiological	Needs for biological maintenance such as food, water, sex, sleep etc.
2		Safety	Needs for physical and social security, protection, and stability such as shelter
3		Belongingness	Needs for love, affection, socialization
4	Higher order needs	Esteem	Needs for respect, prestige, recognition, and self-satisfaction
5		Self-actualization	Need to express oneself, grow, and to fulfill one's maximum potential toward success

On the basis of the needs taxonomies of Maslow, Hull, and Freud, a formal human needs hierarchy model is provided in Definition 13.19, Fig. 13.2, and Table 13.4.

Definition 13.19 The *Human Needs Hierarchy* (HNH) model is a hierarchical model that encompasses five-level fundamental human needs known from the bottom-up as N_0 – physiological needs, N_1 – psychological needs, N_2 – cognitive needs, N_3 – social needs, and N_4 – self-expressive needs.

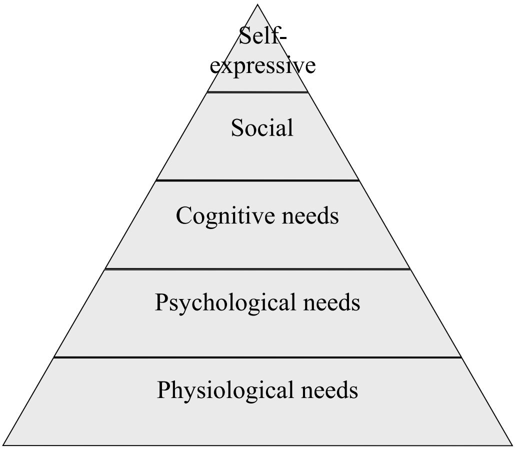
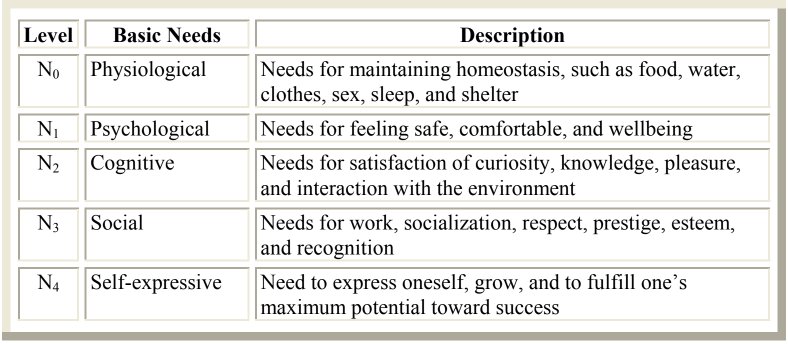


Figure 13.2 The Human Needs Hierarchy (HNH) model

The HNH model can be illustrated as shown in Fig. 13.2. Detailed explanations of each of the basic needs are provided in Table 13.4.

Table 13.4
The Human Needs Hierarchy (HNH) Model



Level	Basic Needs	Description
N_0	Physiological	Needs for maintaining homeostasis, such as food, water, clothes, sex, sleep, and shelter
N_1	Psychological	Needs for feeling safe, comfortable, and wellbeing
N_2	Cognitive	Needs for satisfaction of curiosity, knowledge, pleasure, and interaction with the environment
N_3	Social	Needs for work, socialization, respect, prestige, esteem, and recognition
N_4	Self-expressive	Need to express oneself, grow, and to fulfill one's maximum potential toward success

Lemma 13.4 The lower the level of a need in the HNH hierarchy, the more concrete or *material-oriented* the need. In other words, the higher the level of a need, the more *virtualized* or *perception-oriented* the need.

Definition 13.20 The *predominant need* of an individual is the needs at the lowest unsatisfied level of the HNH model.

Maslow suggests that human needs should be satisfied level by level. That is, the lower level needs should be satisfied before any higher level need comes into play [Maslow, 1970]. This observation leads to the following corollary.

Corollary 13.1 When *multiple needs* of a person are unsatisfied at a given time, satisfaction of the most *predominant need* is most pressing.

Understanding of the nature of basic human needs is not only useful in predicating motivations of human beings in a given context, but also important in identifying the driving forces for the approach of engineering organization, the types of societies, and the corresponding economic structures.

13.3.2 HUMAN PERCEPTIONS AND BEHAVIORS

Perception is the third layer of human cognitive processes modeled in LRMB as developed in Chapter 9. This subsection presents a rigorous treatment of human perceptual processes such as emotions, motivations, and attitudes, and their influences on human behaviors [Wang, 2007i]. A set of mathematical models and cognitive processes is developed. The interactions and relationships between motivation and attitude are formally described. Applications of the mathematical models of motivations and attitudes in software engineering are demonstrated.

According to Definition 9.9, *perception* is a set of sensational cognitive processes at the subconscious cognitive function layers such as emotion, motivation, and attitudes. Perception may be considered as the sixth sense of human beings that almost all cognitive life functions rely on it.

13.3.2.1 Emotions

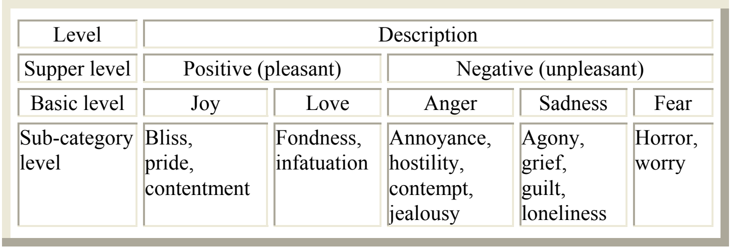
Emotions are a set of states or results of perception that interprets the feelings of human beings on external stimuli or events in the binary categories of pleasant or unpleasant.

Definition 13.21 An *emotion* is a personal feeling derived from one's current internal status, mood, circumstances, historical context, and external stimuli.

Emotions are closely related to desires and willingness. A *desire* is a personal feeling to possess an object, to conduct an interaction with the external world, or to prepare for an event to happen. A *willingness* is the faculty of conscious, deliberate, and voluntary choice of actions.

According to the study of Fischer and his colleagues [Fischer et al., 1990], the taxonomy of emotions can be described as shown in Table 13.5.

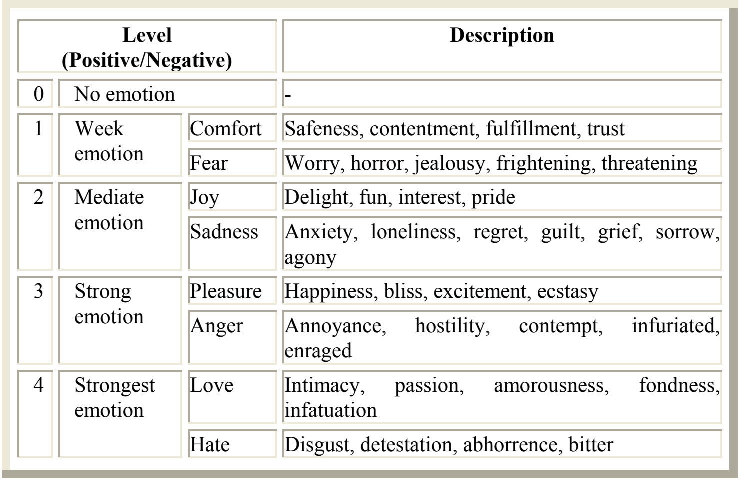
Table 13.5
Taxonomy of Emotions



Level	Description				
Supper level	Positive (pleasant)		Negative (unpleasant)		
Basic level	Joy	Love	Anger	Sadness	Fear
Sub-category level	Bliss, pride, contentment	Fondness, infatuation	Annoyance, hostility, contempt, jealousy	Agony, grief, guilt, loneliness	Horror, worry

It can be observed that human emotions at the perceptual layer may be classified into only two opposite categories: *pleasant* and *unpleasant*. Various emotions in the two categories can be classified at five levels according to its strengths of subjective feelings as shown in Table 13.6, where each level encompasses a pair of positive/negative or pleasant/unpleasant emotions.

Table 13.6
The Hierarchy of Emotions



Level (Positive/Negative)			Description
0	No emotion		-
1	Week emotion	Comfort	Safeness, contentment, fulfillment, trust
		Fear	Worry, horror, jealousy, frightening, threatening
2	Mediate emotion	Joy	Delight, fun, interest, pride
		Sadness	Anxiety, loneliness, regret, guilt, grief, sorrow, agony
3	Strong emotion	Pleasure	Happiness, bliss, excitement, ecstasy
		Anger	Annoyance, hostility, contempt, infuriated, enraged
4	Strongest emotion	Love	Intimacy, passion, amorousness, fondness, infatuation
		Hate	Disgust, detestation, abhorrence, bitter

Definition 13.22 The *strength of emotion* $|E_m|$ is a normalized measure of how strong a person's emotion on a scale of 0 through 4, i.e.:

$$0 \leq |E_m| \leq 4 \qquad (13.3)$$

An organ known as the *hypothalamus* in the brain is supposed to interpret the properties or types of emotions in terms of pleasant or unpleasant [Smith, 1993; Leahey, 1997; Sternberg, 1998].

Definition 13.23 Letting T_e be a type of emotion, *ES* the external stimulus, *IS* the internal perceptual status, and BL the Boolean values true or false, the perceptual mechanism of hypothalamus can be described as a function, i.e.:

$$T_e : ES \times IS \rightarrow \text{BL} \qquad (13.4)$$

It is interesting that sometime the same event or stimulus *ES* may be explained in different types due to the difference of the real-time context of the perceptual status *IS* of the brain. For instance, walking from home to office may be interpreted as a pleasant activity for one who likes physical exercises, but the same walk due to a car breakdown will be interpreted as unpleasant.

Corollary 13.2 The *human emotional system* is a binary system that interprets or perceives an external stimulus and/or internal status as pleasant or unpleasant.

Although there are various emotional categories in different levels, the binary emotional system of the brain provides a set of pairwise universal solutions to express human feelings. For example, anger may be explained as a default solution or generic reaction for an emotional event when there was no better solution available; otherwise, delight will be the default emotional reaction.

13.3.2.2 Motivations

Motivation is an innate potential power of human beings that energizes behavior. It is motivation that transforms thought (information) into action (energy). In other words, human behaviors are the embodiment of motivations. Therefore, any cognitive behavior is driven by an individual motivation.

Definition 13.24 A *motivation* is a willingness or desire triggered by an emotion to pursue a goal or a reason for triggering an action.

As described in the LRMB model [Wang et al., 2006], motivation is a cognitive process of the brain at the perception layer that explains the initiation, persistence, and intensity of personal emotions and desires, which are the faculty of conscious, deliberate, and voluntary choices of actions.

Motivation is a psychological and social modulating and coordinating influence on the direction, vigor, and composition of behavior. This influence arises from a wide variety of internal, environmental, and social sources, and is manifested at many levels of behavioral and neural organizations.

The taxonomy of motives can be classified into two categories known as learned and unlearned [Wittig, 2001]. The latter is the primary motives such as the *survival motives* (hunger, thirst, breathing, shelter, sleep, eliminating), and pain. The former is the secondary motives such as the need for achievement, friendship, affiliation, dominance of power, and relief from anxiety.

Definition 13.25 The *strength of motivation M* is a normalized measure of how strong a person's motivation is on a scale of 0 through 100, i.e.:

$$0 \leq M \leq 100 \tag{13.5}$$

where M = 100 is the strongest motivation and M = 0 is the weakest motivation.

It is observed that the strength of a motivation is determined by multiple factors [Westen, 1999; Wang, 2007d] such as:

a) The *absolute motivation* $|E_m|$: The strength of the emotion.

b) The *relative motivation* E - S: A relative difference or inequity between the expectancy of a person E for an object or an action towards a certain goal and the current status S of the person.

c) The *cost* to fulfill the motivation C: A subjective assessment of the effort needed to accomplish the expected goal.

Therefore, the strength of a motivation can be quantitatively analyzed and estimated by the subjective and objective motivations and their cost as described in the following theorem [Wang, 2007d].

The 47th Principle of Software Engineering

Theorem 13.2 The *strength of motivations* states that a motivation M is proportional to both the strength of emotion $|E_m|$ and the difference between the expectancy of desire E and the current status S, of a person, and is inversely proportional to the cost to accomplish the expected motivation C, i.e.:

$$M = \frac{2.5 \bullet | E_m | \bullet (E\text{-}S)}{C} \tag{13.6}$$

where $0 \leq |E_m| \leq 4$, $0 \leq (E,S) \leq 10$, and $1 \leq C \leq 10$.

In Theorem 13.2, the strength of a motivation is measured in the scope of [0 … 100], i.e., $0 \leq M \leq 100$. When $M > 1$, the motivation is considered being a desired motivation. The higher the value of M, the stronger the motivation.

According to Theorem 13.2, in the software engineering context, the rational action of a manager of a group is to encourage individual emotional desire, and the expectancy of the programmer, and to decrease the required effort for the employees by providing additional resources or adopting certain tools.

Corollary 13.3 There are *super strong motivations* toward a resolute goal by a determined expectancy of a person at any cost.

It is noteworthy that motivation is only a potential mental power of human beings, and a strong motivation will not necessarily result in a behavior or action. The condition for transforming a motivation into a real behavior or action is dependent on multiple factors, such as values, social norms, expected difficulties, availability of resources, and the existence of alternative goals.

The motivation of a person is constrained by the attitude and decision making strategies of the person. The former is the internal (subjective) feasibility of the motivation, and the latter is the external (social) feasibility of the motivation as discussed in Section 11.3. Attitude and decision making will be analyzed in the following subsections.

13.3.2.3 Attitudes

As described in the previous section, motivation is the potential power that may trigger an observable behavior or action. Before the behavior is performed, it is judged by an internal regulation system known as the attitude.

The following humor tells an interesting coincidence between attitudes and behaviors:

> Let A, B, ..., Z be assigned a percentage 1%, 2%, .., 26%, respectively. The importance of the following words or phrases may be described by the sum of the percentages of the letters contained in them, i.e.:
>
> Σ ('KNOWLEDGE') = (11+14+15+23+12+5+4+7+5)% = 96%
> Σ ('HARD WORK') = (8+1+18+4+23+15+18+11)% = 98%
> Σ ('ATTITUDE') = (1+20+20+9+20+21+4+5)% = 100%
>
> The above results interestingly "prove" a common saying that *attitude* is more important than *knowledge* or *hard work*.

Psychologists perceive attitude in various ways. R. Fazio describes an *attitude* as an association between an act or object and an evaluation [Fazio, 1986]. A. Eagly and S. Chaiken define attitude as a tendency of a human to evaluate a person, concept, or group positively or negatively in a given context [Eagly and Chaiken, 1992]. More recently, Arno Wittig describes attitude as a learned evaluative reaction to people, objects, events, and other stimuli [Wittig, 2001]. The remainder of this subsection presents a rigorous definition and a formal model of attitude.

Definition 13.26 An *attitude* is a subjective tendency towards a motivation, an object, a goal, or an action based on an intuitive evaluation of its feasibility.

The modes of attitudes can be positive or negative, which can be quantitatively analyzed using the following definition.

Definition 13.27 The *mode of an attitude A* is determined by both an *objective judgment* of its conformance to the social norm *N* and a *subjective judgment* of its empirical feasibility *F*, i.e.:

$$A = \begin{cases} 1, & N = \mathbf{T} \wedge F = \mathbf{T} \\ 0, & N = \mathbf{F} \vee F = \mathbf{F} \end{cases} \tag{13.7}$$

where A = 1 indicates a positive attitude; otherwise, it indicates a negative attitude.

13.3.2.4 The Motivation/Attitude-Driven Behavioral Model

This section discusses the relationship between a set of interlinked perceptual psychological processes such as emotions, motivations, attitudes, decisions, and behaviors. A motivation/attitude-driven behavioral model will be developed for formally describing the cognitive processes of motivation and attitude.

It is observed that motivation and attitude have considerable impact on behavior and influence the way a person thinks and feels [Westen, 1999]. A reasoned action model is proposed by Martin Fishbein and Icek Ajzen in 1975 that suggests human behavior is directly generated by behavioral intensions, which are controlled by the attitude and social norms [Fishbein and Ajzen, 1975]. An initial motivation before the judgment by an attitude is only a temporal idea; with the judgment of the attitude, it becomes a rational motivation [Wang and Wang, 2006; Wang, 2007i], also known as the behavioral intention.

The relationship between an emotion, motivation, attitude, and behavior can be formally and quantitatively described by the *Motivation/Attitude-Driven Behavioral* (MADB) *model* as illustrated in Fig. 13.3 [Wang, 2007i]. In the MADB model, motivation and attitude have been defined in Eqs. 13.6 and 13.7. It is noteworthy that, as shown in Fig. 13.3, a motivation is triggered by an emotion or desire. The rational motivation, decision, and behavior can be quantitatively analyzed according to the following definitions.

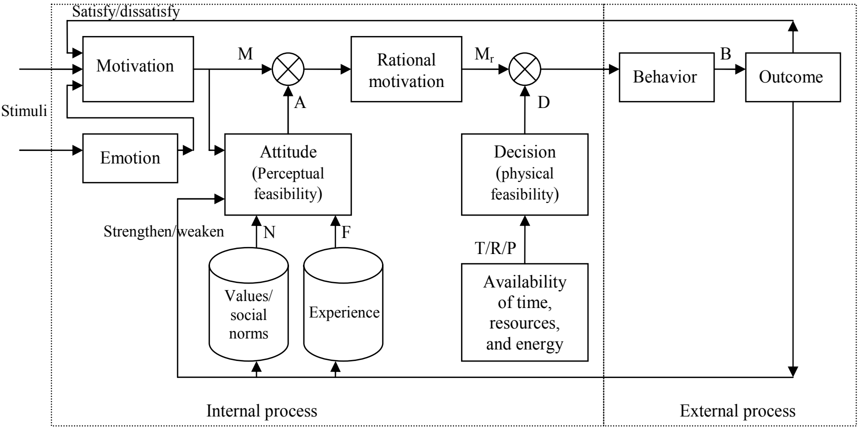


Figure 13.3 The model of motivation/attitude-driven behavior (MADB)

Definition 13.28 A *rational motivation M_r* is a motivation regulated by an attitude A with a positive or negative judgment, i.e.:

$$\begin{aligned} M_r &= M \bullet A \\ &= \frac{2.5 \bullet | E_m | \bullet (E\text{-}S)}{C} \bullet A \end{aligned} \tag{13.8}$$

Definition 13.29 A *decision* for confirming an attitude, D_a, for executing a motivated behavior is a binary choice on the basis of the availability of time T, resources R, and energy P, i.e.:

$$D_a = \begin{cases} 1, & T \wedge R \wedge P = \mathbf{T} \\ 0, & T \vee R \vee P = \mathbf{F} \end{cases} \tag{13.9}$$

Therefore, the formal model of MADB can be described as follows, where a behavior is determined by a product of the strength of motivation and the approval of the decision by a positive attitude.

Lemma 13.5 A *behavior B* driven by a motivation M_r and an attitude is a realized action initiated by a motivation M and supported by a positive attitude A and a positive decision D_a toward the action, i.e.:

$$B = \begin{cases} \mathbf{T}, & M_r \bullet D_a = \dfrac{2.5 \bullet | E_m | \bullet (E\text{-}S)}{C} \bullet A \bullet D_a > 1 \\ \mathbf{F}, & otherwise \end{cases} \tag{13.10}$$

The MADB model presented in Lemma 13.5 and Fig. 13.3 provides a formal explanation of the mechanism and relationship between motivation, attitude, and behavior. The model can be used to describe how the motivation process drives human behaviors and actions, and how the attitude as well as the decision making process help to regulate the motivation and determines whether the motivation should be implemented.

The techniques and models of more rational decision making processes may be referred to Section 11.3 on decision making theories.

13.3.3 COLLECTIVE BEHAVIORS

Organizational psychology studies collective behaviors within groups and organizations, and how structures of them impacts people's behaviors, productivity, and performance. Psychological experiments indicate that individual's behavior may vary in a group influenced by the interactions with other members of the group, which is identified as collective behaviors [Zander, 1979; Wiggins et al., 1994].

Definition 13.30 A *collective behavior* is an integrated behavior of a group in which individuals' behaviors are influenced in different ways by the group.

Collective behaviors are one of the most important social properties of groups and organizations. It is perceived in sociology that any human social behavior may be compared and analyzed against the social norms, which forms a qualitative or quantitative standard for the behavior [Wiggins et al., 1994].

Lemma 13.6 *Individuals' behavior* in the social context is measurable and analytical in term of performance against the social norms.

This subsection describes observable phenomena of collective behaviors such as social conformity, social synchronization, coaction, coordination, groupthink, group polarization, social dilemmas, and social loafing. Two social effects attached to social loafing are the free-rider and sucker effects.

13.3.3.1 Social Conformity

Individuals intend to adjust their behavior or actions, which reflect their thought, to the common goal and norms of a group that they involve and think belong to. This social phenomenon is called conformity.

Definition 13.31 *Social conformity* is a social phenomenon in which an individual's behavior is approached to a social norm or standard in forms of ethical values, role expectations, and laws.

Conformity may be explained by the principle of minimum energy consumption, especially when there is no obvious or intuitive best choice.

13.3.3.2 Social Synchronization

Individuals intend to set their behavior or actions to the timing of the group. This social phenomenon is called synchronization [Wang, 2005k/05*l*].

Definition 13.32 *Social synchronization* is a social phenomenon in which an individual's behavior is timed to a social norm of a group.

Synchronization is a special type of social conformity. Synchronization may be explained by the principles of system synchronization and minimum energy consumption, because synchronization contributes to the maximum output of a group.

13.3.3.3 Coactions

It is found that in temporary social situations and informal groups where no or little coordination is required, people still influence each other when their actions or tasks are identical or have similarity.

Definition 13.33 A *coaction* is a social phenomenon in which the identical or similar actions or tasks are carried out by different individuals with little interaction.

Coaction influences the performance of individuals because it puts the individual in a social context. The phenomena of coaction indicate there is a natural law, as described below, which constrains collective social behaviors of human beings even in a noncohesive social context and a highly temporary and random social relation.

Lemma 13.7 An *autonomous synchronization tendency* between individuals exists in any permanent or temporary social context where people automatically adjust to conjunctive goals and cooperative timing.

The coaction influences on individual's performance can be positive or negative. The former can be a higher expectation, an awareness of difference, and a learning of better practice; while the latter can be a distraction or disappointment.

13.3.3.4 Coordination

Coaction discussed above is an *ad hoc* cooperation in an informal group where there is no common goal as well as predefined means of cooperation and communication. In contrary to coaction, coordination happens in a formal group where common goals as well as means of cooperation and communications exist.

Definition 13.34 A *coordination* is a social phenomenon in which the identical or similar action or task is carried out via intensive interactions between different individuals.

Coordination may influence the performance of individuals dramatically in a group context. Organizational theories of work coordination and efficiencies of group coordination have been extensively studied in Section 8.5, 10.5, and 11.2.3 in engineering science, system science, and management science, respectively.

13.3.3.5 Groupthink

Groupthink and group polarization are two preventable social phenomena of collective behaviors [Janis, 1971].

Definition 13.35 *Groupthink* is a social phenomenon in which the decision-making process within a highly cohesive group is dominated by group consensus that restrains critical thinking of members in the group.

Groupthink may occur in a highly cohesive group where decisions are made by the group and individuals lose their ability to critically evaluate situations or information. Groupthink symptoms identified by Irving Janis in 1971 include illusion of invulnerability, illusion of morality, stereotypes of outsiders, pressure for conformity, self-censorship, and illusion of unanimity [Janis, 1971].

Groupthink acting as a filter of critical ideas may result in another social phenomenon known as group polarization that turns a group to a positive-feedback system.

Definition 13.36 *Group polarization* is a social phenomenon in which group members intend to shift toward the extreme of an already preferred position of the group.

The tendency of group polarization is a powerful positive-feedback mechanism that may result in an instable status of a social unit or system. According to system theories discussed in Chapter 10, the behaviors of a positive-feedback system are sometime unpredictable even destructive. The art of leadership for a group, to some extent, is to prevent the polarization situation from happening.

Lemma 13.8 A weighting system that encourages and appreciates *negative or hesitant feedback* towards a current group's position is a stable system.

The rule of thumb is that, in a group polarization situation, the one who hesitates in the group is perhaps the wiser one. Therefore, Lemma 13.8 indicates that the negative-feedback mechanism is not only suitable for a natural system, but also applicable to social groups and social systems, particularly for software engineering organization.

13.3.3.6 Social Dilemmas

Definition 13.37 The s*ocial dilemma* is a social phenomenon in which members of a group face a conflict choice between the maximization of group's interests by cooperative actions and the maximization of own individual's interests by noncooperative actions.

The collective behaviors of social dilemmas have been identified by many sociologists and social psychologists since 1985 [Komorita and Barth, 1985; Coleman, 1990]. If only egoism is adopted in a society, the social dilemma may exist forever. However, when altruism is recognized to balance egoism as described in Lemmas 13.3, social dilemmas may be resolved systematically.

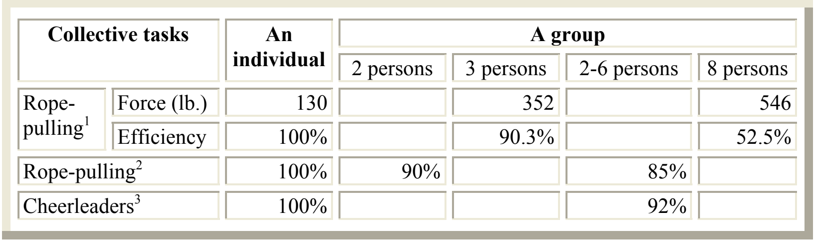
13.3.3.7 Social Loafing

The collective behavior known as social loafing was first identified in Max Ringelmann's experiments on rope-pulling before World War I [Kravits and Martin, 1986]. The same experiment was replicated by Alan Ingham et al. in 1974. This collective phenomenon is then termed as social loafing by Latane and his colleagues in 1979 based on extended studies [Latane et al., 1979; Hardy and Latane, 1986].

Definition 13.38 *Social loafing* is a social phenomenon in which exists the tendency for people to work less hard on a cooperative task in a group than they do individually.

Three independent experiments on the efficiency of coordinated group tasks as shown in Table 13.7 reveal similar patterns of efficiency decreasing when more persons are involved in collective group tasks. These are the main evidences of social loafing. However, it can also be scientifically explained by Theorem 8.4 on coordinate overhead and efficiency.

Table 13.7
Experiments on Efficiency of Coordinated Group Work



Collective tasks		**An individual**	**A group**			
			2 persons	3 persons	2-6 persons	8 persons
Rope-pulling[1]	Force (lb.)	130		352		546
	Efficiency	100%		90.3%		52.5%
Rope-pulling[2]		100%	90%		85%	
Cheerleaders[3]		100%			92%	

Note: Experiment 1 is based on Max Ringelmann [Kravits and Martin, 1986]
Experiment 2 is based on Alan Ingham et al. in 1974.
Experiment 3 is based on Hardy and Latane in 1986.

A typical collective behavior of social loafing is the free-rider effect [Kerr, 1983].

Definition 13.39 The *free-rider effect* is a social phenomenon in which exists the tendency for a member of a group to act noncooperatively based on the assumption that one's individual cooperative action may not be necessary because others will do for the interests of the group.

Another social loafing phenomenon is identified by Jackson and Harkins (1985) known as the sucker effect.

Definition 13.40 The *sucker effect* is a social phenomenon in which exists the tendency for a member of a group to act noncooperatively based on the assumption that others may take advantage of one's individual cooperative contribution to the group.

Social loafing may happen in a group where tasks are parallel allocated and the sum of all parallel capacity is much larger than the workload of the group, for instance, a group of porters and a team of programmers. More rigorous discussion on cooperative work organization [Wang, 2007d] at the system level will be presented in Sections 13.4.2 and 13.5.2.

13.4 Theory of Social Organization

Studies in sociology are mainly empirical and observation-based as described in the preceding sections. This section presents a formal treatment of social organization on the basis of classical thought in sociology. A mathematical model of social organization is developed known as the organization trees. Then, a formal model of socialization is established that explains the inherited interrelationships and interactions between the basic human needs, economic structures, and social types. An important finding based on the formal models of socialization is that the social type and underpinning economic structure of the postindustrial society is transitioning towards a new type of society called the information society driven by the current highest level of unsatisfied human needs. This indicates that software science and engineering will play more and more important roles in human society development and evolution.

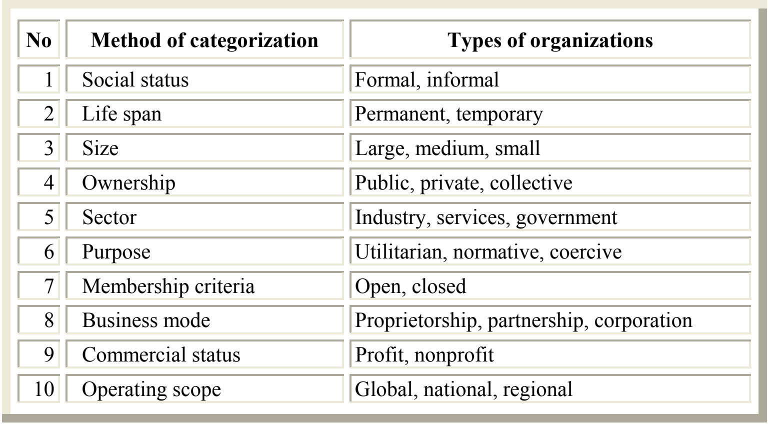
13.4.1 CLASSIC THOUGHT OF SOCIAL ORGANIZATION

From a *geographical* point of view, a society is formed by individuals, families, communities, districts, areas, provinces, and countries from the bottom up. From a *functional* point of view, a society is formed by individuals, groups, organizations, sectors, and the whole economy. This subsection reviews the classical thought on socialization and the conventional forms of social organization.

13.4.1.1 Principles of Social Organization

According to Definition 13.4, an *organization* is a social entity in which a number of groups of people are interconnected and interacting toward common goals. An organization can be formal or informal, permanent or temporary, large or small, public or private, etc. A summary of the types of organizations is provided in Table 13.8.

Table 13.8
Taxonomy of Organizations



No	Method of categorization	Types of organizations
1	Social status	Formal, informal
2	Life span	Permanent, temporary
3	Size	Large, medium, small
4	Ownership	Public, private, collective
5	Sector	Industry, services, government
6	Purpose	Utilitarian, normative, coercive
7	Membership criteria	Open, closed
8	Business mode	Proprietorship, partnership, corporation
9	Commercial status	Profit, nonprofit
10	Operating scope	Global, national, regional

The performance of an organization is determined by both its internal model and its external environment.

Definition 13.41 An *organizational environment* is the external constraints of a society that affect the operation of an organization.

Typical environment constraints for an organization are resources, technologies, politics, population patterns, and the economy.

13.4.1.2 Classic Models of Social Organization

There are various organizational models and methodologies, such as bureaucracy, division of labor, and system organization. System science and system models as discussed in Chapter 10 have provided a formal approach for in social studies, and will be discussed further in Section 13.4.2. This subsection focuses on the conventional approaches of social organization, i.e., bureaucracy and division of labor.

13.4.1.2.1 Bureaucracy

Bureaucracy is originated from the classical forms of public administration in which governments are operated by civil servants known as bureaus.

Definition 13.42 *Bureaucracy* is a classical organizational model of society in which decisions are made from the top-down.

Practical social organizational structures and methodologies were introduced 2,500 year ago in oriental civilizations. The Chinese philosopher and educationist, Confucius (551 – 479BC), proposed that government officers should be systematically selected from the most talented and educated men by civil service examinations. Since then, similar social organizations have been adopted all over the world at almost all levels from governments to businesses. Therefore, Confucius' bureaucracy may be perceived as the earliest sociological inventions of the ancient Chinese civilization.

In modern sociology, Max Webber elicited the basic characteristics of bureaucratic organization in *The Theory of Social and Economic Organization* [Weber, 1947]. He identified the following characteristics: *specialization, hierarchical structures, rule of law, professional competence, impersonality,* and *formal documentation*.

Formal documentation is considered as the central methodology of bureaucracy because written and historical files form a systematic archival system that guides the stable operation of an organization. Based on this it is said that the center of bureaucracy is not people but paperwork [Macionis et al., 1997].

It is noteworthy that bureaucratic organization is designed to improve efficiency. However, it may be alienated in many situations to be inefficient as explained by Parkinson's laws [Parkinson, 1957] and Peter's law below [Peter and Hull, 1969].

Lemma 13.9 The *Parkinson's law* states that work intends to expand to fill the time available for its completion.

However, the difficulty is, in many situations, the effort ad duration of a non-repetitive task is very difficult to be estimated and predicated. In such cases, a payment system based on the completion of the task rather than the time spent on it will be more efficient.

Lemma 13.10 The *Peter's law* states that bureaucrats rise to their level of incompetence in a bureaucratic system.

Peter's law indicates that the maximum competitive level of a person in the hierarchy of a bureaucratic system is n-1, where n is the highest level the person ever achieved without further promotion.

The alienation of bureaucratic organizations is an example of system dissimilation in sociology as presented in Theorem 10.12.

13.4.1.2.2 Division of Labor

Work organization by division of labor and specialization was adopted in bureaucracy [Confucius, 551-479BC]. Division of labor was introduced into industry and mass manufacturing during the industrial revolution [Smith, 1776; Tayler, 1911], which forms the important characteristics of industrialization [Warner and Low, 1947]. Instead of working in a cottage economy fashioning a product through all the processes individually, industrialized mass production demands specialization. The advantages of division of labor are higher productivity and lower entry skills. It is formally presented in Theorem 11.2 that when people are repetitively working on subtasks in a process, the productivity can be greatly improved [Wang, 2005k].

Mcdonaldization is identified as a modern type of division of labor in work organization. The basic organizational principles revealed from Mcdonaldization are *efficiency*, *quantification*, *uniformity*, and *automation* [Ritzer, 1983/93]. Ritzer observed that the most unreliable element in the Mcdonaldization process is human beings, because people are unstable, sometimes letting their minds wander, or simply trying something nonstandard. This factor may be eliminated by using automatic tools and standard process regulations.

Sectors are a macro type of division of labor in a society, where labors are allocated by their organizations oriented to different kinds of products or services. Professionalism is another type of division of labor in postindustrial societies, where people are specialized in various highly skilled disciplines.

13.4.2 THE FORMAL MODEL OF SOCIAL ORGANIZATION

Empirical and practical social organizations have been formed as results of historical, political, and economical processes. However, a few natural laws had been sought in sociology in order to understand the fundamental constraints of human societies. Toward this aim, this subsection presents a set of formal sociological models on the basis of system theory and the *System Organization Tree* (SOT) as developed in Section 10.3.5. A rigorous treatment of social organization in engineering is developed. Based on the sociological models, the theories and laws behind coordinative work organizations at the social organization and the top system level are revealed, which will be used as the foundation for large-scale software engineering project organization.

13.4.2.1 The Formal Organization Tree

The most common organizational structures in science and nature are tree-type architectures [Pattee, 1978; Wang, 2005k]. The complete *n*-nary tree has been described in Section 10.3.5 as a normalized tree in which each node of it can have at most *n* children, and all subtrees and nodes except the the rightmost subtrees and leaves have the maximum number of possible nodes. In other words, a tree that is said to be *complete* means that all levels of the tree have been allocated the maximum number of possible nodes; only the leave-level nodes and the rightmost subtrees may be exceptional.

Definition 13.43 A *normalized Organization Tree* (OT_n) is a complete *n*-nary tree in which all leave nodes represent *employees* and the remainder represent *managers*. When the leaves (employees) do not reach in the maximum possible numbers in the OT, the right most leaves and associated subtrees will be left open.

A ternary OT*,* OT_3*,* is given in Fig. 13.4. The important properties of OT$_s$ have been studied in Section 10.3.5, particularly Corollary 10.6 [Wang, 2005k/05*l*], which are recited in Lemma 13.11 below for self-containment of this section on formal organization trees.

One of the advanced characteristics of OT$_s$ are that their structural and functional properties are highly predictable as stated in Corollary 13.4.

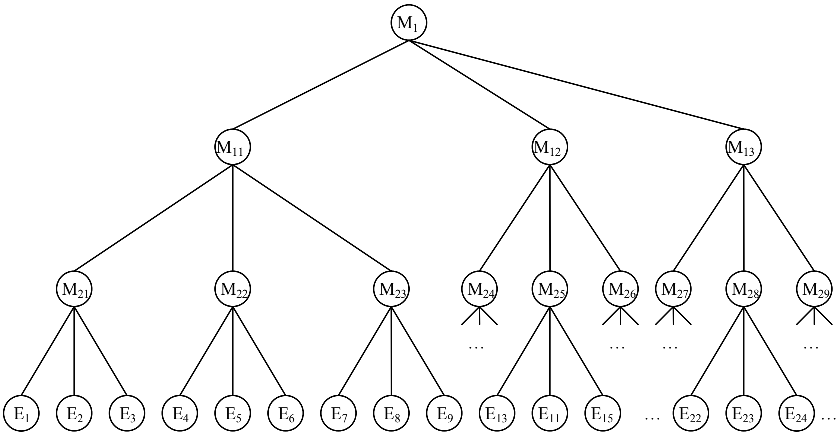


Figure 13.4 A normalized organization tree (n = 3)

Lemma 13.11 A normalized n-nary *organization tree* OT_n with the total number of leave nodes N_e, possesses the following properties:

a) The optimal number of fan-out of any node $\overline{n}_{fo}$:

$$\overline{n}_{fo} = n \tag{13.11}$$

b) The maximum number of nodes at a given level k, n_k:

$$n_k = n^k \tag{13.12}$$

c) The depth of the OT, d:

$$d = \left\lceil \frac{\log N_e}{\log n} \right\rceil \tag{13.13}$$

d) The maximum number of nodes in OT, N_{OT}:

$$N_{OT} = \sum_{k=0}^{d} n^k \tag{13.14}$$

e) The maximum number of *employees* (on all leaves) in OT, N_e:

$$N_e = n^d \tag{13.15}$$

f) The maximum number of *managers* (nodes except all leaves) in OT, N_m:

$$N_m = N_{OT}\text{-}N_e = \sum_{k=0}^{d\text{-}1} n^k \tag{13.16}$$

Corollary 13.4 An $OT(\overline{n}_{fo}, N_e)$ is fully determinable *iff* its number of employees (leaves) N_e and the optimal number of fan-out of groups $\overline{n}_{fo}$ are given.

Definition 13.44 The *organizational overhead* $r_{OT}(n)$ of an n-nary organization tree OT_n is determined by the ratio between the number of management N_m and the total members of OT_n, N_{OT}, i.e.:

$$r_{OT}(n) = \frac{N_m}{N_{OT}} = \frac{\sum_{k=0}^{d\text{-}1} n^k}{\sum_{k=0}^{d} n^k} \approx \frac{1}{n} \tag{13.17}$$

Definition 13.45 The *organizational efficiency* $e_{OT}(n)$ of an n-nary organization tree OT_n is determined by the ratio between the number of employees N_e and the number of management N_m, i.e.:

$$e_{OT} = \frac{N_e}{N_m} = \frac{n^d}{\sum_{k=0}^{d\text{-}1} n^k} \approx n \tag{13.18}$$

The theory of OT provides a mathematical model for formally analyzing the architectures of social organizations and their efficiency. The applications of OT are demonstrated in the following subsection.

13.4.2.2 Formal Models of Social Organization

Lemma 13.12 An *organization is needed* when the size of a group is too large that it exceeds the optimal size of the group, and therefore is no longer efficient.

Lemma 13.12 indicates that when the importance of *organizational efficiency* overpasses the initial purposes of a group for sharing resources and functions, an organization with multiple groups is required.

Theorem 8.10 as well as Corollaries 8.41 and 8.7 developed in Chapter 8 on optimal project team organization reveal there are natural laws constraining the size of groups for a given workload. Based on Theorem 8.10, the optimal sizes of groups with acceptable efficiency are constrained by the following theorem.

The 47th Law of Software Engineering

Theorem 13.3 The *organizational coordination efficiency* states that the natural constraints for social organization that forces the architecture of large groups to be evolved and adapted to tree-form hierarchical structures in an organization is the need to maintain acceptable coordinating efficiency at each level of the organization tree.

This law forces the architecture of large groups to be reformed to tree-type hierarchical structures of groups in order to maintain acceptable coordinating efficiency [Wang, 2005k/05*l*/07d].

Corollary 13.5 The *optimal architecture* of large-scale organizations, by which an optimized structure can be maintained at each level of its hierarchy, is an OT where the *average optimal fan-out* of a node $\overline{n}_{fo}$ or the size of the group n_G, is larger than 3 and smaller than 10, i.e.:

$$3 \leq \overline{n}_{fo} = n_G \leq 10 \qquad (13.19)$$

where $\overline{n}_{fo}$ is the optimum labor allocation.

Corollary 13.5 can be proven by Theorem 8.10. A set of typical data between the expected duration and expected workload against different labor allocations is given in the *pigeon diagram* as shown in Fig. 8.5, where the trends of expected project durations against different labor allocations and the interpersonal coordination rate r in the group are illustrated.

Generally, for a certain ideal workload $W_I(r)$, the corresponding optimal labor allocation L_0 and the shortest duration T_{min} can be determined according to Theorem 8.7. Theorem 8.7 provides a solution to determine the average optimal fan-out $\overline{n}_{fo}$ of OTs. On the basis of Theorem 8.7, an optimal organization tree can be determined according to the following corollary.

Corollary 13.6 The *optimal architecture of a normalized organization tree*, OT(r, N_e) for an organization with N_e members on the first line (the leaves of the tree) and a given average interpersonal coordination rate r is determined as follows:

a) The average optimal fan-out of the OT, $\overline{n}_{fo}$:

$$\overline{n}_{fo} = n_G = L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil, \; r \neq 0 \quad [P] \qquad (13.20)$$

b) Number of optimal groups (subprojects) of the OT, N_G:

$$N_G = \left\lceil \frac{N_e}{\overline{n}_{fo}} \right\rceil \qquad (13.21)$$

c) Depth of the OT, d_{OT}:

$$d_{OT} = \left\lceil \frac{\log N_e}{\log \overline{n}_{fo}} \right\rceil = \left\lceil \frac{\log N_e}{\log L_o} \right\rceil \qquad (13.22)$$

Example 13.1 Given an organization with N_e = 100 employees in the first line and the work cooperation rate r = 10%, analyze the optimal architecture of the organization tree OT_1 for this organization.

According to Corollary 13.6, the optimal architecture of the organization tree can be derived as follows:

(a) The average optimal fan-out of OT_1:

$$\overline{n}_{fo} = L_0 = \left\lceil \frac{1.414}{\sqrt{0.1}} \right\rceil = 5.0 \text{ [P]}$$

(b) Number of optimal groups of OT_1:

$$\begin{aligned} N_G &= \left\lceil \frac{N_e}{\overline{n}_{fo}} \right\rceil \\ &= \left\lceil \frac{100.0}{5.0} \right\rceil = 20.0 \end{aligned}$$

(c) Depth of OT_1:

$$\begin{aligned} d_{OT} &= \left\lceil \frac{\log N_e}{\log\, L_0} \right\rceil \\ &= \left\lceil \frac{\log 100.0}{\log\, 5.0} \right\rceil = \left\lceil \frac{2.0}{0.70} \right\rceil = 3.0 \end{aligned}$$

Referring to Fig. 13.4, readers may draw a diagram of the organization tree based on the derived characteristics of OT_1.

Example 13.2 What is the optimal architecture of the OT of a large organization with N_e = 100,000 employees in the first line given L_0 = 10.0P?

According to Corollary 13.6, the characteristic of OT_2 can be derived as follows:

a) The average optimal fan-out of OT_2: $\overline{n}_{fo} = L_0 = 10.0\text{P}$

b) Number of optimal groups of OT_2: $N_G = 10^4$

c) Depth of OT_2: $d_{OT2} = 5$

Example 13.3 Given a country with a billion people in the working force at the leave level, i.e., $N_e = 1 \bullet 10^9$, what is the optimal architecture of the organization tree OT_3 when the average optimal fan-out $\overline{n}_{fo}$ = 10? What is the number of managers required for optimal organization in OT_3? What is the ratio or overhead of management of OT_3?

According to Corollary 13.6, the characteristic of OT_3 can be derived as follows:

a) Number of optimal groups of OT_3: N_G: = 10^8

b) Depth of OT_3: d_{OT3} = 9

c) Number of managers of OT_3: According to Eq. 13.16,

$$N_m = \sum_{k=0}^{d\text{-}1} \overline{n}_{fo}{}^k = \sum_{k=0}^{8} 10^k \approx \frac{10^9}{\overline{n}_{fo}} = \frac{10^9}{10} = 1 \bullet 10^8$$

d) Ratio of management of OT_3:

$$r_{OT3} = \frac{N_m}{L} \approx \frac{1}{\overline{n}_{fo}} = \frac{1}{10}.$$

It is noteworthy that the more advance development in a society, the smaller the optimal group size $\overline{n}_{fo}$ due to higher coordination among people; therefore, the larger the ratio of management in the total population.

13.4.2.3 Coordinative Work Organization

The preceding subsections discussed the optimal structures of social organizations in the form of normalized organization trees. This subsection describes the structures of work organization and allocation on the basis of the theory of the maximum output of abstract system in Section 10.5.4. Major methods of work organizations are serial and parallel structures, as well as their combinations known as hybrid structures.

13.4.2.3.1 Serial Structures

Definition 13.46 A *serial work organization* is a work allocation structure in which a given work is decomposed into a series of tasks and each task is allocated to a person or a group.

The serial structure of work organization can be illustrated in Fig. 13.5, where W and W_{os} are the input and output work of a serial system.

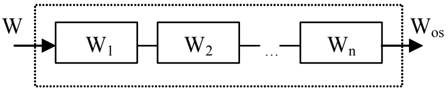


Figure 13.5 The serial structure of work organization

Lemma 13.13 The *output work of a serial work organization W_{os}* equals to the minimum work done by the least capable unit W_{min}, i.e.:

$$\begin{aligned} W_{os} &= \min(W_i \mid 1 \le i \le n) \\ &= W_{\min} \end{aligned} \tag{13.23}$$

Corollary 13.7 The *capacity of a serial work system* is determined by the least capable unit W_{min} known as the *bottleneck.*

Corollary 13.8 The key to *optimal serial work organization* is there is no bottleneck in the social system, i.e.:

$$W_{os} = W = W_i = W_{\min} \tag{13.24}$$

13.4.2.3.2 Parallel Structures

Definition 13.47 A *parallel work organization* is a work allocation structure in which a given work is done repetitively or jointly by multiple persons or group.

The parallel structure of work organization can be illustrated in Fig. 13.6, where W and W_{op} are the input and output work of a parallel system.

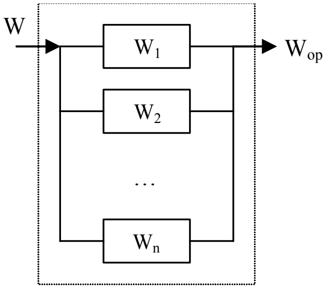


Figure 13.6 The parallel structure of work organization

Lemma 13.14 The *output work of an n parallel work organization W_{op}* is equal to the sum of work done by each unit W_i, i.e.:

$$W_{op} = \sum_{i=1}^{n} W_i \tag{13.25}$$

Corollary 13.9 The *capacity of a parallel work system* is dominated by the most capable unit W_{max} known as the *critical unit.*

Corollary 13.10 The key to *optimal parallel work organization* is not to over-allocated work capacity in any unit of the system, i.e.:

$$W_{op} = \sum_{i=1}^{n} W_i \tag{13.26}$$

Recalling the discussion on the phenomena of social loafing in Section 13.3.3, it may be seen that the parallel organization of work in a group may allow it to happen. This is formally described as in Corollary 13.11.

Corollary 13.11 The *necessary condition of social loafing* is that a group is parallel organized, where the output work of a unit W_j is zero, i.e.:

$$W_{op}\,' = (\sum_{i=1}^{n} W_i) - W_j \tag{13.27}$$

where 1 ≤ j ≤ n.

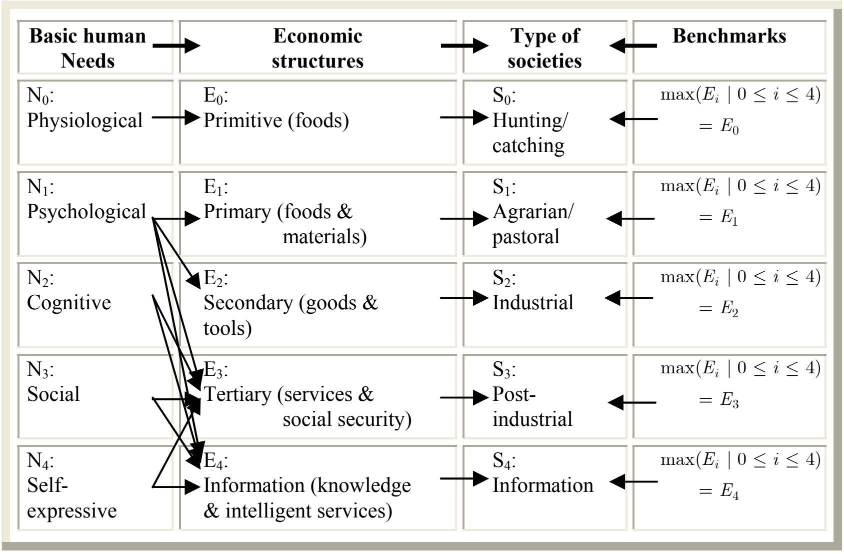
Complex work organizations may adopt a hybrid structure of serial and parallel organizations.

13.4.3 THE FORMAL MODEL OF SOCIALIZATION

Not only task performances and engineering organizations are influenced by the fundamental human factors and needs. The forms of societies and their organizations are indirectly determined by the basic human needs as well.

There are various types of societies corresponding to different economic structures and their levels of development. The relationships between the basic human needs, economic structures, and social types can be explained by the following model.

Definition 13.48 The *Formal Socialization Model* (FSM) is a relational model that describes the relationships between the basic human needs, economic structures, and social types, as shown in Fig. 13.7.



Basic human Needs	**Economic structures**	**Type of societies**	**Benchmarks**
N_0: Physiological	E_0: Primitive (foods)	S_0: Hunting/ catching	$\max(E_i \mid 0 \le i \le 4) = E_0$
N_1: Psychological	E_1: Primary (foods & materials)	S_1: Agrarian/ pastoral	$\max(E_i \mid 0 \le i \le 4) = E_1$
N_2: Cognitive	E_2: Secondary (goods & tools)	S_2: Industrial	$\max(E_i \mid 0 \le i \le 4) = E_2$
N_3: Social	E_3: Tertiary (services & social security)	S_3: Post-industrial	$\max(E_i \mid 0 \le i \le 4) = E_3$
N_4: Self-expressive	E_4: Information (knowledge & intelligent services)	S_4: Information	$\max(E_i \mid 0 \le i \le 4) = E_4$

Figure 13.7 The Formal Socialization Model (FSM) of human societies

The FSM model reveals that natural rules exist between the types of society, the underlying economic structures, and the dominant sector in the economy, because both social architectures and economic structures are driven by the current level of predominantly unsatisfied fundamental human needs.

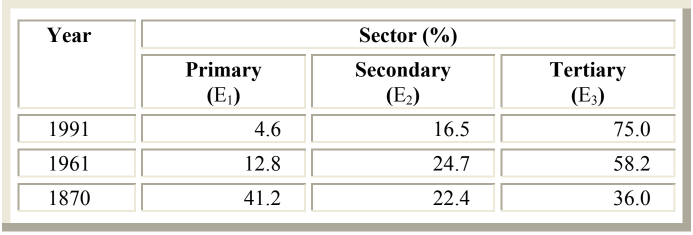
There are various types of societies corresponding to different economic structures and their levels of development. The relationships between the basic human needs, economic structures, and social types can be described below by Lemma 13.15.

Lemma 13.15 The *type of society* $T_{S_i}, 0 \le i \le 4$, is determined by the dominated sector T_{E_i} of the corresponding economic structure, which is constituted by the current level of predominately unsatisfied human needs $\overline{T_{N_i}}, 0 \le i \le 4$, i.e.:

$$\begin{aligned} T_{S_i} &= \max\,(T_{E_i}) \\ &= \max\,(\overline{T_{N_i}}),\ 0 \le i \le 4 \end{aligned} \qquad (13.28)$$

Example 13.4 According to Statistics Canada, Catalogue Nos. 93-151 (1986) and 93-327 (1991), the Canadian economic structures during 1870 to 1991 are shown in Table 13.9.

Table 13.9
The Canadian Statistics of Social Development



Year	Sector (%)		
	Primary (E_1)	Secondary (E_2)	Tertiary (E_3)
1991	4.6	16.5	75.0
1961	12.8	24.7	58.2
1870	41.2	22.4	36.0

Based on Lemma 13.15, it can be determined that the dominant type of economy and corresponding types of society in years 1870, 1961, and 1991 in Canada are as follows:

a) In 1870:

$$\begin{aligned} T_{S_i}(1870) &= \max\ (T_{E_i}),\ 0 \leq i \leq 4 \\ &= E_1\ \text{(Primary economy dominated)} \\ &\Rightarrow S_1\ \text{(Argrarian society)} \end{aligned}$$

b) In 1961:

$$\begin{aligned} T_{S_i}(1961) &= \max\ (T_{E_i}),\ 0 \leq i \leq 4 \\ &= E_3\ \text{(Tertiary economy dominated)} \\ &\Rightarrow S_3\ \text{(Postindustrial society)} \end{aligned}$$

c) In 1991:

$$\begin{aligned} T_{S_i}(1991) &= \max\ (T_{E_i}),\ 0 \leq i \leq 4 \\ &= E_3\ \text{(Stronger tertiary economy dominated)} \\ &\Rightarrow S_3\ \text{(Postindustrial society, and a trend} \\ &\qquad\quad \text{to the information society)} \end{aligned}$$

It is noteworthy that the trend of socialization according to Lemma 13.15 may be predicated that the emerging information-based economy will drive the society into a new era, the *information society*, where the major sector of the information society will be information processing related and intelligent services providing professions.

Corollary 13.12 The *next type of society* after post-industrialization is the information society driven by the current level of predominantly unsatisfied social and self-expressive needs and the underlying information-oriented economy.

The fundamental driving forces for this trend are that the higher level human needs built upon the satisfied lower-level ones, such as cognitive (N_2), social (N_3), and self-expressive (N_4) needs, will be the new focus of post-industrialized societies. Because all N_2 through N_4 needs are based on information and intelligent services when the material level needs are satisfied, the form of economy and type of society will be evolved into the information-oriented society naturally.

13.5 Sociology and Software Engineering

As discussed in Section 13.4.3 the type of society and the associated economy is evolving towards the information-oriented ones in order to satisfy the higher-level human needs in the postindustrial society. As a logical consequence, the fundamental theories and techniques for information processing, such as software engineering, computing, information science, and cognitive informatics, will be increasingly important over time.

This section explores the applications of sociology in software engineering, and discusses how software engineering may be scientifically and efficiently organized.

13.5.1 SOCIAL ORGANIZATION OF SOFTWARE ENGINEERING

Theorems 13.1 through 13.3 developed in preceding sections provide the theoretical foundation for software engineering organization. Optimal social and project organization of a software enterprise can be designed and implemented using the quantitative analysis techniques of organization trees.

13.5.1.1 The Role of the Information Economy in Postindustrial Societies

The top-level application of sociology in software engineering is the identification of the central role of all information-related theories and techniques in modern societies.

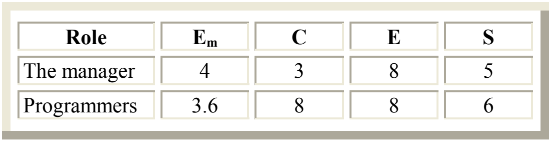
According to the formal socialization model (FSM), the information sector will be the fourth main sector following the primary, secondary, and ternary sectors, in order to meet the current highest level of unsatisfying human needs. Therefore, the new economy is information- and knowledge-based economy, and the future type of society after the post-industrialization is the information society. Thus, it can be predicated that software engineering and computing will play an increasingly important role in the transition of the new economy.

13.5.1.2 Maximizing Strengths of Individual Motivations in Software Engineering

Sociology provides a rich theoretical basis for perceiving insights into the organization of software engineering. It is noteworthy that in a software organization, according to Theorem 13.2, the strength of a motivation of individuals *M* is proportional to both the strength of emotion E_m and the difference between the expectancy *E* and the current status *S* of a person. At the same time, it is inversely proportional to the cost to accomplish the expected motivation *C*. The job of management at different levels of the organization tree is to encourage and improve E_m and *E*, and to help employees to reduce *C*.

Example 13.5 In software engineering project organization, the manager and programmers may be motivated to the improvement of software quality in different extents. Assuming the following factors as shown in Table 13.10 are collected from a project on the strengths of motivations to improve the quality of a software system, analyze how the factors influence the strengths of motivations of the manager and the programmers, respectively.

Table 13.10
Motivation Factors of a Project



Role	E_m	C	E	S
The manager	4	3	8	5
Programmers	3.6	8	8	6

According to Theorem 13.2, the strengths of motivations of the manager M_1 and the programmer M_2 can be estimated using Eq. 13.6, respectively:

$$\begin{aligned} M_1(manager) &= \frac{2.5 \bullet \ | E_m | \ \bullet (E\text{-}S)}{C} \\ &= \frac{2.5 \bullet 4 \bullet (8 - 5)}{3} \\ &= 10.0 \end{aligned}$$

and

$$\begin{aligned} M_2(programer) &= \frac{2.5 \bullet 3.6 \bullet (8 - 6)}{8} \\ &= 2.3 \end{aligned}$$

The results show that the manager has much stronger motivation to improve the quality of software than that of the programmer in the given project. Therefore, the rational action of the manager is to encourage the expectancy of the programmers or to reduced the required effort for the programmers by providing additional resources or adopting additional development tools.

13.5.1.3 Social Environments of Software Engineering

According to social psychology discussed in Section 13.3, social environment, such as culture, ethical norms, and attitude, greatly influences people's motivation, behavior, productivity, and quality toward coordinative work. The chain of individual motivation in a software organization can be illustrated as shown in Fig. 13.8.

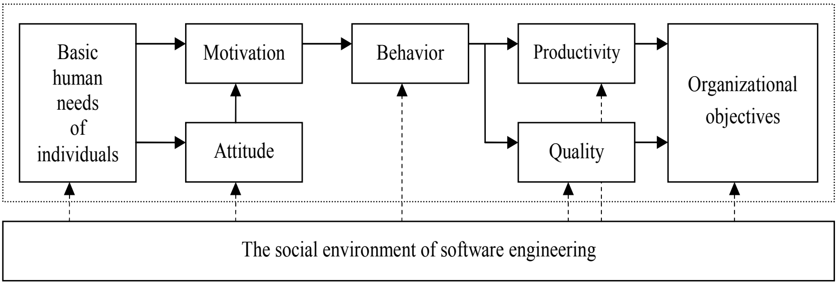

Basic human needs of individuals
Motivation
Attitude
Behavior
Productivity
Quality
Organizational objectives
The social environment of software engineering

Figure 13.8 The chain of motivation in a software organization

Cultures and values of a software development organization help to establish a set of ethical principles or standards shared by individuals of the organization for judging and normalizing social behaviors. The identification of larger set of values and organizational policy towards social relations may be helpful to normalize individual and collective behaviors in the software development organization that produces information products for a global market.

Another condition for supporting creative work of individuals in a software development organization is to encourage diversity in both ways of thinking and work allocation. It is observed in social ecology that a great diversity of species and a complex and intricate pattern of interactions among the populations of a community may confer greater stability on an ecosystem.

Definition 13.49 *Diversity* refers to the social and technical differences of people in working organizations.

Diversity includes a wide range of differences between people such as those of *race, ethnicity, age, gender, disability, skills, educations, experience, values, native language,* and *culture*.

The principle of system mutation indicates that if the number of components of a system reaches a certain level – the critical mass, then the functionality of the system may be dramatically increased as stated in Theorem 10.5 in Section 10.5.2. That is, the increase of diversity in a system is the condition to realize the system fusion effect, which results in a totally new system.

Lemma 13.16 The *diversity lemma* states that the more diverse the workforce in an organization (particularly the creative software industry), the higher the opportunity to form new relations and connections that leads to the gain of the system fusion effect.

13.5.1.4 Ergonomics for Software Engineering

The term *ergonomics* was proposed by Wojciech Jastrzebowski in 1857. It is derived from the Greek words *ergos* (work) and *nomos* (study of) [Salvendy, 2006].

Definition 13.50 *Ergonomics* is a branch of engineering and behavioral science that studies human efficiency in different working environments.

Ergonomics is the science of work such as abilities, limitations, and characteristics of human beings and their adaptation to the working environment. Ergonomics can be divided into two overlapped branches known as the *industrial ergonomics* and *human factors*. The former focuses on engineering biomechanics, or the physical aspects of human capabilities, such as force, posture, and repetition. The latter study engineering psychology, or the mental aspects of human capability, such as the strengths and weaknesses of human brain in the working environment.

In general, ergonomics aims at fitting tasks, processes, tools, and environments to people, in order to improve productivity, quality, and safety [Chaffin and Andersson, 1984]. Applications of ergonomics can be found in a wide range of engineering disciplines, industrial psychologists, occupational physicians, industrial hygienists, safety monitors, and quality engineers.

Recent emphases of human factors research have been put on improving the ways of information usages known as information design.

Definition 13.51 *Information design* is a branch of ergonomics that studies the design of signs, symbols, and instructions of information and software systems in order to enable their meaning can be quickly and safely comprehended.

Virtually everyone has experienced the frustration of using computer software that doesn't work the way they expect it to. For the majority of end users of computer programs, if the system is not working they have no recourse but to call for technical help, or find creative ways around system limitations, using those parts that are usable, and circumventing the rest or increasing stress levels by using a substandard system. Often the problems in systems could have been avoided, if a more complete understanding of the users' tasks and requirements had been present from the start. The development of easily usable human-computer interfaces is a major issue for ergonomists today.

13.5.2 THEORY FOR LARGE-SCALE SOFTWARE ENGINEERING PROJECT ORGANIZATION

Comparatively analyzing the results shown in Examples 12.6 and 12.5, it can be observed that for large-scale software engineering projects as given in Example 12.6, a more efficient organizational form is to break the project up into *n* lightly-coupled parallel subprojects as that of Example 12.5, where $n = 5$, and each subproject is dealt with by an independent subgroup. Based on this organizational strategy, an *n*-fold shorter project duration may be achieved under the same level of workload and project costs. This leads to an important law of software engineering organization as stated in Theorem 13.4.

The 48th law of software engineering as stated in Theorem 13.4 presents the theoretical foundation of the empirical principles of *division and* *conquer*, *modularization,* and *system decomposition* in software engineering. It is a natural extension of Theorem 8.10 in the contexts of large-scale systems and social environments, which completed the entire theory of coordinative work organization in system/management science in general and in software engineering in particular. Theorem 13.4 also indicates that the much balanced the partitions among the subsystems, the more efficient the gain for reducing project duration in large-scale software development in software engineering.

The following examples demonstrate how Theorem 13.4 explains the differences between the organizational forms of the structured multi-group projects and the unstructured large-single-group projects in software engineering.

The 48th Law of Software Engineering

Theorem 13.4 *Time-oriented optimization for large-scale project organization* states that in order to further reduce the shortest duration T_{min} of an entire large-scale project constrained by Theorem 8.7, the optimal form of organization is to evenly partition the whole project into n lightly-coupled parallel subprojects that may be conducted by independent groups with a shorter duration T^i_{min}, $1 \le i \le n$, so that an average n-fold time deduction can be gained, i.e.:

$$\begin{aligned} \overline{T^i{}_{\min}} &= \frac{1}{n}\sum_{i=1}^{n} T^i{}_{\min} \\ &= \frac{1}{n} T_{\min} + \varpi \end{aligned} \qquad (13.29)$$

where $\overline{T^i{}_{\min}}$ is the average shortest duration of all subsystem, and ϖ is a positive overhead needed for system integration or composition.

Example 13.6 Assume the large-scale project as given in Example 12.6 is divided into five lightly-coupled parallel subprojects, and each subproject can be conducted independently by an individual subgroup that obeys the generic constraint on group size in coordinative work (Theorem 8.10). Then, discuss the effects and impacts of this organizational form on project duration and costs.

When the large-scale project given in Example 12.6 is evenly partitioned into five parallel subjects, the analysis results of each of them are the same as those obtained in Example 12.5. The results are closely met with the predications as those directly derived on the basis of Theorem 13.4, i.e.:

$$\begin{aligned}\overline{T^i{}_{\min}} &= \frac{1}{n}T_{\min} + \varpi \\ &= \frac{1}{5}\bullet 14.4 + \frac{1}{10}\bullet 14.4 \\ &= 2.9 + 1.4 \\ &= 4.3 \ \ [\mathrm{M}]\end{aligned}$$

where the integration overhead is assumed as 10% of the shortest project duration of the whole project before partition.

The above comparative study also demonstrates that the linear partition of a large-scale project and the form of *n*-group OT organization may reduce the project duration up to *n* times without change the entire project workload and total costs. In other words, OT/SOT is an ideal organizational mechanism for large-scale project and society-level organizations, which enables labor to be traded with time in the system infrastructure of OT/SOT.

Therefore, Law 25 (Theorem 8.7) and Law 48 (Theorem 13.4), or the *coordinative work organization theory* and *the social system organization theory* in terms of OT/SOT, provide a complete theoretical framework for explaining the age-old *mythical man-month* at the group level and the system level, respectively.

Corollary 13.13 Large-scale projects should always be organized as a structured system in the form of OT or SOT because it enables complicated work to be done in a linear predictability in terms of effort and costs while gaining greatly for up to *n*-fold reduction of project duration.

Corollary 13.14 In large-scale projects organization, project duration cannot be reduced in a single-group structure by increasing the size of the group contingently, because nonoptimal man-powered groups against the laws of group size constraints (Theorems 8.7 and 8.10) will result in an exponential incremental of project duration, expected workload, and costs.

Example 13.7 Suppose, in order to reduce the project duration, the large-scale project given in Example 12.6 is not partitioned into multiple subprojects as Theorem 13.4, Corollary 13.13, and Corollary 13.14 suggested, but is subjectively organized with an extended group of L = 25.0P deviated from the optimal allocation L_0 = 5.0P. What would be the consequences of this irrational decision?

According to the FEMSEC model and Example 12.6, the above project can be analyzed below:

The expected duration *T* can be estimated using Eq. 12.56, i.e.:

$$\begin{aligned} T &= \frac{1}{2}W_1(rL - r + \frac{2}{L}) \\ &= 0.5 \bullet 40.0\ (0.08 \bullet 25.0 - 0.08 + 2/25.0) \\ &= 40.0 \quad [M] \end{aligned}$$

The expected workload can be estimated using Eq. 12.57, i.e.:

$$\begin{aligned} W &= L \bullet T \\ &= 25.0 \bullet 40.0 \\ &= 1,000.0 \quad [\text{PM}] \end{aligned}$$

The total cost can be estimated using Eq. 12.58, i.e.:

$$\begin{aligned} C &= W \bullet C_L \\ &= 1,000.0 \bullet 80,000/12 \\ &= \$6,666,667.00 \quad [\$] \end{aligned}$$

According to Corollary 8.5, the wasted effort and budget would be as high as the follows, respectively:

$$\begin{aligned} \Delta W &= W - W_{\text{exp}} \\ &= 1,000.0 - 72.0 \\ &= 928.0 \quad [\text{PM}] \end{aligned}$$

$$\begin{aligned} \Delta C &= C - C_{\text{exp}} \\ &= 6,666,667.0 - 480,000.0 \\ &= 6,186,667.0 \quad [\text{PM}] \end{aligned}$$

These results have already indicated a mission impossible for this given project in the irrational organization form. Nevertheless, the results could have been worse if *r* of the project were higher. This is why it is identified in

Theorem 1.5 that the *key problems* of software engineering are not only pure technical issues rather than organizational and cognitive issues.

13.5.3 THE HUMAN FACTORS IN SOFTWARE ENGINEERING

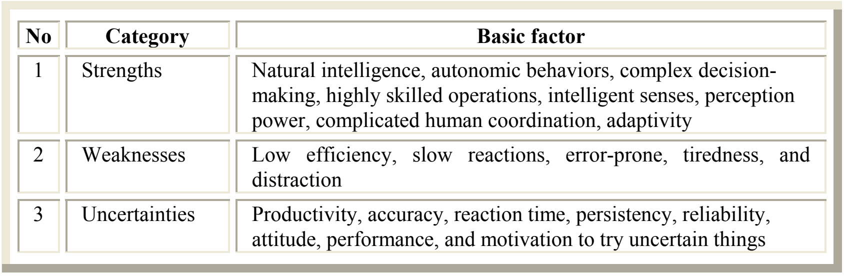
The human factors are not only a constantly important constraint in almost all disciplines of science and engineering, but also the most active and dynamic factors to be considered. Nevertheless, human beings themselves are directly the object of study in a number of disciplines such as psychology, cognitive science, ergonomics, sociology, cognitive informatics, medical science, neuroscience, and natural intelligence.

Definition 13.52 The *human factors* are the roles and effects of humans in a system that introduces additional strengths, weaknesses, and uncertainty.

13.5.3.1 Taxonomy of Human Factors

There are numerous human factors identified in science, engineering, sociology, psychology, and everyday life. The taxonomy of human factors in engineering can be classified into human strengths, weaknesses, and uncertainties as shown in Table 13.11.

Table 13.11
Taxonomy of Human Factors



No	Category	Basic factor
1	Strengths	Natural intelligence, autonomic behaviors, complex decision-making, highly skilled operations, intelligent senses, perception power, complicated human coordination, adaptivity
2	Weaknesses	Low efficiency, slow reactions, error-prone, tiredness, and distraction
3	Uncertainties	Productivity, accuracy, reaction time, persistency, reliability, attitude, performance, and motivation to try uncertain things

Widely varying productivities are one of the major factors of human beings, particularly in creative work such as software development. It is found that the productivity of human creative work is conservative. That is, the creative productivity is independent from languages and processes; however, it depends on human cognitive, physiological, and psychological capabilities.

Definition 13.53 *Conservative productivity* is a basic constraint of software engineering due to cognitive complexity and due to the cognitive mechanism in which abstract artifacts need to be represented physiologically in the brain via growing synaptic neural connections.

Human psychology, such as motivations and attitudes, influences human factors very much [Fischer et al., 1990; Eagly and Chaiken, 1992; Wang, 2007i]. The great variety of human psychological and cognitive capacity influenced by motivations, attitudes, focuses, and attentions are the major reasons of human uncertainties on productivity, accuracy, reaction time, persistency, and task performance.

13.5.3.2 Types of Human Errors

It is a fact that people do make mistakes, and fortunately, most of them may be corrected by additional undo or redo actions. However, in safety- or mission-critical contexts, the impact of human errors can be catastrophic, such as in the nuclear and chemical industries, rail and sea transports, and aviation.

Definition 13.54 A *human error* is an error of human caused by wrong actions and inappropriate behaviors.

Christopher Wickens and his colleagues identified a long list of reasons that cause operator errors in systems, such as inattentiveness, poor work habits, lack of training, poor decision making, personality traits, social pressure [Wickens et al., 1998]. The Systematic Human Error Reduction and Prediction Approach (SHERPA) proposed by D. Embry in 1986 identified sixteen potential psychological errors [Embry, 1986]. J. Reason developed a similar system in 1987 known as the Generic Error Modeling System (GEMS) [Reason, 1987/90]. The set of human behavioural errors identified in SHERPA are as follows:

- Action omitted
- Action too early
- Action too late
- Action too much
- Action too little
- Action too long
- Action too short
- Action in wrong direction

- Right action on wrong object
- Wrong action on right object
- Misalignment
- Information not obtained/transmitted
- Check omitted
- Check on wrong object
- Wrong check
- Check mistimed

A comparative study of the above work indicates that there is still a need to seek a more logical taxonomy of human errors, which will be developed in the next subsection.

13.5.3.3 The Mathematical Model of Human Errors

A formal behavioral model of human errors [Wang, 2005f] is derived in this subsection according to Theorem 3.10 on human and system behaviors as developed in Section 3.4.2.

Definition 13.55 A *human behavior B* is constituted by four basic elements known as the *object* (O), *action* (A), *space* (S), and *time* (T), i.e.:

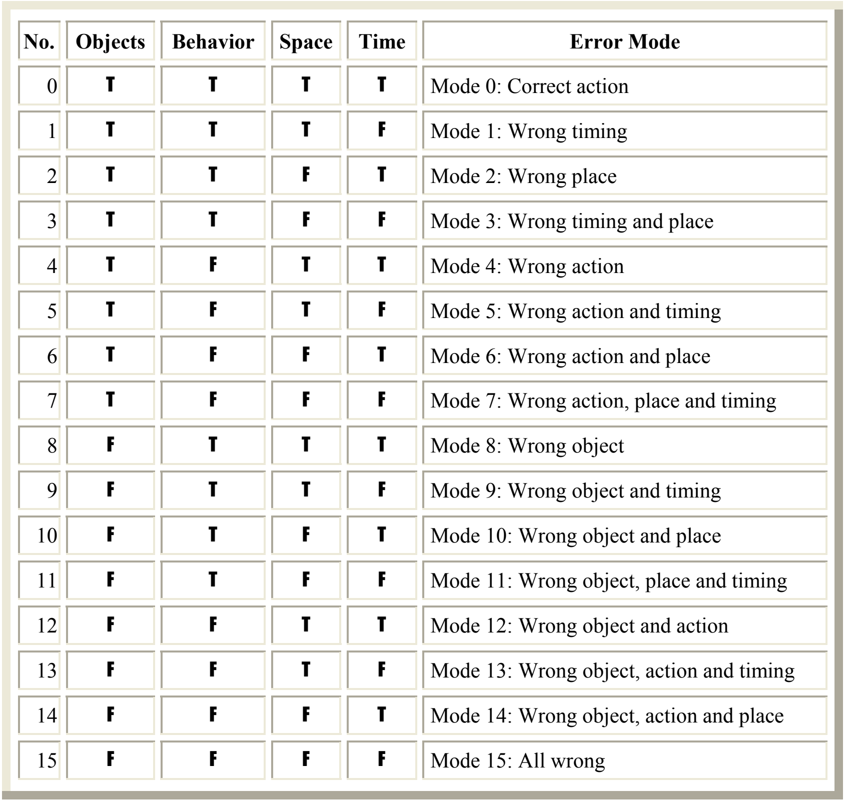
$$\begin{aligned} B &= (O, A, S, T) \\ &= O \times A \times S \times T \end{aligned} \tag{13.30}$$

Any incorrect configuration of any of these four elements results in a human error in task performance. Therefore, there are 16 modes of human errors on the basis of the combinations of these four basic elements, which form the Behavioral Model of Human Errors (BMHE) as shown in Table 13.12.

Corresponding to Table 13.12, a Human Error Tree (HET) is illustrated in Fig. 13.9. It is noteworthy that the identification of the object is the most important task in a chain of actions, because it is obvious that a correct action in a correct location at a correct time but on a wrong object is still an error action. Observing Fig. 13.9 and Table 13.12, it may be found that for a human operator, there is only 1/16 chance to get a given action or behavior to be correct, but there are 15/16 chance to get it wrong. That is, the probabilities of human success $p(+)$ and human error $p(-)$ in performing a specific task, respectively, are:

$$\begin{cases} p(+) = \dfrac{1}{16} = 6.25\% \\ p(-) = \dfrac{15}{16} = 73.75\% \end{cases} \tag{13.31}$$

Table 13.12
The Behavioral Model of Human Errors (BMHEs)



No.	Objects	Behavior	Space	Time	Error Mode
0	T	T	T	T	Mode 0: Correct action
1	T	T	T	F	Mode 1: Wrong timing
2	T	T	F	T	Mode 2: Wrong place
3	T	T	F	F	Mode 3: Wrong timing and place
4	T	F	T	T	Mode 4: Wrong action
5	T	F	T	F	Mode 5: Wrong action and timing
6	T	F	F	T	Mode 6: Wrong action and place
7	T	F	F	F	Mode 7: Wrong action, place and timing
8	F	T	T	T	Mode 8: Wrong object
9	F	T	T	F	Mode 9: Wrong object and timing
10	F	T	F	T	Mode 10: Wrong object and place
11	F	T	F	F	Mode 11: Wrong object, place and timing
12	F	F	T	T	Mode 12: Wrong object and action
13	F	F	T	F	Mode 13: Wrong object, action and timing
14	F	F	F	T	Mode 14: Wrong object, action and place
15	F	F	F	F	Mode 15: All wrong

The BMHE and HET models indicate that the natural rate of human errors in performing tasks would be very high. Fortunately, a well trained human being is fault-tolerant when performing tasks and a well established engineering process is fault-tolerant too. The major means for fault-tolerant in task performing is checking and rechecking. By adopting all checking and monitoring techniques in each step of HET, the error ratio as shown in Eq. 13.31 can be greatly decreased.

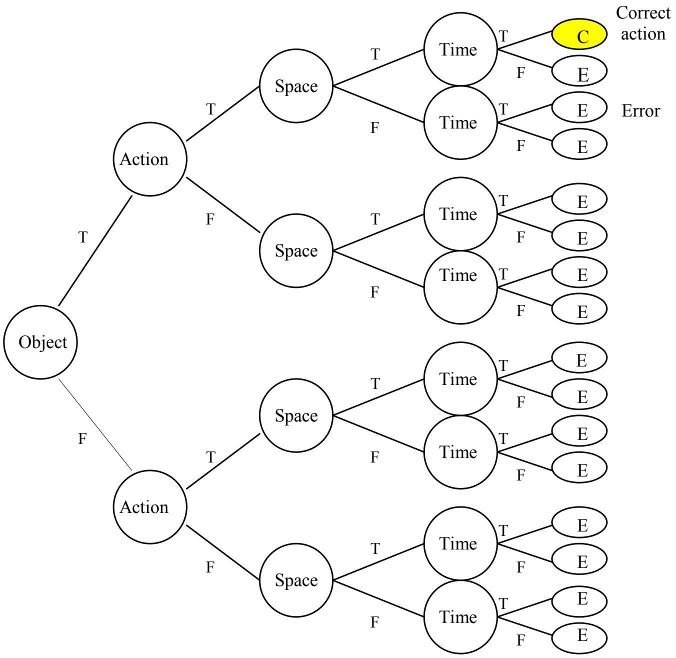


Figure 13.9 The model of Human Error Tree (HET)

13.5.3.4 The Random Properties of Human Errors

On the basis of various fault-tolerant measures and referring to Fig. 13.9, the following statistical properties of human errors may be observed.

Lemma 13.17 The *statistical properties of human errors* are as follows:

a) *Oddness*: Although individuals make different errors in performing tasks, the chance of making a single error for a given task is most of the cases than that of multiple errors.

b) *Independence*: Different individuals have different error patterns in performing the same task.

c) *Randomness*: Most of the different individuals make the same error in different times in performing tasks.

Properties (a) through (c) reveal the random nature of human errors on object, action, space, and time during performing tasks in a group.

Corollary 13.15 The *random nature of human errors* during performing tasks in a group is determined by the statistical properties that the occurrences of the same errors by different individuals are most likely at different times.

13.5.3.5 The Theoretical Foundation of Quality Assurance in Creative Work

The findings as stated in Lemma 13.17 and Corollary 13.15 form a theoretical foundation for fault-tolerance and quality assurance in software engineering. They indicates that human errors mat be prevented from happening or be corrected after their presence as soon as possible in a coordinative group context by means of peer reviews.

The 49th Law of Software Engineering

Theorem 13.5 The *n-fold error reduction structure* states that the *error rate of a work product* can be reduced up to *n* folds from the average error rate of individuals r_e in a coordinative group via *n*-nary *peer reviews* based on the random nature of error distributions and independent nature of error patterns of individuals, i.e.:

$$R_e = \prod_{k=1}^{n} r_e(k) \qquad (13.32)$$

Example 13.8 A software engineering project is under development by a group of four programmers. Given the individual error rates of the four group members as: $r_e(1) = 10\%$, $r_e(2) = 8\%$, $r_e(3) = 20\%$, and $r_e(4) = 5\%$, estimate the error rate of the final software system by adopting the following quality assurance techniques: (a) Pairwise reviews between Programmers 1 vs. 2 and Programmers 3 vs. 4; and (b) 4-nary reviews between all group members.

a) The pairwise reviews between Programmers 1–2 and Programmers 3–4 will result in the following error rates R_{e1} and R_{e2}:

$$\begin{aligned} R_{e1} &= \prod_{k=1}^{2} r_e(k) \\ &= 10\% \bullet 8\% \\ &= 0.8\% \end{aligned}$$

$$
\begin{aligned}
R_{e2} &= \prod_{k=3}^{4} r_e(k) \\
&= 20\% \bullet 5\% \\
&= 1.0\%
\end{aligned}
$$

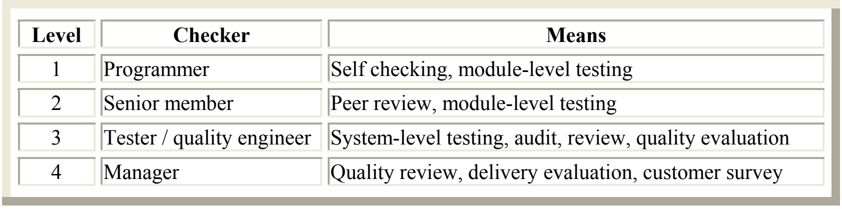
b) The 4-nary reviews between Programmers 1 through 4 will yield the following error rate R_{e3}:

$$
\begin{aligned}
R_{e3} &= \prod_{k=1}^{4} r_e(k) \\
&= 10\% \bullet 8\% \bullet 20\% \bullet 5\% \\
&= 0.008\%
\end{aligned}
$$

Theorem 13.5 and Example 13.8 explain why multiple peer reviews may greatly reduce the probability of error in program development and software engineering. Theorem 13.5 is also applicable in the academic community where quality peer-reviewed results may virtually prevent most mistakes in a final article before its publication.

In software engineering quality assurance, a four-level quality assurance system is needed for certain critical software functions and projects as shown in Table 13.13.

Table 13.13
The Four-Level Quality Assurance System of Software Engineering



Level	Checker	Means
1	Programmer	Self checking, module-level testing
2	Senior member	Peer review, module-level testing
3	Tester / quality engineer	System-level testing, audit, review, quality evaluation
4	Manager	Quality review, delivery evaluation, customer survey

Example 13.9 For a given program reviewed according to the four-level quality assurance system as shown in Table 13.13, assuming $r_e(10) = 10\%$, $r_e(2) = 5\%$, $r_e(3) = 2\%$, and $r_e(4) = 10\%$, estimate the quality of the final result of this program.

According to Eq. 13.32, the 4-nary quality assurance system can yield an expected error rate R_{e4}:

$$\begin{aligned} R_{e4} &= \prod_{k=1}^{4} r_e(k) \\ &= 10\% \bullet 5\% \bullet 2\% \bullet 10\% \\ &= 0.001\% \end{aligned}$$

The results indicate that the error rate of the above system has been significantly reduced from initial 100bugs/kLOC to 1bugs/kLOC. This demonstrates that the hierarchical organizational form for software system reviews can greatly increase the quality of software development and significantly decrease the requirement for individual capability and error rates in software engineering.

13.6 Summary

Sociology studies the structure, organization, operation, and development of human societies. The objects under study in sociology are human societies and social relations. Therefore, to some extent, it may be perceived that management science is the microsociology, while sociology is the macro management science. In both fields, the theories of system science and methodologies of system organizations play an important role in formalization of the theoretical framework of sociology.

This chapter has presented a **rigorous treatment of social organization** in the engineering context. A human society has been constructed by *individuals, groups, organizations,* and *sectors* from the bottom up. Theories and methodologies of coordinative work organization have been served as one of the main threads across chapters from engineering science, system science, management science, and economics foundations to sociology foundations. The final piece of the puzzle of the **cooperative work organization theory** has been completed in this chapter at the highest level of scopes in work organization, which provides a systematic methodology for optimal allocation of labor, resources, and schedules for a given workload in a society in general, and in a software engineering context in particular.

Social psychology such as the fundamental human traits, collective behaviors, and the perceptual influence on them have been presented, which form the underlying theory for explaining the human factors in engineering systems and societies. Theories of social organization have provided an essential understanding for coordinative work organization at various levels of societies with the new structure of the social organization trees. Then, sociology has been extended into the domain of software engineering, where social organization and ergonomics for software engineering have been analyzed that explains how human strengths, weaknesses, and uncertainty may be dealt with in the context of software engineering. As a result, the **sociology foundations of software engineering** have been established.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Sociology Foundations of Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 13. Sociology Foundations of SE

■ Principles of Sociology

- Social Structures
  - Individuals
  - Groups
  - Organizations
  - Sectors
  - Societies
- Social Behaviors
  - Social functions and relations
  - Social roles
  - Social systems
- Social Norms
  - Cultures
  - Values
  - Socialization
  - The social philosophy of Confucianism

■ Social Psychology

- The Fundamental Human Traits
  - Axiomatic human traits
  - The hierarchical model of basic human needs
- Human Perceptions and Behaviors
  - Emotions
  - Motivations

- Attitudes
- The motivation/attitude-driven behavioral model

- Collective Behaviors
  - Social conformity
  - Social synchronization
  - Coactions
  - Coordination
  - Groupthink
  - Social dilemmas
  - Social loafing

■ Theory of Social Organization

- Classic Thought of Social Organization
  - Principles of social organization
  - Classical models of social organization
- The Formal Model of Social Organization
  - The formal organization tree
  - Formal models of social organization
  - Coordinative work organization
- The Formal Model of Socialization

■ Sociology and Software Engineering

- Social Organization of Software Engineering
  - The role of the information economy in postindustrial societies
  - Maximizing strengths of individual motivations in software engineering
  - Social environments of software engineering
- Ergonomics for Software Engineering
- Human Factors in Software Engineering
  - Taxonomy of human factors
  - Types of human errors
  - The mathematical model of human errors
  - The random properties of human errors
  - The theoretical foundation of quality assurance in creative work

SIGNIFICANT FINDINGS OF THIS CHAPTER

• There are various types of societies characterized by the underpinning structures of their economies. According to the foundational economic structures, **human societies** have evolved through **five phases** known as the *hunting/gathering, horticultural/pastoral, agrarian, industrial,* and *postindustrial societies*.

- A **group** is needed because the *interdependency* among members when a given work cannot be carried out by separated individuals limited by their **resource dependency** or **functional dependency**.

- When the scale of a group increases to a certain extent, internal coordination and synchronization between members in the group will become the dominant problem. This problem forces a large group to adopt more complicated structural forms, which extend the group into an **organization**.

- As **social relations** adhere people to people in a society, **social roles** adhere people to social functions. Because both social relations and roles can be 1-to-1, 1-to-*n*, *n*-to-1, and *n*-to-*m*, the natural structures of human societies are **hierarchical trees and networks**.

- The **human emotional system** is a **binary system** that interprets or perceives an external stimulus and/or internal status as pleasant or unpleasant.

    - Although there are various emotional categories at different levels, the **binary emotional system** of the brain provides a set of pairwise universal solutions to express human feelings. For example, anger may be explained as a default solution or generic reaction for an emotional event when there is no better solution available; otherwise, delight will be the default emotional reaction.

- **Motivation** is a psychological, social modulating and coordinating influence on the direction, vigor, and composition of behaviors. This influence arises from a wide variety of internal, environmental, and social sources, and is manifested at many levels of behavioral and neural organizations.

- The **strength of a motivation** is determined by multiple factors such as: a) The *absolute motivation* $|E_m|$: it is the strength of the emotion. b) The *relative motivation* $E - S$: it is the relative difference or inequity between the expectancy of a person E for an object or an action towards a certain goal and the current status S of the person. c) The *cost* to fulfill the motivation C: A subjective assessment of the effort needed to accomplish the expected goal.

- The **theorem of strength of motivations** states that a motivation M is proportional to both the strength of emotion $|E_m|$ and the difference between the expectancy of desire E and the current status S, of a person, and is inversely proportional to the cost to accomplish the expected motivation C, i.e., $M = \frac{2.5 \bullet | E_m | \bullet (E\text{-}S)}{C}$, where $0 \leq |E_m| \leq 4$, $0 \leq (E, S) \leq 10$, and $1 \leq C \leq 10$.

- The relationship between an emotion, motivation, attitude, and behavior can be formally and quantitatively described by the **Motivation/Attitude-Driven Behavioral (MADB) model**. It states that a *behavior B* driven by a motivation M_r and an attitude is a realized action initiated by a motivation M and supported by a positive attitude A and a positive decision D_a toward the action, i.e.:

$$B = \begin{cases} \mathbf{T}, \; M_r \bullet D_a = \dfrac{2.5 \bullet | E_m | \bullet (E\text{-}S)}{C} \bullet A \bullet D_a > 1 \\ \mathbf{F}, \; otherwise \end{cases}$$

- A motivation is only a potential mental power of human beings, and a strong motivation will not necessarily result in a behavior or action. The **condition for transforming a motivation into a real behavior** or action is dependent on multiple factors, such as values, social norms, expected difficulties, availability of resources, and the existence of alternative goals.

- The **motivation of a person** is constrained by the attitude and decision making strategies of the person. The **attitude** is the **internal (subjective) feasibility** of the motivation, and the **decision making strategies** is the **external (social) feasibility** of the motivation.

- A **society** is a **dynamic human system** that is interacting not only among members of the society via social relations, but also between the society, other societies, and the natural environment.

- **Socialization** is a conforming process that a person is integrated into a society at various levels of its hierarchy by adopting certain roles, relations, cultures, customs, value systems, and norm behaviors.

- The **identification of a common set of values** is helpful to normalize individual and collective behaviors in an organization, especially a software development organization in software engineering that produces information products for a global market.

- The **basic essences for evolution** state that the *basic evolutional needs* of mankind are to preserve both the species' biological traits via *gene pools*, and the cumulated knowledge via various *information systems*.

- The lower the level of a need in the HNH hierarchy, the more *concrete* or **material-oriented** the need. In other words, the higher the level of a need, the more *virtualized* or **perception-oriented** the need.

- Understanding of the **nature of basic human needs** is not only useful in predicating motivations of human beings in a given context, but also important in identifying the driving forces for the approach of engineering organization, the types of societies, and the corresponding economic structures.

- An **autonomously synchronization tendency** between individuals exists in any permanent or temporary social context where people automatically adjust to conjunctive goals and cooperative timing.

- A weighting system that encourages and appreciates **negative or hesitant feedback** towards a current group's position is a **stable social system**.

- **Social loafing** may happen in a group where tasks are parallel allocated and the sum of all parallel capacity is greater than the workload of the group.

- Empirical and practical social organizations have been formed as results of historical, political, and economical processes. However, a few natural laws had been sought in sociology in order to understand the fundamental constraints of human societies. Toward this aim, a set of **formal sociological models** has been developed on the basis of system theory and the *System Organization Tree* (SOT). A rigorous treatment of social organization in engineering is developed. Based on the sociological models, the laws and principles behind coordinative work organization are revealed. Some of the results are particularly useful for software engineering organization.

- The **organizational coordination efficiency** states that the natural force of social organization that requires the architecture of large groups to be evolved and adapted to tree-form hierarchical structures in an organization is the need to maintain acceptable coordinating efficiency at each level of the organization tree.

- The **optimal architecture of large-scale organizations**, by which an optimal structure can be maintained at each level of its hierarchy, is an OT where the *average optimal fan-out* of a node $\overline{n}_{fo}$ or the size of the group n_G is larger than 3 and smaller than 10, i.e., $3 \leq \overline{n}_{fo} = n_G \leq 10$, where $\overline{n}_{fo}$ is the optimum labor allocation.

- The **capacity of a series work system** is determined by the least capable unit W_{min} known as the *bottleneck*. The key to *optimal serial work organization* is there is no bottleneck in the social system.

- The **capacity of parallel work system** is dominated by the most capable unit W_{max} known as the *main unit*. The key to *optimal parallel work organization* is there is no over-allocated work capacity.

- The **Formal Socialization Model (FSM)** is a relational model that describes the relationships between the *basic human needs, economic structures,* and *social types*.

  - The **type of society** $T_{S_i}, 0 \leq i \leq 4$, is determined by the dominant sector T_{E_i} of the corresponding economic structure, which is constituted by the current level of predominately unsatisfied human needs $\overline{T_{N_i}}, 0 \leq i \leq 4$, i.e., $T_{S_i} = \max(T_{E_i}) = \max(\overline{T_{N_i}}), 0 \leq i \leq 4.$.

  - The **next type of society** after post-industrialization is the information society driven by the current level of predominantly unsatisfied social and self-expressive needs and the underlying information-oriented economy.

- The **fundamental driving forces** for this trend are that the higher level human needs built upon the satisfied lower-level ones, such as cognitive (N_2), social (N_3), and self-expressive (N_4) needs, will be the new focus of post-industrialized societies. Because all N_2 through N_4 needs are based on information and intelligent services when the material level needs are satisfied, the form of economy and type of society will be evolved into the information-oriented society naturally.

- The **diversity lemma** states that the more diverse the workforce in an organization (particularly the creative software industry), the higher the opportunity to form new relations and connections that leads to the gain of the system fusion effect.

- **The random feature of human errors:** The following phenomena reveal the random nature of human errors on *object, action, space,* and *time* in performing tasks in a group:

a) Although individuals make different errors in performing tasks, the chance of making a single error for a given task is most of the cases than that of multiple errors.

b) Different individuals have different error patterns in performing tasks.

c) Most of the different individuals make the same error in different times in performing tasks.

- In software engineering quality assurance, a **hierarchical quality assurance system** is needed at four levels known as those of *programmers, senior members, testers/quality engineers*, and *managers.*

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Principles of sociology

- A **society** is a huge organized human system in which people are grouped, coordinated, interconnected, and interacted by a variety of organizations. A society as a whole is constructed by individuals, groups, organizations, and sectors from the bottom up.

- **Social structures** study the hierarchical architectures of societies at different levels and their social characteristics and interactions.

  - An **individual** is a single human being that forms the basic social unit of a society.

  - A **group** is a formal or informal social unit formed by two or more persons working towards a particular purpose.

  - An **organization** is a formal and stable social unit formed by one or more groups of people working towards a particular purpose.

  - A **sector** is a distinct branch of a society with multiple organizations that produce the same category of products or provide the same category of services.

  - A **society** is the community of people in which members of it are geographically connected and socially integrated with common customs, organizations, and values.

- **Social behaviors** can be studied hierarchically via social functions, relations, roles, and systems from the bottom up in a society.

  - A **social function** $\mathfrak{F}$ is a set of tasks and/or actions within a society that can be carried out by individuals.

  - A **social relation** R is a function between two or more persons, p, in a society, i.e., $R(p) = g : \ p \to P$, where P is all the individuals, $p \in P$, in the given society.

  - A **social role** is a set of coherent social functions that is represented by a title of a category and is expected to be conformed in the society. The *social roles RL* of a person p is a relation between the person p and a set of social functions F, $F \subseteq \mathfrak{F}$, i.e., $RL(p) = f : p \to F$, where F is a subset of all defined social functions $\mathfrak{F}$.

- **Social norms** are the '*shoulds*' of a society for regulating social behaviors that members of the society share and are expected to conform. Social norms can be considered from the aspects of cultures and values.

  - The **culture** of a society is the collected ideas, customs, behaviors, and values shared by members of the unit.

  - **Values** of a social unit are a set of ethical principles or standards shared by the unit that are used to judge and normalize social behaviors.

  - The **union** of all proven social norms from different societies, or at least their **intersection**, represents a set of univeral values of humanity.

Social psychology

- **Social psychology** is a branch of psychology that studies social interactions and their effects on human behaviors. Because the basic studying objects of sociology are individual human beings and their interactions, social psychology is the key to understand a wide range of complicated social phenomena and the driving forces underpinning them.

- The study on **human traits** forms the foundation of sociology, because every individual's social behavior is driven and constrained by those axiomatic human traits and characteristics and the derived needs based on them. The study on human traits also forms the foundation for engineering organization.

  - **Human traits** and **needs** are the fundamental force underlying almost all phenomena in human task performances, engineering organizations, and societies.

- The **basic evolutional need of humans** is the tendency to maximize the inclusive fitness of individuals and the whole mankind.

  - **Egoism** is a social behavior of human beings in which individuals put their own interests first in decision makings.

  - **Altruism** is a social behavior in which individuals sacrifice their own interests for the welfare of a group or society.

  - **Relationship:** Egoism is constrained by altruism; and the implementation of altruism is dependent on the natural egoism.

- **The Hierarchical Model of Basic Human Needs:** As an individual, the basic biological need of humans is a stable inner environment regulated by a mechanism known as homeostasis. *Homeostasis* is an adaptive biological mechanism of the human body that maintains a relatively constant state in order to live and function.

  - The **Maslow hierarchy** of human needs is identified at five levels known as the needs of *physiological, safety, social, esteem,* and *self-actualization* from the bottom up.

  - The **Human Needs Hierarchy (HNH) model** is a hierarchical model that encompasses five levels of fundamental human needs known from the bottom-up as N_0 – physiological needs, N_1 – psychological needs, N_2 – cognitive needs, N_3 – social needs, and N_4 – self-expressive needs.

- The **predominant need** of an individual is the needs at the lowest unsatisfied level of the HNH model. When multiple needs of a person are unsatisfied at a given time, satisfaction of the most *predominant need* is most pressing.

- **Perception** is a set of sensational cognitive processes at the subconscious cognitive function layers such as emotion, motivation, and attitudes. Perception may be considered as the sixth sense of human beings that almost all cognitive life functions rely on it.

  - An **emotion** is a personal feeling derived from one's current internal status, mood, circumstances, historical context, and external stimuli.

  - A **motivation** is a willingness or desire triggered by an emotion to pursue a goal or a reason for triggering an action.

- An **attitude** is a subjective tendency towards a motivation, an object, a goal, or an action based on an intuitive evaluation of its feasibility.

- **Organizational psychology** studies collective behaviors within groups and organizations, and how structures of them impact people's behaviors, productivity, and performance.

- A **collective behavior** is an integrated behavior of a group in which individuals' behaviors are influenced in different ways by the group.

  - **Individuals' behavior** in the social context is measurable and analytical in term of performance against the social norms.

  - **Social conformity** is a social phenomenon in which an individual's behavior is approached to a social norm or standard in forms of ethical values, role expectations, and laws.

  - **Social synchronization** is a social phenomenon in which an individual's behavior is timed to a social norm of a group.

  - A **coaction** is a social phenomenon in which the identical or similar actions or tasks are carried out by different individuals with little interaction.

  - A **coordination** is a social phenomenon in which the identical or similar action or task is carried out via intensive interactions between different individuals.

  - **Groupthink** is a social phenomenon in which the decision-making process within a highly cohesive group is dominated by group consensus that restrains critical thinking of members in the group.

  - **Group polarization** is a social phenomenon in which group members intend to shift toward the extreme of an already preferred position of the group.

  - **Social dilemma** is a social phenomenon in which members of a group face a conflict choice between the maximization of group's interests by cooperative actions and the maximization of own individual's interests by noncooperative actions.

  - **Social loafing** is a social phenomenon in which exists the tendency for people to work less hard on a cooperative task in a group than they do individually.

  - The **free-rider effect** is a social phenomenon in which exists the tendency for a member of a group to act noncooperatively based on the assumption that one's individual cooperative action may not be necessary because others will do for the interests of the group.

- The **sucker effect** is a social phenomenon in which exists the tendency for a member of a group to act noncooperatively based on the assumption that others may take advantage of one's individual cooperative contribution to the group.

- The **Parkinson's law** states that work intends to expand to fill the time available for its completion.

- The **Peter's law** states that bureaucrats rise to their level of incompetence in a bureaucratic system.

Theory of social organization

- From a **geographical** point of view, a society is formed by individuals, families, communities, districts, areas, provinces, and countries from the bottom-up. From a **functional** point of view, a society is formed by individuals, groups, organizations, sectors, and the whole economy.

- The **performance of an organization** is determined by both its internal model and its external environment.

- An **organizational environment** is the external constraints of a society that affect the operation of an organization.

- There are various **organizational models** and methodologies, such as *bureaucracy, division of labor,* and *system organization*. System science and system models as discussed in Chapter 10 have provided a formal approach in social studies.

- **Bureaucracy** is a classical organizational model of society in which decisions are made from the top-down.

- **Division of labor** was introduced into industry and mass manufacturing during the industrial revolution, which forms the important characteristics of industrialization

- **The Formal Organization Tree:** A **normalized organization tree (OT$_n$)** is a complete *n*-nary tree in which all leave nodes represent *employees* and the remainder represent *managers*. When the leaves (employees) are not reached in the maximum possible numbers in the OT, the right most leaves will be left open.

- An OT is fully **determinable** *iff* its number of employees (leaves) N_e and the optimal number of fan-out $\overline{n}_{fo}$ are given. The OT provides a **mathematical model** for formally analyzing the architectures of social organizations and their efficiency.

- An **organization is needed** when the size of a group is too large that it exceeds the optimal size of the group, and therefore is no longer efficient.

- The **organizational efficiency** $e_{OT}(n)$ of an n-nary organization tree OT_n is determined by the ratio between the number of employees N_e and the number of management N_m, which is approaching n when the size of the organization is large enough.

- **Cooperative Work Organization:** The structures of work organization and allocation can be organized on the basis of the theory of the maximum output of abstract system.

- A **series work organization** is a work allocation structure in which a given work is decomposed into a series of parts and each part is allocated to a person or a group.

- A **parallel work organization** is a work allocation structure in which a given work is done repetitively or jointly by multiple persons or group.

Sociology for software engineering

- **Social organization of software engineering:** Cultures and values of a software development organization help to establish a set of ethical principles or standards shared by individuals of the organization for judging and normalizing social behaviors.

- The identification of larger set of **values** and **organizational policy** towards social relations may be helpful to normalize individual and collective behaviors in the software development organization that produces information products for a global market.

- Another condition for supporting creative work of individuals in a software development organization is to **encourage diversity** in both ways of thinking and work allocation.

- **Diversity** refers to the social and technical differences of people in working organizations. Diversity includes a wide range of differences between people such as those of race, ethnicity, age, gender, disability, skills, educations, experience, values, native language, and culture.

- The **principle of system mutation** indicates that if the number of components of a system reaches a certain level – the critical mass, then the functionality of the system may be dramatically increased. That is, the increase of diversity in a system is the condition to realize the system fusion effect, which results in a totally new system.

- **Ergonomics** is a branch of engineering and behavioral science that studies human efficiency in working environment. Ergonomics is the science of work such as abilities, limitations, and characteristics of human beings and their adaptation to the working environment.

    - Ergonomics can be divided into two overlapped branches known as the **industrial ergonomics** and **human factors**. The former focuses on engineering biomechanics, or the physical aspects of human capabilities, such as force, posture, and repetition. The latter studies engineering psychology, or the mental aspects of human capability, such as the strengths and weaknesses of human brain in the working environment.

    - **Information design** is a branch of ergonomics that studies the design of signs, symbols, and instructions of information and software systems in order to enable their meaning can be quickly and safely comprehended.

- The **human factors** are the roles and effects of humans in a system that introduces additional strengths, weaknesses, and uncertainty.

    - The human factors are a **constantly important constraint** in almost all disciplines of science and engineering, even the most active and **dynamic factors** to be considered. Nevertheless, human beings themselves are directly the object of study in a number of disciplines such as psychology, cognitive science, ergonomics, sociology, cognitive informatics, medical science, neuroscience, and natural intelligence.

    - **Conservative productivity** is a basic constraint of software engineering due to cognitive complexity and due to the cognitive mechanism in which abstract artifacts need to be represented physiologically in the brain via growing synaptic neural connections.

- **The Behavioral Model of Human Errors** (BMHE): A *human error* is a human operator error caused by wrong actions and inappropriate behaviors.

- A **human behavior** *B* is constituted by four basic elements known as the *object* (*O*), *action* (*A*), *space* (*S*), and *time* (*T*), i.e., *B* = (*O, A, S, T*). Any incorrect configuration of any of these four elements results in a human error in task performance.

- Therefore, there are 16 modes of human errors on the basis of the combinations of these four basic elements.

- The **random nature of human errors** in performing tasks in a group is the statistical phenomenon that the occurrences of the same errors by different individuals are most likely at different times.

- **The theoretical foundation of quality assurance in creative work:** The *n-fold error reduction by reviewing* states that the *error rate of a work product* can be reduced up to *n* folds of the average error rate of individuals r_e in a group via *n*-nary *peer reviews* based on the random nature of error distributions and independent nature of error patterns of individuals, i.e.,

$$R_e = \prod_{k=1}^{n} r_e(k).$$

- The **hierarchical review system** in software engineering can greatly increase the quality of software and decrease the requirement for individual capability and error rates in software engineering.

Questions and Research Opportunities

13.1 Why may sociology be perceived as a special type of system science of human organizations?

13.2 Why are group studies one of the centers of sociology?

13.3 What are the natural forces and needs that expand groups into organizations?

13.4 Why can a society be modeled as a set of social functions, roles, and relations?

13.5 What are the roles of cultures and value systems in forming social norms?

13.6 Why do software development organizations need to identify a common set of values in order to normalize individual and collective behaviors in software engineering?

13.7 What are the axiomatic human traits and their relationships?

13.8 What is Maslow's hierarchy of basic human needs?

13.9 What is the Human Needs Hierarchy (HNH) model and what are its differences from Maslow's hierarchy of needs?

13.10 Why does the human emotional system tend to be a binary system?

13.11 According to Theorem 13.2, explain what determines the strength of human motivations.

13.12 What determines the mode of human attitudes?

13.13 According to the Motivation/Attitude-Driven Behavior (MADB) model, explain why an observable behavior is a result of long-chain social and psychological reasoning.

13.14 Use a table to describe the eight categories of collective behaviors such as social conformity, social synchronization, coaction, coordination, groupthink, group polarization, social dilemmas, and social loafing.

13.15 Why is a weighting system that encourages and appreciates negative or hesitant feedback towards the current position of the group a stable system?

13.16 What are the conditions of social loafing and how to avoid it?

13.17 What are the classic models of social organization?

13.18 What is Parkinson's law in social organization? How may it be formally explained by using the coordinative work organization theory developed in Chapter 8? (Hint: Consider the unlimited or unconstrained interpersonal coordination rate.)

13.19 Use an organization tree (OT) to explain when a group has to be expanded into an organization.

13.20 For an $OT(r, N_e) = OT(10\%, 100)$ as given in Example 13.1, draw a structural diagram for it in the form of a complete 5-nary tree.

13.21 According to Corollary 13.6, determine the optimal structures of a $OT(r, N_e) = OT(20\%, 30)$, and draw a structural diagram for it.

13.22 Given a software development organization with $L = N_e = 100$ employees in the first line and $r = 20\%$, analyze the optimal architecture of the normalized organization tree (OT) for the following architectural attributes:

a) The average optimal fan-out $\overline{n}_{fo}$

b) Number of optimal groups N_G

c) Depth of the organization tree d

d) The maximum number of managers required for the software company N_m.

13.23 Draw the diagram of the organization tree $OT(10, 10{,}000)$ based on the derived architectural attributes as obtained in Ex.13.22.

13.24 What are the optimal architectural attributes of an organization with $OT(\overline{n}_{fo}, N_e) = OT(10, 10{,}000)$?

13.25 What are the optimal architectural attributes of a country with 100 million people in the working force at the leave level, i.e., $OT(\overline{n}_{fo}, N_e) = OT(20, 100{,}000{,}000)$?

13.26 The complete theory of engineering work organization at the group and system levels, respectively, can be presented by Law 25 (Theorem 8.7) and Law 48 (Theorem 13.4), i.e., the *coordinative work organization theory* and *the social system organization theory* in terms of OT/SOT. Try to summarize their mathematical models and physical meanings in software engineering context.

13.27 On the bases of Theorem 13.4, Corollaries 13.13/13.14, and Example 13.7, discuss why are the key problems of software engineering not only pure technical issues rather than organizational issues.

13.28 An usual practice in the software industry is to layoff a manager when there is a financial crisis, because it's thought that such a decision is most directly and financially effective and efficient to get out of the crisis.

According to Theorem 13.3 and corollary 13.5, explain why the above decision is not a rational action rather than one that may worsen the situation.

13.29 According to the Formal Socialization Model (FSM), explain why the next form of society after the post industrial society is the information society. What is the role of software industry and software engineering in the information society?

13.30 Discuss the roles of individual motivation and cultural diversity in the social environments of software engineering.

13.31 What are the characteristics and effects of human factors in systems where humans are part of them?

13.32 Should human beings be encouraged or limited to be incorporated into system solutions? Why?

13.33 What is the Behavioral Model of Human Errors (BMHE) and how may it be used to explain the techniques in quality assurance for creative work such as software engineering review and inspection?

13.34 Read the following chapter in social psychology:

> James A. Wiggins et al. (1994), Chapter 4, Social Relationships and Groups, *Social Psychology*, 5th ed., McGraw-Hill Inc, NY.

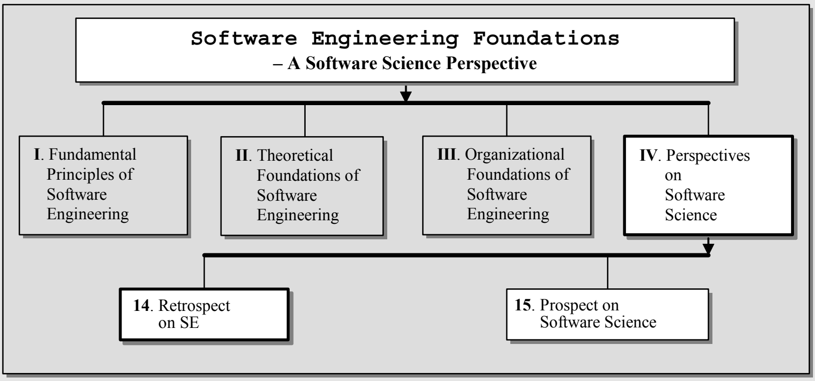
Discuss the following topics in a group or individually:

- About the author.
- Which factor plays a key role in groups: *social relationship* or *interpersonal cooperation*? Why?

- When a group is getting too large, what kind of structural changes should be made? What is the main reason that drives the changes?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

PART IV

PERSPECTIVES ON SOFTWARE SCIENCE



Software Engineering Foundations
– A Software Science Perspective
I. Fundamental Principles of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
14. Retrospect on SE
15. Prospect on Software Science

Software engineering is immature because it lacks a theoretical framework with underpinning foundations. A vast volume of empirical knowledge has been documented in software engineering without efficient and intensive theoretical processing and refinement. Therefore, the formal documentation of software engineering theories and fundamental body of knowledge is the key towards the maturity of software engineering. This book is devoted as a rational attempt to establish the formal and coherent theoretical framework of software engineering for its maturity.

The knowledge structure of Part IV on *Perspectives on Software Science* is as follows:

- Chapter 14. Retrospect on Software Engineering
- Chapter 15. Prospect on Software Science

This part addresses the theoretical and empirical framework of software science and engineering. The preceding chapters of this book have revealed that almost all the fundamental problems that could not be solved in the last four decades in software engineering stemmed from the lack of coherent theories in the form of software science. The objective of this part is to demonstrate how software science may be established on the basis of the theoretical foundations about it, the empirical observations on it, and the transdisciplinary knowledge gained from other much matured disciplines.

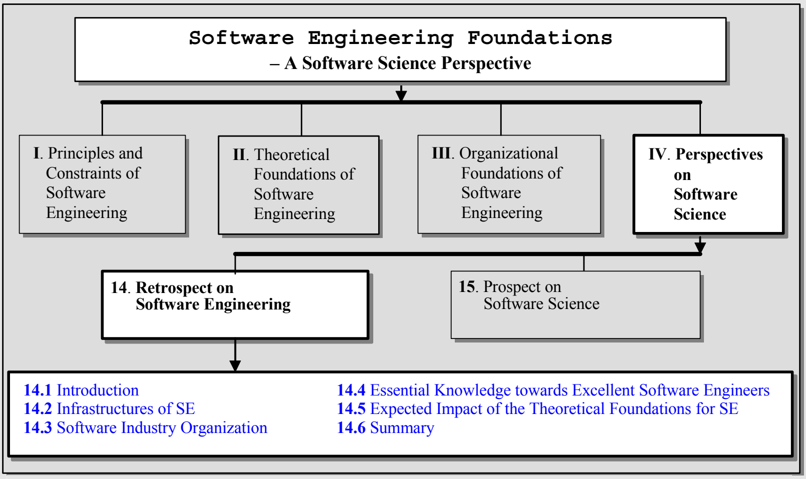
Chapter 14, *Retrospect on Software Engineering*, wraps up the entire framework of theoretical and empirical foundations of software engineering. On the basis of the first three parts of this book on principles, constraints, theoretical foundations, and organizational foundations of software engineering, this chapter moves the focus onto the entire picture. It studies the infrastructure of software engineering and discusses the organization of the software industry, particularly the organizational structure and methodologies of the software industry, and the hidden phenomenon of software maintenance crisis in software engineering. The formalized principles and laws of software engineering developed throughout this book are summarized, which form the essential body of knowledge for excellent software engineers and researchers.

Chapter 15, *Prospect on Software Science*, presents a perspective on the emergence of software science complementing to software engineering. The former is the theoretical inquiry of software and the laws constraining it; while the latter is the empirical study of engineering methodologies and techniques for software development and software industry organization. It is recognized that without theoretical physics there would be no matured applied physics; and without dynamics there would be no matured mechanical engineering. So it is with software science and software engineering. The formal structure of generic knowledge systems for all the science and engineering discipline is described first. Based on the generic knowledge system model, the theoretical framework of software engineering knowledge towards software science is modeled. Potential impacts of software science theories and methodologies on computing and conventional software engineering are discussed. New trends in software science and engineering are presented.

Part IV will wrap up this book by a retrospect on the coherent framework of software engineering theories, and a prospect on the structure of the emerging software science. This part reveals that software engineering encompasses not only a wider domain of empirical applications, but also a richer set of theoretical essences that are closer to the root of human knowledge in terms of mathematics, philosophy, cognitive informatics, computation, sociology, and system science. In software science and matured software engineering, denotational mathematics, intelligent code generation, hyper-programming, and rational work organization methodologies will play the most significant roles in this discipline.

Chapter 14

RETROSPECT ON SOFTWARE ENGINEERING


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
14. Retrospect on Software Engineering
15. Prospect on Software Science

14. Retrospect on Software Engineering

Knowledge Structure

❍ Infrastructures of software engineering

- The process infrastructure of software engineering
- Process-based software engineering (PBSE)

❍ Software industry organization

- The nature of the software industry
- Principles of software industry organization
- A perspective on the software maintenance crisis

❍ Essential knowledge towards excellent software engineers

- Basic constraints of software engineering
- Empirical principles of software engineering
- Laws of software engineering
- Formal principles of software engineering

❍ Impact of the theoretical foundations to software engineering

- The cognitive model of multidisciplinary knowledge
- Expected impacts of Wang's laws and theorems for software engineering
- Students' feedback

Learning Objectives

- To understand the process infrastructure of software engineering.
- To understand the basic methodologies for software industry organization.
- To know how to organize process-based software engineering.
- To know how to organize distributed time-shared development in software engineering.
- To be familiar with the knowledge structure for excellent software engineers, which encompasses empirical and formal principles and laws for software engineering.
- To know the cognitive model of multidisciplinary knowledge.
- To be aware of the impact of the theoretical foundations for software engineering.

"An investment in learning software engineering principles is a particularly good investment for a software professional to make because that knowledge will last a whole career – not be half obsolete within three years."

Steve McConnell (1999)

"All objects in nature and their relations are constrained by invariable laws, no matter one observed them or not at a given time."

Yingxu Wang (2004)

"We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers."

Edsger W. Dijkstra (1930 - 2002)

14.1 Introduction

Historically, software engineering has been perceived as a branch of computer science. Following the systematical study of this book, it is revealed that software engineering encompasses not only a wider domain of empirical applications, but also a richer set of theoretical essences that are closer to the root of human knowledge in terms of mathematics, philosophy, cognitive informatics, computation, sociology, and system science.

Software engineering was immature because it lacks a coherent theoretical framework and solid foundations. A vast volume of empirical knowledge has been documented without further theoretical processing and refinement. The formal documentation of software engineering theories and the fundamental body of knowledge in this book is a systematic attempt to establish the formal and coherent knowledge framework of software engineering towards a matured discipline.

On the basis of the first three parts of this book on principles, constraints, theoretical foundations, and transdisciplinary foundations of software engineering, this chapter moves the focus onto the entire picture, which explores the infrastructure of software engineering and discusses the organization of the software industry. Then, in order to wrap up this book, this chapter provides a retrospect on software engineering theories and foundations, and reviews what readers have achieved so far in acquiring essential knowledge toward excellent software engineers and researchers. The impacts of the interdisciplinary foundations for software engineering are discussed, and students' feedback on this book in the form of lecture notes is reported. The theoretical framework of software engineering presented in this book encompasses the fundamental principles and constraints of software engineering, theoretical foundations of software engineering, and transdisciplinary foundations of software engineering.

In the remainder of this chapter, the retrospect on software engineering will be presented in five sections. Section 14.2 establishes the infrastructure of software engineering, particularly the process framework of software engineering and process-based software engineering. Section 14.3 explores the organizational structure and methodologies of the software industry, where hidden phenomenon in software engineering called the software maintenance crisis is identified and analyzed. Section 14.4 reviews the essential knowledge developed in this book towards excellent software engineers. The impact of the theoretical framework of software engineering in research and practice is discussed in Section 14.5.

14.2 Infrastructures of Software Engineering

A recent trend in empirical software engineering is the shift from a focus on laboratory-oriented software engineering to a more industry-oriented view of software engineering processes. This complements preceding ideas about software engineering in terms of organization and process-orientation. From the domain coverage point of view, many of the existing software engineering approaches have mainly concentrated on the technical aspects of software development. Important areas of software engineering, such as the organizational and managerial infrastructures, have been left untouched. As software systems increase in scales, issues of complexity and professional practices become involved. Software development as an academic or laboratory activity has to engage with software development as a key industrialized process.

This expanded domain of software engineering exposes the limitations of existing methodologies that often address only individual sub-domains. There is, therefore, a demand for an overarching approach that provides a basis for theoretical and practical infrastructures capable of accommodating the whole range of modern software engineering practices and requirements. One approach is provided by process-based software engineering [Wang, 2001c; Wang and Bryant, 2002]; part of the more general trend towards a focus on the process infrastructure. Typical approaches and techniques for the establishment, assessment, and improvement of software engineering process systems are introduced in the following subsections, and further details may be referred to [Wang and King, 2000a].

14.2.1 THE PROCESS INFRASTRUCTURE OF SOFTWARE ENGINEERING

As the scale of software increases continually at an ever faster rate, greater complexity and professional practices become critical. Software development is no longer solely a black art or laboratory activity; instead, it has moved inexorably toward a key industrialized engineering process. In software engineering, the central role is no longer that of the programmers; project managers and corporate management have critical roles to play. As programmers use programming technologies, software corporation managers seek organizational and strategic management methodologies, and project managers seek professional management and software quality assurance methodologies. These developments have resulted in a modern, expanded domain of software engineering which includes three important aspects: development methodology, organization and infrastructure, and management.

Understanding the need to examine the software engineering process follows naturally from the premise that has been found to be true in other engineering disciplines, that is, that better products result from better processes. For the expanded domain of software engineering, the existing methodologies that cover individual subdomains are becoming inadequate. Therefore, an overarching approach is sought for a suitable theoretical and practical infrastructure to accommodate all the modern software engineering practices and requirements. An interesting approach, which is capable of accommodating the complete domain of software engineering, has been recognized and termed the *software engineering process*. Research into and adoption of the software engineering process paradigm will encompass all the approaches to software engineering.

Generally, a process may be described as a set of linked activities that takes an input and transform it to create an output. The software engineering process as a system is no different; it takes a software requirement as its input, while the software product is its output.

Definition 14.1 A *software engineering process* is a set of sequential practices that are functionally coherent and reusable for software engineering organization, development, and management.

The software engineering process is usually referred to as the *software process*, or simply the process. As such it is part of a more general trend that focuses on process, pushing structure and product into the background [Wang and King, 2000a].

To model the software engineering processes, a number of software process system models have been developed in the last decade. The variety and proliferation of software engineering process research and practices characterize the software engineering process as a young subdiscipline of software engineering that still needs integration and fundamental research. Studies in the software process reflect a current trend that shifts from controlling the quality of the final software product to the optimization of the processes that produce the software. It is also understood that the software engineering process, rather than the software products themselves, can be well established, stabilized, reused, and standardized.

The technical and organizational infrastructures of software engineering rely on the software engineering processes. The processes of software engineering are complex systems as described by various process models and standards such as CMM [Humphrey, 1988/89/95; Paulk et al., 1991/93/95], ISO 9001 [ISO 9001, 1989/94; ISO 9000-3, 1991], BOOTSTRAP [Koch 1993; Haase et al. 1994; Kuvaja et al. 1994], ISO/IEC 15504 [ISO/ICT, 2000; Dorling, Wang et al., 1999], and SEPRM [Wang et al., 1998b/99a; Wang and King, 2000a]. These models collected a set of processes ranging from 18 to 51. This section comparatively explores current process models and the relationships among them.

CMM was initially developed as an assessment model for software engineering management capabilities. As such it was expected that it would provide useful measures of organizations bidding or tendering for software contracts. However, it was soon realized that the concept of *process* for software engineering had more utility than that of capability assessment. Software development organizations may use the process model as an infrastructure for internal process organization and improvement. As a result of this deeper understanding, new practices in process-based software engineering have been emerging in the last decade. This may be considered as one of the important inspirations arising from CMM and related research.

In the software industry, software development is commonly perceived as a one-off activity. On the other hand, one of the most interesting findings in software engineering practices is that the processes of software development are relatively stable, repeatable, and reusable. Therefore, software engineering processes can be adopted as the *infrastructure* for software engineering. This leads to the development of the concept and technology known as process-based software engineering.

The Software Engineering Process Reference Model (SEPRM) was developed in 1998 [Wang et al., 1998b/99a; Wang and King, 2000a], which provides a comprehensive process framework of 51 processes and 444 base practice activities for software engineering. SEPRM is an integration and extension of the major process models of CMM, ISO 9001, BOOTSTRAP, and ISO/IEC 15504 as shown in Fig. 14.1. The process framework and capability model of SEPRM has been presented in Section 11.5.2, which encompasses three process subsystems known as the *organization, engineering, and management* processes.

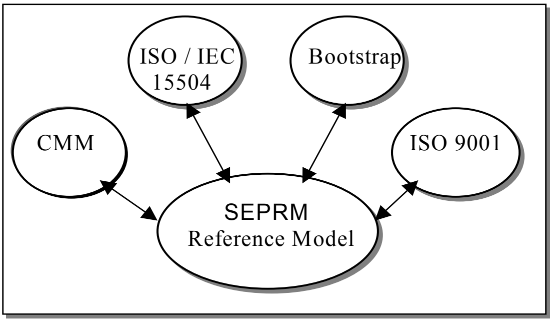


Figure 14.1 The role of the SEPRM reference model for software engineering

14.2.2 PROCESS-BASED SOFTWARE ENGINEERING (PBSE)

Software engineering is a discipline that has emerged from computer science and is based on interdisciplinary theoretical and empirical methodologies. Initial approaches developed thus far have concentrated on technical aspects of software engineering, such as programming methodologies, software development models, and formal methods, while a cutting-edge approach, process-based software engineering, has been formed [Wang and Bryant, 2002] in the last decades for integrating the modern domain of software engineering.

Definition 14.2 *Process-Based Software Engineering* (PBSE) is an organizational methodology for software engineering, by which the infrastructure of software engineering, encompassing the three process subsystems of organization, development, and management, is integrated by a well-defined process reference model.

This expanded view of the domain exposes the limitations of conventional approaches, methodologies, and tools; but this is not to imply that the wealth of experience that they embody should be jettisoned. On the contrary, we would advocate the development of an inclusive and integrative approach that offers a suitable theoretical and practical infrastructure capable of accommodating both new demands and existing expertise: Hence the process-oriented view.

The software process approach towards software engineering encompasses systematic, organizational, and managerial infrastructures for software engineering. It is necessary to expand the horizons of software engineering in this way because of the rapidly increasing complexity and scale demanded by software products. The need to ensure software quality and to increase productivity also provides impetus for PBSE.

14.2.2.1 The Organization Model of PBSE

In software engineering process research, it has been assumed that a process system should have already existed in a software development organization so that a process assessment and improvement project could be carried out directly. However convenient this assumption is, it is not true that the majority of software organizations have formal and definable processes.

In reality, a process assessment project starts by the mapping of a software organization's existing processes to a process model that has been chosen for the assessment. The usual cases are that a software development organization has only some loose and informal practices, rather than a defined and coherent process system. This scenario leads to the observation that rigorous PBSE has to start from process establishment rather than process assessment in a software development organization. Therefore, the right order of events in achieving software engineering process excellence in an organization is first, process establishment; second, process assessment; and then process improvement as shown in Fig. 14.2.

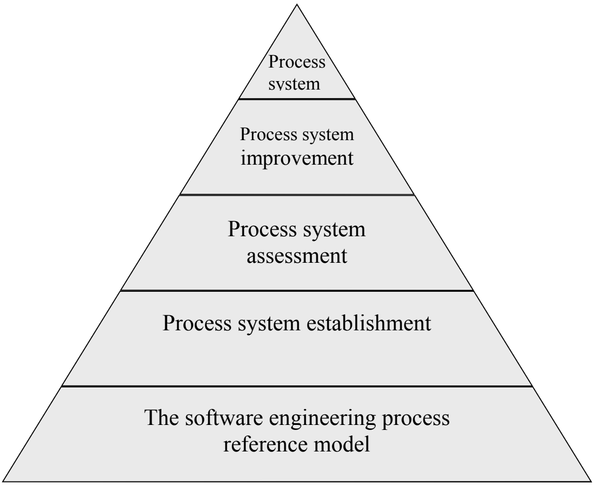

Process system
Process system improvement
Process system assessment
Process system establishment
The software engineering process reference model

Figure 14.2 Software engineering process system establishment

For *modeling* a process system, processes are elicited and integrated from the bottom up. Processes in the development subsystem are first analyzed and modeled. Corresponding to the development processes, the management processes are then deployed as measures to support and control the development processes. The third step is to design the organization processes, which are the top-level management processes oriented to the whole software development organization, and which are applicable to all software engineering projects within the organization.

It is generally considered that there would be a number of parallel development and management processes for individual projects within a software development organization. For the purpose of *controlling* a process system, software engineering processes are implemented and activated top down, from the organization level to the project level. Therefore, the relationship between the *organization*, *management*, and *development processes* can be further refined [Wang and King, 2000a; Wang and Bryant, 2002] as shown in Fig. 14.3.

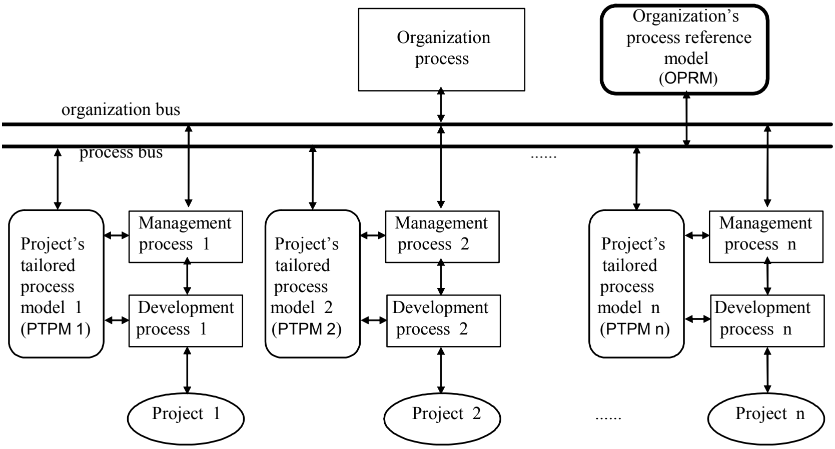


Figure 14.3 Practices in process-based software engineering

Fig. 14.3 shows the common practices in organizing a software engineering process system. It is noteworthy that there is only one organization process subsystem in a software development organization, which will be based on the Organization's Process Reference Model (OPRM).

Definition 14.3 A *process reference model* is an established, validated, and proven software engineering process model that consists of a comprehensive set of software processes and reflects the benchmarked best practices in the software industry.

At the top level, a software development organization may adopt an existing international standard or an established process model as its OPRM; or, it can develop a specific organization-oriented OPRM based on the existing models and the organization's own practices and experiences in software engineering. The OPRM plays a crucial rule in the regulation, coordination, and standardization of an organization's software engineering practices.

At project level, a number of parallel development and management processes may exist based on the individual Project's Tailored Process Model (PTPM), which are derived models of the OPRM reference model. In Fig. 14.3, the process reference model OPRM is the key for empirical PBSE. If an OPRM is well established in an organization, the PTPMs at project level can easily be derived.

In PBSE, the OPRM reference model could, and usually should, be tailored or adapted to a specific project according to the nature of the project, taking into account application domain, scope, complexity, schedule, experience of project team, reuse opportunities identified and/or resources availability, and so on. For a PTPM of an individual project, the management and development processes should be one-to-one designed and synchronized. Tailoring of a PTPM from a comprehensive OPRM makes the software project leaders' tasks dramatically easier. Using this approach, project organization and conduct can be effectively performed within an organization's unified software engineering process infrastructure.

14.2.2.2 Software Engineering Process System Establishment

An initial and fundamental step in PBSE is process system establishment. The major aim of process establishment is to build up a software engineering process reference model for a software development organization. When a process system is established and experienced, improvement can be initiated effectively via process assessment and benchmarking.

14.2.2.2.1 Procedure to Derive a Software Project Process Model

There are three basic steps for deriving a software project process model. Referring to the illustration of PBSE methodologies in Fig. 14.3, the procedure to derive a process model at the project level is explored in the following subsections.

(a) Select and Reuse a Process System Reference Model at Organization Level

The most efficient way to establish a process system is to reuse a standard or well-accepted process model. In selecting an existing process model as an organization's reference model, one of the key issues is that the reference model should be reasonably comprehensive in order to enable an easy derivation of working process models at project level. Another issue is that the reference model should be able to serve many purposes in software engineering such as multi-type process assessment, improvement, training, and internal standardization. The third issue is the flexibility of the reference model, i.e., the selected reference model should allow incorporation of the host organization's experience and special needs into the reference model and derived models.

When an organization's process system is determined, the next step is to uphold it as the organization's official and unified software engineering platform. Based on this, various process models should be derived for different projects.

(b) Derive a Process Model at Project Level

Before commencing a new project, the first thing that a project manager needs to do is to derive the project's process model as the infrastructure for the project. The project process model will serve as a blueprint for organizing all activities that are going to be enacted within the scope of the project, including technical, managerial, organizational, customer, and supporting activities.

A checklist of factors for consideration in deriving a project process model from the chosen reference model is shown in Table 14.1. When all factors are weighted by high (H), medium (M), or low (L), a rating for what kind of project process model is needed can be determined according to Eqs. 14.1 and 14.2.

Assuming that S_i is the ith weight for factor i and n is the number of total factors, the average score, S, or the level of requirement for a derived model is defined as:

$$S = \frac{1}{n}\sum_{i=1}^{n} S_i \tag{14.1}$$

According to the average score S, the type of derived model determined by the weighted factors can be estimated as follows:

$$S \begin{cases} > 3, \text{ the need is for a } \textit{complete} \text{ project process model} \\ = 3, \text{ the need is for a } \textit{medium} \text{ project process model} \\ < 3, \text{ the need is for a } \textit{light} \text{ project process model} \end{cases} \tag{14.2}$$

Table 14.1
Determining Type of Derived Process Models for a Project

No	**Project Factor**	**Weight**			**Score**
		H	**M**	**L**	
1	Importance	✓			S1 = 5
2	Difficulty			✓	S2 = 1
3	Complexity	✓			S3 = 5
4	Size		✓		S4 = 3
5	Domain knowledge requirement		✓		S5 = 3
6	Experience requirement	✓			S6 = 5
7	Special process needed			✓	S7 = 1
8	Schedule constraints		✓		S8 = 3
9	Budget constraints	✓			S9 = 5
10	Other process constraints	✓			S10 = 5
Total		25	9	2	S = 3.6

Note: H = High (5), M = Medium (3), L = Low (1).

For instance, applying Eq. 14.1 to the weights of the ten factors as shown in Table 14.1 results in an estimated average score *S* = 3.6. According to Eq. 14.2, the project process model has to be a relatively complete model that covers almost related process areas modeled in the reference model.

Note the factors shown in Table 14.1 are examples for demonstrating how the type of project process model may be determined in a formal way. It is by no means exhaustive. Therefore, readers may add, delete, and/or modify the factors in order to make them suitable for their specific projects.

(c) Apply the Derived Project Process Model

When a project process system model is derived, the next step is to accept, as a common platform, the process model at both project and individual levels, and to apply the project process model to all activities within the project scope.

It can be seen that the reference model approach to implement software engineering provides project managers with a means to derive and organize a project process model in a consistent and transparent manner. It also provides software engineers and others in a software project with a clear picture of their roles, interactions, and relationships to each other.

14.2.2.2.2 Methods for Deriving a Software Project Process Model

To establish a process model for a software project, three types of methods may be introduced. They are process model tailoring, extension, and adaptation, ordered increasingly according to their technical difficulty in applications.

(a) Process Model Tailoring

Definition 14.4 *Process model tailoring* is a model customization method for making a process model suitable for a specific software project by eliminating unnecessary processes.

Model tailoring is the simplest method to derive a project process model from a comprehensive organizational process reference model. The only technique is to delete what is not needed in order to establish a specific software project based on an understanding of both the reference model and the nature of the project.

(b) Process Model Extension

Definition 14.5 *Process model extension* is a model customization method for making a process model suitable for a specific software project by adding additional processes.

Model extension requires a project manager capable of integrating new processes, adopted from either process models or best practices repositories, into the current project process model or organizational process reference model. When new processes are introduced, a validation phase is needed for monitoring their fitness and performance in the whole process system.

(c) Process Model Adaptation

Definition 14.6 *Process model adaptation* is a model customization method for making a process model suitable for a specific software project by modifying, updating, and fine-tuning related processes.

Model adaptation is useful when a project manager is experienced with respect to one process reference model and prepared to monitor the performance of adapted processes during a project life span. All of the above three approaches for process model derivation and establishment can be used individually or together to derive an effective project process model for software engineering.

14.2.2.3 Software Engineering Process System Assessment

It is believed that if one cannot measure a process system, one cannot improve it. Therefore, *Software Process Assessment* (SPA) is critical for process improvement. Various methodologies for SPA have been developed in the last decade. This section describes the integrated SPA framework of SEPRM, and demonstrates that current process models, such as CMM, ISO 9001, BOOTSTRAP, and ISO/IEC 15504, can be perfectly fitted into this framework.

14.2.2.3.1 Process Assessment Methods against Different Reference Systems

From the viewpoint of reference systems there are four types of assessment methods: model-based, standard-based, benchmark-based, and integrated (model-and-benchmark-based) assessment.

(a) Model-Based Assessment

Definition 14.7 *Model-based assessment* is an SPA method by which a software development organization is evaluated against a specific process and capability model, and according to a specific capability determination method provided by the model.

Model-based assessment is a kind of *absolute* assessment approach. Using this approach, a software development organization is evaluated against a fixed process framework and a defined capability scale. The assessment result reports a capability level of a software development organization against the capability scale of the model. CMM and BOOTSTRAP are examples of model-based assessment methodologies.

(b) Standard-Based Assessment

Definition 14.8 *Standard-based assessment* is an SPA method by which a software development organization is evaluated against a specific process and capability model defined by a standard, and according to a specific capability determination method provided in the standard.

Standard-based assessment is a special type of model-based assessment method. It also provides an absolute assessment approach by which a software development organization's process capability is rated against a defined capability scale. ISO/IEC 15504 and partially ISO 9001 are examples of standard-based assessment methodologies.

(c) Benchmark-Based Assessment

Definition 14.9 *Benchmark-based assessment* is an SPA method by which a software development organization is evaluated against a set of benchmarks of software processes, and according to a specific capability determination method.

Benchmark-based assessment is a kind of *relative* assessment approach. By this approach a software development organization is evaluated against a set of benchmarks. Thus, the assessment result associated with a software development organization's capability level may be presented in three relative levels: below, equal, or above the benchmark of each process.

(d) Integrated Assessment

Definition 14.10 *Integrated assessment* is a kind of composite model-based and benchmark-based SPA method in which a software development organization is evaluated against both a benchmarked process model and a capability model, and according to a specific capability determination method provided in the model.

The integrated assessment method inherits the advantages of both absolute and relative SPA methods as described in this section. Using the integrated assessment method, a software development organization can be evaluated against both a benchmark and an absolute capability scale at the same time. The SEPRM model is such an integrated SPA model. Another advantage of the integrated assessment method is its ability to provide a quantitative guide for software process improvement.

14.2.2.3.2 Process Assessment Methods Based on Different Model Structures

From the viewpoint of model framework structures, there are three types of assessment methods. They are: checklist-based assessment, 1-D process-based assessment, and 2-D process-based assessment, as illustrated in Fig. 14.4.

Fig. 14.4 shows that a 2-D process model allows all processes to be performed and rated at any process capability level. A 1-D process model is a special case of 2-D models, where a group of processes is defined and rated at a certain capability level. For example, according to the 1-D process model, processes 7 – 13 in Fig. 14.4 can only be performed, and therefore rated at level 3 or below. Similarly, the checklist-based process model is a simpler 1-D process model, where all processes are defined and rated at a single level with equal importance.

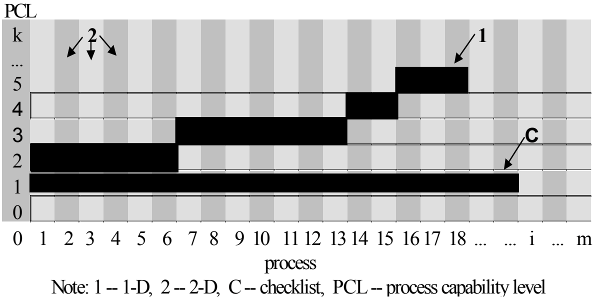


Figure 14.4 Structures of process assessment models

(a) Checklist-Based Assessment

Definition 14.11 *Checklist-based assessment* is an SPA method that is based on a pass/fail checklist for each practice and process specified in a process model.

A checklist-based assessment model is the simplest assessment methodology. This kind of method is only suitable for SPA. It is not much help in step-by-step process improvement. The ISO 9001 model provides a checklist-based assessment method.

(b) 1-D Process-Based Assessment

Definition 14.12 *1-D-based assessment* is an SPA method that determines a software development organization's capability from a set of processes in a single process dimension.

The 1-D assessment is an extension of the checklist-based assessment. This type of model is suitable for process improvement in project or organization scopes while, at the same time, being relatively weak in detailed process scope simply because processes have been grouped and pre-allocated at specific capability levels as shown in Fig. 14.4. CMM and BOOTSTRAP are examples of 1-D assessment models.

An issue presenting in such methods is that there are no widely accepted criteria prescribing how a set of software processes are grouped and mapped onto different capability levels. In principle, the processes defined in a model would be practiced at any capability level. That is, software processes in practice have no inherited capability levels; only the software development organizations and the people who are performing the processes can be measured by capability levels.

(c) 2-D Process-Based Assessment

Definition 14.13 *2-D process-based assessment* is an SPA method that employs both process and capability dimensions in a process model, and derives process capability by evaluating the process model against the capability model.

The 2-D assessment method enables every process in the process dimension to be performed and evaluated against the capability dimension at all levels. This is a flexible approach to software process assessment, although effort spent in a 2-D process assessment would be much higher than that of a 1-D or checklist assessment. This type of model is suitable for process improvement from process scope to project and organization scopes because it provides precise measurement for every process at all the capability levels. ISO/IEC 15504 and SEPRM are examples of 2-D assessment models.

Conventionally, 1-D methods were considered to have provided a process dimension in process assessment. By comparing this with the 2-D assessment methods described above and in Fig. 14.4, it may be predicted that there is another kind of 1-D process assessment model which implements only the capability dimension, while leaving the process dimension open for a software development organization or the process model providers to design and implement. This would provide a level of flexibility in software process assessment and standardization.

14.2.2.4 Software Engineering Process System Improvement

Software engineering process system improvement is the goal of process assessment, acting on issues found in an assessment and enhancing the performances of processes in the process system. This section attempts to describe major philosophies in *Software Process Improvement* (SPI) and alternative SPI methodologies.

14.2.2.4.1 Software Process Improvement Philosophies

There are various philosophies underpinning SPI. Key categories of SPI philosophy are goal-oriented process improvement, benchmark-based process improvement, and continuous process improvement. This subsection discusses philosophies behind the process improvement methodologies. The usability of various SPI approaches and their relationships are also commented upon.

(a) Goal-Oriented Process Improvement

Definition 14.14 *Goal-oriented process improvement* is an SPI approach by which process system capability is improved by moving towards a predefined goal, usually a specific process capability level.

This approach is simple, and is the most widely adopted philosophy in software engineering. For example, ISO 9001 provides a pass/fail goal with a basic set of requirements for a software process system. CMM, ISO/IEC 15504, and SEPRM provide a 5/6-level capability scale that enables software development organizations to set more precise and quantitative improvement goals.

(b) Benchmark-Based Process Improvement

Definition 14.15 *Benchmark-based process improvement* is an SPI approach by which process system capability is improved by moving towards an optimum combined profile according to software engineering process benchmarks, rather than a maximum capability level.

This is a realistic and pragmatic philosophy for process improvement. It is argued that in order to maintain sufficient competence, a software organization does not need to push all its software engineering processes to the highest level because it is neither necessary nor economic. This philosophy provides alternative thinking to the idea "the higher the better for process capability" as is presented in the goal-oriented process improvement approach.

Using the benchmark-based improvement approach, an optimized process improvement strategy identifies a sufficient (the minimum required) and economic target process profile, which provides an organization with sufficient margins of competence in every process. It does not necessarily set them all at the highest level of a capability scale.

(c) Continuous Process Improvement

Definition 14.16 *Continuous Process Improvement* is an SPI approach by which a process system's capability is required to be improved all the time, and toward ever higher capability levels.

This is considered an oriental philosophy that accepts no top limits or discrete goals because "ideal" standards are continuously changing. It is this assumption that change is normal that is in tune with modern management theory. Continuous process improvement has been proven effective in engineering process optimization and quality assurance. Using this approach, SPI is a continuous, spiral-like procedure. The Deming Circle, plan-do-check-act, is a typical component of this philosophy.

In continuous process improvement there is no end to process optimization, and all processes are supposed to be improved all the time. There is a criticism that the goals for improvement are not explicitly stated in this philosophy. Therefore, when adopting continuous process improvement, top management should make clear the current goals, as well as the short, middle, and long-term ones.

Generally, goal-oriented methodologies will still constitute the mainstream in SPI. However, 2-D process models provide more precise process assessment results, and the benchmark-based process models provide empirical indications of process attributes, benchmark-based improvement will gain wider application. Also, the continuous process improvement approach will provide a basis for sustainable long-term strategic planning.

14.2.2.4.2 Software Process Improvement Methodologies

The above discussion on the philosophies for process improvement yields the basis for an investigation of possible software process improvement methodologies. There are two basic SPI methods – assessment-based and benchmark-based process improvement. The former improves a process system from a given level in a defined scale to a next higher level; the latter provides improvement strategies by identifying gaps between a software development organization's process system and a set of established benchmarks. In addition, a combined approach may be adopted.

(a) Model-Based Improvement

Definition 14.17 *Model-based improvement* is an SPI method by which a process system can be improved by basing its performance and capability profile on a model-based assessment.

Using this idea, the processes inherent in a software development organization are improved according to a process system model with step-by-step suggestions. CMM and BOOTSTRAP are examples of such a model-based process improvement methodology.

(b) Standard-Based Improvement

Definition 14.18 *Standard-based improvement* is an SPI method in which a process system can be improved by basing its performance and capability profile on a standard-based assessment.

Using this approach, the processes inherent in a software development organization are improved according to a standardized process system model. ISO/IEC 15504 provides a standard-based improvement method. However, it is noteworthy that ISO 9001 is probably not suitable because it lacks a process improvement model and a step-by-step improvement mechanism [Wang and King, 2000a].

(c) Benchmark-Based Improvement

Definition 14.19 *Benchmark-based improvement* is an SPI method in which a process system can be improved by basing its performance and capability profile on a benchmark-based assessment.

Benchmark-based improvement is a kind of relative improvement approach. Using this approach, the processes inherent in a software development organization are improved according to a set of process benchmarks. It provides an optimized and economical process improvement solution. SEPRM is the first benchmarked model for enabling benchmark-based process improvements [Wang et al., 1998b/99a; Wang and King, 2000a].

(d) Integrated Improvement

Definition 14.20 *Integrated improvement* is a combined model-based and benchmark-based SPI method in which the process system can be improved by basing its performance and capability profile on an integrated model-based and benchmark-based assessment.

The integrated process improvement method inherits the advantages of both absolute and relative SPI methods. Using the integrated improvement method, the processes of a software development organization are improved according to a benchmarked process system model. SEPRM is designed to support integrated model- and benchmark-based process improvement.

14.3 Software Industry Organization

Although Brooks perceived that there is "no silver bullet in software engineering [Brooks, 1987]," software engineering itself has already been a silver bullet for other engineering disciplines. This is because, as discussed in

Section 10.5.2, the most powerful means for describing complex system behaviors and relations is software and supporting denotational mathematics.

Therefore, the IT industry in general, and the software industry in particular, is gaining its profound importance in the information society. However, the organizational theories and methodologies for the software industry, as an important part of software engineering in the large, have been almost overlooked in this discipline.

This subsection explores the nature of the software industry, and describes fundamental principles of software industrial organization. Important methodologies of software industrial organization are proposed on the basis of the engineering, system, management, and economics foundations of software engineering developed throughout this book.

14.3.1 THE NATURE OF THE SOFTWARE INDUSTRY

Software engineering in the large has been organized with the mass production metaphors in the industry, which analogizes the machine-building origins of manufacture engineering and the system of mass production by interchangeable parts that grew out of them.

In the Dagstuhl Seminar Series #9635 on *History of Software Engineering* held in Germany in August 1996 [Aspray et al., 1996], the organizers, W. Aspray, R. Keil-Slawik, and D.L. Parnas, identified the characteristics and idiosyncracies of computer science in general and software engineering in particular as follows:

- Highly innovative and rapidly changing field with no broadly recognized core of material that every practitioner must know.
- Few results are supported by empirical or comparative studies.
- Work within the field older than 3–4 years is rarely acknowledged or referenced.
- Old problems are given new names and old solutions overlooked.
- Evolution of the discipline is tightly coupled to economic and societal demands.
- There is a need for interdisciplinary work comprising, e.g., mathematics, psychology, business, or management science,
- Continuing debate about whether there should be a discipline called software engineering, and if so, whether this should be treated as another discipline among the set of traditional engineering disciplines.

Barry Boehm classified software engineering problems into two areas [Boehm, 1976/83]: a) Detailed design and coding of systems software by experts in a relatively economics-independent context; and b) Requirements analysis, design, text, and maintenance of application software by technicians in an economics-driven context. The former is the domain for software scientists whilst the latter is for software engineers. Boehm thought that, although those scientific principles available to support software engineering address problems in Area (a), the most pressing software engineering problems are in Area (b).

Michael Mahoney wrote: "That the search continues after twenty five years suggests that software may be fundamentally different from any of the artifacts or processes that have been the object of traditional branches of engineering: it is not like machines, it is not like masonry structures, it is not like chemical processes, it is not like electric circuits or semiconductors [Aspray, et al., 1996]."

Further, Stuart Shapiro identified the uniqueness of software engineering [Aspray et al., 1996]:

> "Misconceptions of the nature of engineering aside, though, computing and software appear fundamentally different from other areas of technological practice owing to their wide ranging applicability. Computers are general-purpose problem-solving devices and their wide utility is a function of this. However, their utility in a specific context is due to the software which turns them into special-purpose problem-solving devices. Software can play this role because it is abstract and thus unusually malleable. With this abstractness, however, comes a complexity which challenges both the cognitive processes of the individual and the degree to which the software development process can be automated.
>
> "Because computer systems span a virtually limitless number of problem domains but must function within specific ones, fundamental problem-solving processes are of exceptional concern in computing and this is one reason for the seeming inadequacy of any one model of professional activity. Moreover, this irreducible tension between specificity and generality marks both software development techniques as well as software applications. Software technologists must find a balance between sophisticated and powerful context-dependent features usable in a narrow domain and less sophisticated and powerful features amenable to more general usage. This is one reason why a software 'industrial revolution' seems quite unlikely, as it suggests the difficulty of producing high-level yet widely usable standard software components."

According to economics, in a normal market where equivalent alternatives exist, either increasing price or lowering quality will result in a loss of market share of the producer. Whilst if monopoly exists in a market, a producer may behave so without affecting its market share. The software market, particularly the system software one, is basically a monopolistic market. Therefore, the minimization of price and assurance of quality are difficult to be guaranteed, at least at the same time.

The software market is a sector of the information processing market, where *standardization* and *human cognitive familiarity* play an important role in market share. Therefore, international or industrial standards, as well as intellectual properties, are important virtual capitals in the software industry.

14.3.2 PRINCIPLES OF SOFTWARE INDUSTRY ORGANIZATION

With the understanding of the uniqueness of the software industry as discussed in Section 14.3.1, this subsection attempts to explore the basic principles of software industry organization and useful organizational forms for the software industry, such as separation of software designers, builders, and quality assurors in software engineering, as well as the new trend of software engineering known as Distributed Time-Shared Development (DTSD).

14.3.2.1 Basic Principles of Software Industrial Organization

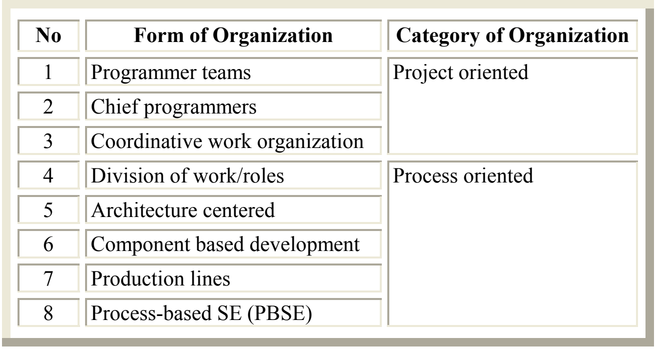
The key organization principles for the software industry are as follows:

- To improve productivity
- To practice specialization or division of labor
- To deal with the labor-time interlock constraint

The basic forms of software industrial organizations [Baker, 1972; Aron, 1983; Perry et al., 1994; ISO 9001, 1989/94; Schael, 1998; Wang and Bryant, 2002; Wang, 2007d] can be summarized in Table 14.2.

The major organizational methodology for the software industry is PBSE as presented in Section 14.2. Based on a generic software engineering process model such as SEPRM [Wang et al., 1998b/99a; Wang and King, 2000a], software engineering activities and processes at personal, project (team), and enterprise levels can be well organized in the three essential aspects of software technology, organization, and management.

Table 14.2
Forms of Software Industrial Organization



No	Form of Organization	Category of Organization
1	Programmer teams	Project oriented
2	Chief programmers	
3	Coordinative work organization	
4	Division of work/roles	Process oriented
5	Architecture centered	
6	Component based development	
7	Production lines	
8	Process-based SE (PBSE)	

14.3.2.2 Separation of Software Designers, Builders, Quality Assurors, and Maintainers in Software Engineering

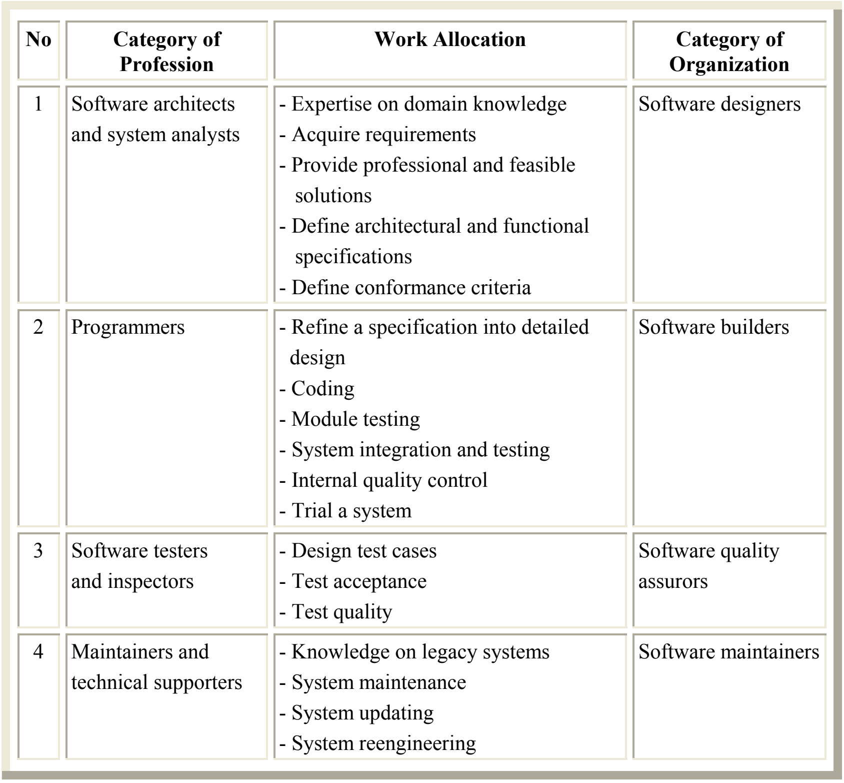
Major current strategic problems in the software industry may be identified as follows:

- *Referees are also players:* All the responsibilities in software design, implementation, and quality assurance are carried out by the same organization, even the same engineer or group. As a consequence of this confused and overlapped allocation of responsibilities, whenever time, budget, or skills are limited, quality tends to be the first victim in a software engineering project under this form of organization.

- *Too high requirements and responsibility are put onto the shoulders of customers*: The fact that is often overlooked in software engineering is that customers may not be able to understand and evaluate the requirements, functionality, quality, reliability, and complete correctness of complex software systems. Therefore in software engineering it is unwise to rely on customers for complete or thoughtful system requirements. It is also unwise to let or to agree by any party that customers should ensure the sole responsibility for testing and evaluating a new software system.

In order to solve the above inherited problems, a separation of roles in the software industry is necessary. As shown in Table 14.3, the software industry is ideally split into four sectors known as the organizations of software designers, software builders, software quality assurors, and software maintainers with totally separated and explicitly designated roles and responsibilities.

Table 14.3
Specialization of Roles and Responsibilities in Software Engineering



No	Category of Profession	Work Allocation	Category of Organization
1	Software architects and system analysts	- Expertise on domain knowledge - Acquire requirements - Provide professional and feasible solutions - Define architectural and functional specifications - Define conformance criteria	Software designers
2	Programmers	- Refine a specification into detailed design - Coding - Module testing - System integration and testing - Internal quality control - Trial a system	Software builders
3	Software testers and inspectors	- Design test cases - Test acceptance - Test quality	Software quality assurors
4	Maintainers and technical supporters	- Knowledge on legacy systems - System maintenance - System updating - System reengineering	Software maintainers

14.3.2.3 Distributed Time-Shared Development in Software Engineering

Distributed time-shared development is a new approach of division of labor in the time-dimension in contrary to division of labor in the functional or specialization dimension.

Definition 14.21 *Distributed Time-Shared Development* (DTSD) is a software engineering methodology that geographically allocates software development work broadly distributed in time zones with a wide-area Intranet.

This methodology takes advantages of geographically allocated project teams distributed in different time zones, but interconnected through a wide-area Intranet and supported by remote execution capabilities. Well organized and synchronized DTSD projects may gain time greatly in development, because DTSD provides a virtual 24-hour software development organization with the teams deployed in two or three countries globally.

14.3.3 A PERSPECTIVE ON THE SOFTWARE MAINTENANCE CRISIS

Although the software crisis has been predicted since the 1950s before the establishment of software engineering as a discipline, experts perceive it differently. Some of them may still doubt if it has ever existed; the rest may claim that we were able to cope with the crisis via software engineering techniques in the last decades. Despite the continuous argument on the generic software crisis, the author perceives there is a real and hidden crisis in software engineering and the software industry known as the software maintenance crisis.

One of the important findings according to the economic models of software engineering as developed in Chapter 12 is the tendency for software maintenance crisis in the software industry [Wang, 2005d]. The economical and technical reasons behind the software maintenance crisis will be explored and possible solutions will be presented in this subsection from a software industry organizational perspective.

14.3.3.1 The Mathematical Model of Software Maintenance Crisis

The *Software Legacy Maintenance Cost* (SLMC) *model* developed in Section 12.6.5 and Theorem 12.5 on the exponential growth of maintenance costs reveals that the ratio of maintenance cost C_m in a software development organization, r_m%, tends to exponentially increase over time, and it is proportional to the number of legacy systems N_L that the organization has produced.

Definition 14.22 *Software Maintenance Crisis* (SMC) is a phenomenon that happens when the demand for software maintenance exceeded the capability that a software development organization can provide, or when the costs of legacy software maintenance predominantly override the investment for new software development.

Based on Theorem 12.5 and Definition 14.19, the following theorem on software maintenance crisis can be derived.

The 48th Principle of Software Engineering

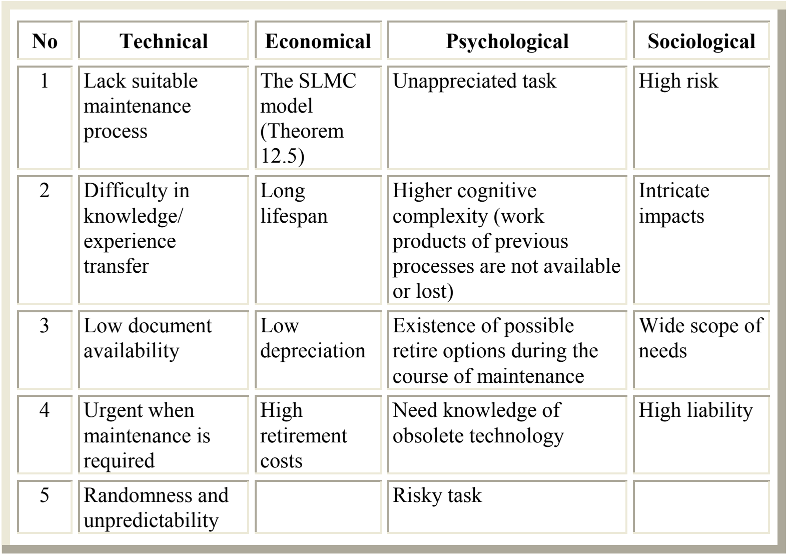
Theorem 14.1 The *mechanism of Software Maintenance Crisis* (SMC) states that a software development organization may face a situation known as the *software maintenance crisis*, in which the ratio of the maintenance costs r_m% is approaching 100% of the total costs that the organization spent.

A wide range of economic behaviors of the software industry may be explained on the basis of Theorems 14.1 and Theorem 12.5. For example, it explains why the usual lifespan of software systems is quite short, why software venders are voluntarily upgrading their systems from time to time, why a user would not expect to use a software system for a few decades, and why so many new software companies have been establishing while so many famous software brands have faded away in last decades.

14.3.3.2 Reasons behind Software Maintenance Crises

According to Law 39 of software engineering as stated in Theorem 10.10, it can be explained that SMC is actually a special phenomenon of software organization dissimilation in software engineering. The reasons behind SMC can be analyzed from the technical, economical, psychological, and sociological aspects as summarized in Table 14.4.

Table 14.4
Causal Analysis of Software Maintenance Crises



No	Technical	Economical	Psychological	Sociological
1	Lack suitable maintenance process	The SLMC model (Theorem 12.5)	Unappreciated task	High risk
2	Difficulty in knowledge/ experience transfer	Long lifespan	Higher cognitive complexity (work products of previous processes are not available or lost)	Intricate impacts
3	Low document availability	Low depreciation	Existence of possible retire options during the course of maintenance	Wide scope of needs
4	Urgent when maintenance is required	High retirement costs	Need knowledge of obsolete technology	High liability
5	Randomness and unpredictability		Risky task	

14.3.4.3 Solutions to Software Maintenance Crisis

The following solutions may be taken to deal with the SMC problems in software engineering and in the software industry.

a) *Enhance technologies:* i) To enhance software lifecycle processes to include software maintenance and retirement; ii) To increase depreciation of software systems; iii) To adopt a public agent acting like a library to store all code and documents of commercial software systems. Whenever the maintenance services can not continue as caused by an SMC, the design documentation and code will become a public resource. This measure will help to deal with document losses in individual organizations caused by programmer turnover, application migration across platforms, outsourced development, ceased maintenance, and shut downs of businesses.

b) *Software industry reorganization:* iv) To create a new affiliated service industry to maintain the legacy systems as that of garages for the automobile industry; v) To establish software insurance agencies who take responsibility for supporting any interrupted service of vendors.

c) *Honor the responsibility:* The current abnormal practice for releasing the liability in the software industry is either by shutting down the business (or change names of companies), or by forcing users to retire existing systems. Therefore, measures (i) through (v) proposed above should be adopted for establishing a more responsible software industry.

Learning from the automobile industry, one of the rational solutions to the problems of SMC may be derived below.

Corollary 14.1 There is a need of a sector in the software industry, known as the professional software legacy maintainers or "*software garages*."

When the society is highly dependent on the functionality of various software systems, the risks and impacts of SMC are too high to be ignored. Considering that the automobile industry and related maintenance garages have created a significant industrial sector for the societies on the wheels, a software maintenance sector will create a significant amount of important services in societies of the information era.

14.4 Essential Knowledge towards Excellent Software Engineers

Steven McConnell (1999) wrote: "An investment in learning software engineering principles is a particularly good investment for a software professional to make because that knowledge will last a whole career – not be half obsolete within three years."

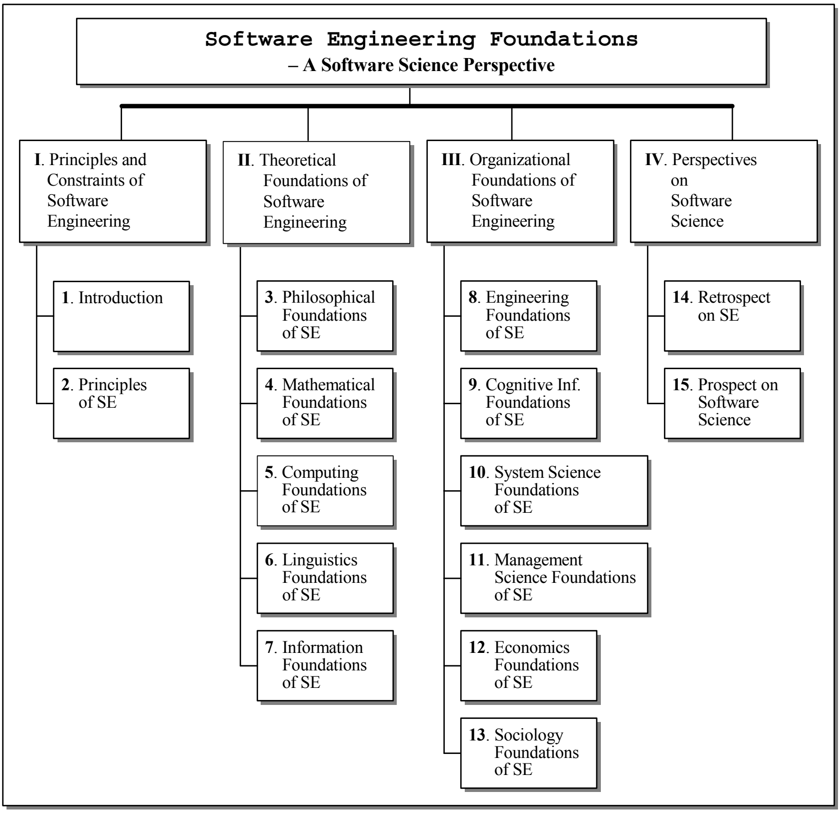

Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
1. Introduction
2. Principles of SE
3. Philosophical Foundations of SE
4. Mathematical Foundations of SE
5. Computing Foundations of SE
6. Linguistics Foundations of SE
7. Information Foundations of SE
8. Engineering Foundations of SE
9. Cognitive Inf. Foundations of SE
10. System Science Foundations of SE
11. Management Science Foundations of SE
12. Economics Foundations of SE
13. Sociology Foundations of SE
14. Retrospect on SE
15. Prospect on Software Science

Figure 14.5 Summary of the architecture of this book

The preceding chapters of this book have systematically explored the principles of software engineering in a rigorous and transdisciplinary approach. The principles of software engineering are formally documented as a comprehensive set of theorems and laws. This section summarizes the theoretical framework of software engineering principles and laws, which form the fundamental, durable, and enlightening knowledge for researchers and practitioners in software engineering.

As a summary of the architecture of this book, the key subject areas of software engineering foundations are highlighted in Fig. 14.5. Throughout this book, new theories for software engineering and related fields are developed, and formal treatments of existing theories and empirical practice are presented. This demonstrates the bidirectional impact of this work on the transdisciplinary investigation into the theoretical foundations of software engineering.

14.4.1 BASIC CONSTRAINTS OF SOFTWARE ENGINEERING

The essential knowledge on the 14 basic constraints of software engineering as developed in Chapter 1 can be summarized in Table 14.5. Further details and explanations of these constraints may be referred to Section 1.3.

Table 14.5
Basic Constraints of Software Engineering

No	Constraints	Description	Remark
1	**Cognition**	A set of innate cognitive attributes of software and the nature of the problems in software engineering that create the intricate relations of software objects and make software engineering inheritably difficult.	Def. 1.8
1.1	Intangibility	Software is abstract artifacts which is not constituted by physical objects or presence, and is difficult to be defined or expressed.	Def. 1.9
1.2	Complexity	Software is innately complex and its intricate internal connections and external couplings make it extremely difficult to be expressed or cognized.	Def. 1.10
1.3	Indeterminacy	The events, behaviors, or their sequence of occurring in a software system are not fully determinable on the basis of a given algorithm during design time; Instead, some of them may only be determined until run-time.	Def. 1.11
1.4	Diversity	The great variety of software in types, styles, architectures, behaviors, platforms, application domains, implementation techniques, usability, reliability, and quality.	Def. 1.12
1.5	Polymorphism	The approaches and styles of both software design and implementation are multifaceted and polyglottic.	Def. 1.13
1.6	Inexpressiveness	Software architectures and behaviors are inherently difficult to be expressed, modeled, represented, and quantified both formally and rigorously.	Def. 1.14
1.7	Inexplicit embodiment	Architectures and behaviors of software systems should be explicitly described by coherent symbolic notations in order to be processed and executed by computers.	Def. 1.15
1.8	Unquantifiable quality measures	The model of software quality has intricate facets and is difficult to be quantitatively modeled and measured.	Def. 1.16

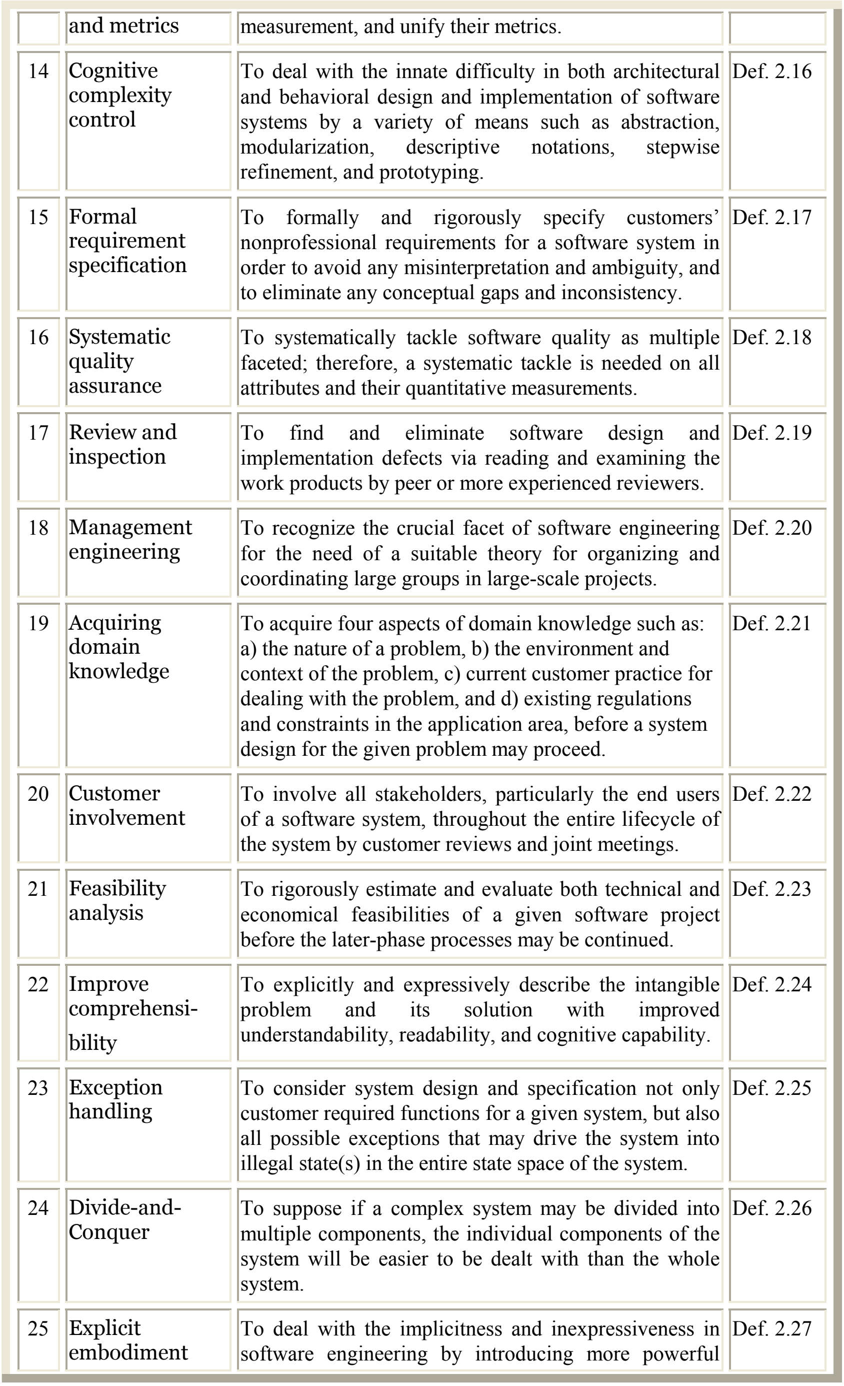
2	**Organization**	A set of coordinative and managerial requirements for software engineering that enables coordinative work to be efficiently carried out among a group of software engineers with different roles.	Def. 1.17
2.1	Time dependency	Almost all organizational issues in software engineering, such as software development scheduling, business goal of time to market, and labor allocation, are dependent on time.	Def. 1.18
2.2	Conservative productivity	Abstract artifacts and their relations in system designs need to be represented physiologically in the brain via growing synaptic connections, which is constrained by natural laws and its speed is not consciously controllable.	Def. 1.19
2.3	Labor-time interlock	The nature of software project organization is dominated by the extremely high interpersonal coordination rate, which prevents the workload (effort) from free decomposition into a sum of products of arbitrary amount of labor and periods of time.	Def. 1.20
3	**Resources**	The development costs and budgets, human resources, and the supporting and operating platforms of hardware.	Def. 1.21
3.1	Costs	Software engineering costs are incurred from both necessary and futility costs, and from both development and maintenance costs.	Def. 1.22
3.2	Human dependency	All software engineering activities and processes are human-based and constrained by basic human traits, cognitive and creative capabilities, as well as motivations and attitudes.	Def. 1.23
3.3	Hardware dependency	Software behaviors and functionality can only be embodied via the computing platform and related interactive I/O devices.	Def. 1.24

14.4.2 EMPIRICAL PRINCIPLES OF SOFTWARE ENGINEERING

Empirical software engineering principles are a set of fundamental and heuristic theories for software engineering. The essential knowledge on the 31 empirical principles of software engineering as developed in Chapter 2 can be summarized in Table 14.6. Further details and explanations of these empirical and heuristic principles may be referred to Section 2.2 through 2.4.

Table 14.6
Empirical Principles of Software Engineering

No	Principle	Description	Remark
1	Abstraction	To elicit essential properties of a set of objects while omitting inessential details of them.	Def. 2.3
2	Decomposition and modularization	To partition and divide the functions of a software system into individual modules or components.	Def. 2.4
3	Information hiding	To mask and simplify unnecessary information of software at a given level from the lower level details.	Def. 2.5
4	Engineering approach	To adopt the proven generic engineering methodology and practice in software development and its organization.	Def. 2.6
5	Professionalism	To recognize the competence or skills expected for a professional software engineer gained in training and practice.	Def. 2.7
6	Tools and environments	To adopt software development tools and software engineering supporting environment in order to facilitate efficient organization of coordinative work or extend human physical and intelligent capability in software development.	Def. 2.8
7	Documentation	To represent system design and architectures, record work products, maintain traceability of serial decisions, log problems and maintenance solutions, and enable postmortem analysis.	Def. 2.9
8	Stepwise refinement	To deductively extend a conceptual model of the requirement for a given software system by a series of expatiated and incremental specifications at increased degrees of details.	Def. 2.10
9	Prototyping	To evaluate or validate a design and feasibility of a required system based on the implementation of a prototype of the system.	Def. 2.11
10	Adopting engineering notations	To abstract, denote, and model user requirements and system specifications expressively and explicitly.	Def. 2.12
11	Process modeling	To deal with organizational and managerial issues in software engineering as well as software behaviors.	Def. 2.13
12	Reuse	To adopt higher-level building blocks, such as algorithms, methods, processes, patterns, frameworks, in order to improve efficiency, productivity, and quality of software engineering.	Def. 2.14
13	Measurements	To elicit generic software attributes, quantify their	Def. 2.15



	and metrics	measurement, and unify their metrics.	
14	Cognitive complexity control	To deal with the innate difficulty in both architectural and behavioral design and implementation of software systems by a variety of means such as abstraction, modularization, descriptive notations, stepwise refinement, and prototyping.	Def. 2.16
15	Formal requirement specification	To formally and rigorously specify customers' nonprofessional requirements for a software system in order to avoid any misinterpretation and ambiguity, and to eliminate any conceptual gaps and inconsistency.	Def. 2.17
16	Systematic quality assurance	To systematically tackle software quality as multiple faceted; therefore, a systematic tackle is needed on all attributes and their quantitative measurements.	Def. 2.18
17	Review and inspection	To find and eliminate software design and implementation defects via reading and examining the work products by peer or more experienced reviewers.	Def. 2.19
18	Management engineering	To recognize the crucial facet of software engineering for the need of a suitable theory for organizing and coordinating large groups in large-scale projects.	Def. 2.20
19	Acquiring domain knowledge	To acquire four aspects of domain knowledge such as: a) the nature of a problem, b) the environment and context of the problem, c) current customer practice for dealing with the problem, and d) existing regulations and constraints in the application area, before a system design for the given problem may proceed.	Def. 2.21
20	Customer involvement	To involve all stakeholders, particularly the end users of a software system, throughout the entire lifecycle of the system by customer reviews and joint meetings.	Def. 2.22
21	Feasibility analysis	To rigorously estimate and evaluate both technical and economical feasibilities of a given software project before the later-phase processes may be continued.	Def. 2.23
22	Improve comprehensibility	To explicitly and expressively describe the intangible problem and its solution with improved understandability, readability, and cognitive capability.	Def. 2.24
23	Exception handling	To consider system design and specification not only customer required functions for a given system, but also all possible exceptions that may drive the system into illegal state(s) in the entire state space of the system.	Def. 2.25
24	Divide-and-Conquer	To suppose if a complex system may be divided into multiple components, the individual components of the system will be easier to be dealt with than the whole system.	Def. 2.26
25	Explicit embodiment	To deal with the implicitness and inexpressiveness in software engineering by introducing more powerful	Def. 2.27

		descriptive means at a higher level of abstraction and precision.	
26	Establishing theoretical foundations	To elicit rigorous theories and generic laws once there are a wide variety of observed phenomena and alternative practices.	Def. 2.28
27	Architecture and behavior modeling	To understand software system models are a hybrid model where both architectures and behaviors should be coherently described.	Def. 2.29
28	Standardization	To integrate, regulate, unify, and optimize existing principles, best practices, and industrial norms into standards.	Def. 2.30
29	Systems engineering	To adopt system science theories and approaches to deal with complicated architectures and behaviors of software.	Def. 2.31
30	Engineering organization	To recognize that the organization issue is as important as that of pure technical and the cognitive issues in software engineering.	Def. 2.32
31	Cognitive engineering	To be aware that the cognitive complexity is the dominant problem in almost all processes of software design, implementation, and maintenance, which should be tackled by cognitive informatics theories.	Def. 2.33

14.4.3 LAWS OF SOFTWARE ENGINEERING

A law of software engineering is a proven statement of a causality between a deducted result and its formal conditions. The essential knowledge on laws of software engineering developed in this book can be summarized in Table 14.7 highlighted by the 50 Wang's laws. Further details and explanations of these set of laws of software engineering may be referred to previous chapters using the links provided under the title of each law.

Table 14.7
Laws of Software Engineering

No	Law	Description	Mathematical model
1	The characteristics of theoretical and empirical problems	*Software engineering problems* must be treated by both *theoretical* and *empirical* methodologies. The former is characterized by abstract, inductive, mathematics-based, and formal-inference-centered studies; while the latter is characterized by concrete,	

	(Theorem 1.1)	deductive, data-based, and experimental-validation-centered studies.	
2	The Information-Matter-Energy (IME) model (Theorem 1.2)	The natural world (*NW*) which forms the context of human intelligence and software science is a dual world: one aspect of it is the *physical* world (*PW*), and the other is the *abstract* world (*AW*), where *matter* (*M*) and *energy* (*E*) are used to model the former, and *information* (*I*) to the latter, where p, a, and n are functions that determine a certain *PW*, *AW*, or *NW*, respectively.	$NW \mathrel{\hat{=}} PW \,\|\, AW$ $= p(M, E) \,\|\, a(I)$ $= n(I, M, E)$
3	Abstract objects under study (Theorem 1.3)	The *nature of software* stems from intangibility of the abstract objects under study, intricate inner connections of software systems, adaptive interactions to external events and environments, and the cognitive complexity to explicitly describe them.	
4	Explicit descriptivity (Theorem 1.4)	Only a *higher-level abstract, precise,* and *rigorous means* is adequate to express an object at a given level of abstraction, where denotational mathematics is the top-level abstraction means.	
5	The basic constraints of SE (Theorem 1.5)	Software engineering faces the *cognitive, organizational,* and *resources constraints*.	
6	Conservative productivity (Theorem 1.7)	Software productivity is physiologically constrained by the growing speed of synaptic connections inside the brain, because before any creative artifact is generated externally, it must be created and represented physiologically inside the brain by the synaptic connections.	
7	Universal constraints (Theorem 3.1)	Both the natural world and the perceived abstract world are constrained by certain known restrictions and laws, or by those yet to be known due to both current limitations of natural resources and/or human cognitive capability.	
8	Law of causality	A condition must be both necessary and sufficient to qualify as a cause, where the *necessary* condition is a condition that must be present in order for the	

	(Theorem 3.3)	effect to occur, while the *sufficient* condition is a condition that will always produce the effect.	
9	Inclusive intelligent capability (Theorem 3.5)	*Artificial intelligence* (AI) is a subset of *natural intelligence* (NI).	AI $\subseteq$ NI
10	Behavior space of software (Theorem 3.11)	The software behavior space Ω is innately three-dimensional, which can be described by a Cartesian product of computational operations *OP*, time *T*, and memory space *S*.	$\Omega = OP \times T \times S$
11	Utility of mathematics (Theorem 4.1)	*Denotational mathematics* is the means and rules to rigorously and explicitly express design notions and conceptual models on abstract architectures and complex interactive behaviors at the highest level of abstraction and in the largest scope of systems.	
12	Cumulative Relational Model (CRM) of processes (Theorem 4.3)	A *process* $\mathfrak{P}$ is the basic unit of an applied computational behavior that is composed by a set of statements s_i, $1 \leq i \leq n\text{-}1$, with left-associated cumulative relations, where $s_i \in \mathfrak{P}$ and $r_{ij} \in \mathfrak{R}$.	$\mathfrak{P} = \underset{i=1}{\overset{n-1}{R}} (s_i\ r_{ij}\ \mathrm{s}_j), j = i+1$ $= (...(((s_1) r_{12}\ \mathrm{s}_2) r_{23}\ \mathrm{s}_3) \ldots r_{n-1,n}\ \mathrm{s}_n)$
13	Express power of algebraic modeling (Theorem 4.8)	The *express power of RTPA* states that the total number of the possible computational operations $\mathcal{N}$ is a set of combinations between two arbitrary meta processes $\mathbb{P}_1, \mathbb{P}_2 \in \mathfrak{P}$ composed by each of the process relations $\mathbb{R} \in \mathfrak{R}$ in RTPA.	$\mathcal{N} = \#\mathfrak{R} \bullet C^2_{\#\mathfrak{P}}$ $= 17 \bullet \frac{17!}{2!(17\text{-}2)!}$ $= 17 \bullet 136$ $= 2{,}312$
14	Essential facets of software system modeling (Theorem 4.9)	Software systems can be formally specified by its *architectures, static behaviors,* and *dynamic behaviors* with multiple-level refinements.	
15	The root of computing and information science (Theorem 5.1)	The *most fundamental data object model* shared in both computing and information science is *binary digits* (bits).	
16	Domain	Letting D_m, D_l, and D_u be the domains of	$D_u \subseteq D_l \subseteq D_m$

	constraints of data objects (Theorem 5.6)	*mathematical* (logical), *language defined,* or *user defined,* respectively, the following relationship between the domains of an identifier in programming is always held.	
17	The generic mathematical model of programs (Theorem 5.7)	A software system or a program $\wp$ is a set of complex embedded cumulative relational processes P_k dispatched by system-level events e_k .	$\wp = \overset{m}{\underset{k=1}{R}}(@e_k \mathbf{s} \hookrightarrow P_k)$ $= \overset{m}{\underset{k=1}{R}}[@e_k \mathbf{s} \hookrightarrow$ $\overset{n-1}{\underset{i=1}{R}}(s_i(k)\, r_{ij}(k)\, \mathrm{s}_j(k))],$ $j = i+1$
18	Tradeoff between syntaxes and semantics (Theorem 6.1)	In the DGE system, the complexities of the syntactic rules (or grammar) C_{syn} and of the semantic rules C_{sem} are inversely proportional, i.e.:	$C_{syn} \propto \frac{1}{C_{sem}}$
19	Asynchronicity of program semantics (Theorem 6.2)	The semantics of a relatively timed program is invariant with the changes of executing speed, as long as any absolute time constraint is met.	
20	The least complete set of instructions in programming (Theorem 6.3)	A program is *composable* with sufficient descriptive power in a given language *iff* both the sufficient sets of meta instructions ($\mathfrak{P}$, Theorem 4.6) and compositional rules ($\mathfrak{R}$, Theorem 4.7) are rigorously defined.	
21	Informatics laws of software (Theorem 7.2)	Software architectures, behaviors, and processes are constrained by the 19 *informatics* laws of basic information properties.	
22	Conservation of basic engineering constraints (Theorem 8.2)	The three basic constraints of engineering goals known as time (T), costs (C), and utility (U) are conservative in a given engineering context, where both δ and k are a constant.	$f_t(T^{-1}) + f_c(C^{-1}) + f_u(U)$ $= \quad k\frac{U}{T \bullet C}$ $= \delta$

23	Coordinative workload in engineering (Theorem 8.4)	The *actual workload W* of a coordinative project is a function of the average interpersonal coordination rate *r* and the number of labor *L* in the project, where T_1 is the *indicative duration* needed to complete the work by only one person, and W_1 is the *ideal workload* without the interpersonal overhead *h* or that of a single person project.	$W = L \bullet T$ $= L \bullet T_1(1+h)$ $= W_1(1+h)$ $= W_1(1 + r \bullet \frac{L(L-1)}{2})$ [PM]
24	Interchange-ability of labor and time (ILT) (Theorem 8.6)	For a given workload *W*, labor *L* and duration *T* are transformable under the condition as given in the mathematical model.	$T = \frac{W}{L}$ $= \frac{W_1}{L}(1 + r \bullet \frac{L(L-1)}{2})$ $= \frac{W_1}{L}(\frac{1}{2}rL^2 - \frac{1}{2}rL + 1)$ $= \frac{1}{2}W_1(rL - r + \frac{2}{L})$
25	The shortest duration of coordinative work (Theorem 8.7)	There exists the *shortest duration* T_{min} under the *optimum labor allocation* L_0 for a given ideal workload W_1 with a certain interpersonal coordination rate *r*.	$\begin{cases} T_{\min} = \{T \mid L = L_0\} \\ \quad = \frac{1}{2}W_1(rL_0 - r + \frac{2}{L_0})\ [M] \\ L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil,\ r \neq 0 \quad [P] \end{cases}$
26	Quantitative advantage of human brain (Theorem 9.1)	The magnitude of the memory capacity of the brain is tremendously larger than that of the closest species.	
27	Qualitative advantage of human brain (Theorem 9.2)	The possession of the abstract layer of memory and the abstract reasoning capacity makes the human brain profoundly powerful on the basis of the quantitative advantage.	
28	Generic forms of information (Theorem 9.4)	There are four categories of internal information $\mathcal{I}$ in the brain known as *knowledge* (*K*), *behaviors* (*B*), *experience* (*E*), and *skills* (*S*).	$\mathcal{I} = (\mathcal{I}_k, \mathcal{I}_b, \mathcal{I}_e, \mathcal{I}_s)$
29	The nature of intelligence (Theorem 9.5)	*Intelligence* $\Im$ is a capability that transfers between data, information, knowledge, and behaviors known as the *perceptive intelligence* $\Im_p$, *cognitive intelligence* $\Im_c$, *instructive intelligence* $\Im_i$, and *reflective intelligence* $\Im_r$.	$\Im \triangleq \Im_p : D \rightarrow I$ (Perceptive) $\parallel \Im_c : I \rightarrow K$ (Cognitive) $\parallel \Im_i : I \rightarrow B$ (Instructive) $\parallel \Im_r : D \rightarrow B$ (Reflective)
30	Dynamic properties of neural clusters	The LTM is dynamic. New neurons (to represent *objects or attributes*) are assigning, and new synaptic connections	

	(Theorem 9.9)	(to represent *relations*) are creating and reconfiguring all the time in the brain.	
31	Establishment cycle of LTM (Theorem 9.11)	The cycle of LTM establishment requires at least 24 hours, where the 24-hour cycle includes any kind of combinations of awake, asleep, and siesta.	*LTM establishment* $cycle \geq 24$ [hrs]
32	Holism complexity of systems (Theorem 10.1)	Within the 7-level magnitudes of systems, known as the *empty, small, medium, large, giant, immense,* and *infinite* systems, almost all systems are too complicated to be cognitively understood or mentally handled as a whole, except small systems or those that can be decomposed into small systems.	
33	Generic topology of normalized systems (Theorem 10.2)	Systems tend to be normalized into a hierarchical structure in the form of a complete *n*-nary tree.	
34	System gain of functionality (Theorem 10.4)	System conjunction or composition between two systems S_1 and S_2 creates *new relations* ΔR_{12} and/or *new behaviors* (functions) ΔB_{12} that are solely a property of the newly established super system *S*, which can be determined by the sizes of the two intersected component sets $\#C_1$ and $\#C_2$.	$\Delta R_{12} = \#R - (\#R_1 + \#R_2)$ $= (\#(C_1 + C_2))^2 -$ $((\#C_1)^2 + (\#C_2)^2)$ $= 2\ (\#C_1 \bullet \#C_2)$
35	System mutation (Theorem 10.5)	The gradual increment of quantity of system, i.e., ΔC or ΔR, in a system beyond the point of the critical mass Q_{cm} triggers the abrupt generation of functionality (quality) F_{cm} of the system.	
36	System gain of work (Theorem 10.6)	Work done by a system is always greater than any of its components, but must not greater than the sum of those of its components	$\begin{cases} W(S) \leq \sum_{i=1}^{n} W(C_i), & \eta \leq 100\% \\ W(S) > \max(W(C_i)), & C_i \in E_S \end{cases}$
37	Conservative work of equilibrium systems (Theorem 10.9)	The sum of all types of work is always zero in an equilibrium system, where $W(C_i)$ is the abstract work of a system component C_i.	$\sum_{i=1}^{n} W(C_i) = 0$

38	Conditions of system self-organization (Theorem 10.10)	The *necessary* and *sufficient* condition of self-organization is the existence of at least one minimum on the state curve of a system *f*(*x*), which satisfies the following requirements, where *f'*(*x*) and *f "*(*x*) are the first and second order derivatives of *f*(*x*) on (a, b).	$\begin{cases} f'\,(x_{min} \mid x_{min} \in\ (a,b)) = 0 \\ f''(x_{min} \mid x_{min} \in\ (a,b)) \neq 0 \end{cases}$ $\begin{cases} f'\,(x_{min} \mid x_{min} \in\ (a,b)) = 0 \\ f''(x \mid x < x_{min} \in\ (a,b)) < 0 \\ f''(x \mid x > x_{min} \in\ (a,b)) > 0 \end{cases}$
39	System synchronization (Theorem 10.11)	A system reaches its maximum utility $\vec{S}_{\max}$ when all components' efforts $\vec{S_1}$ and $\vec{S_2}$ are synchronized.	$\begin{cases} \vec{S} = \vec{S_1} + \vec{S_2} \\ \vec{S}_{\max} = \ \mid \vec{S_1} \mid + \mid \vec{S_2} \mid \end{cases}$
40	System dissimilation (Theorem 10.12)	Any system tends to undergo a continuous degradation that leads to the eventual loss of its designed utility and against its initial purposes to form the system.	
41	Cognitive complexity of software (Theorem 10.14)	The *cognitive complexity* of a software system *S*, *C*$C_c(S)$, is a product of the operational complexity $C_{op}(S)$ and the architectural complexity $C_a(S)$.	$\begin{aligned} S_f(S) &= C_{op}(S) \bullet C_a(S) \\ &= \{\sum_{k=1}^{n_C}\sum_{i=1}^{m_k} w(k,i)\} \bullet \\ &\quad \{\sum_{k=1}^{n_{CLM}} \text{OBJ}(CLM_k) \\ &\quad + \sum_{k=1}^{n_C} \text{OBJ}(C_k)\}\ \text{[FO]} \end{aligned}$
42	Gain of management (Theorem 11.1)	Management is required to reduce the complexity of working group organization, to improve the efficiency of groups (*e*(*n*)), and to simplify the forms of interpersonal coordination.	$\begin{aligned} e(n) &= \frac{\Delta m(n)}{C_2(n)} \bullet 100\% \\ &= (1 - \frac{c_m(n)}{c_2(n)}) \bullet 100\% \\ &= (1 - \frac{\text{n+1}}{n \bullet (n\text{ - }1)}) \bullet 100\% \end{aligned}$
43	Gain of division of labor (Theorem 11.2)	The relative gain $g_r(k)$ via division of labor in work organization is proportional to the repetitive times *k* at specialized subtask-level, where *c* is a positive constant, 1 < *c* < *e*.	$\begin{aligned} g_r(k) &= \frac{E(k)\text{ - }E_d(k)}{E(k)} \bullet 100\% \\ &= (1\text{-}\frac{E_d(k)}{E(k)}) \bullet 100\% \\ &= (1\text{-}\frac{\sum_{i=1}^{k} \frac{1}{(\frac{e}{c})^{k\text{-}1}}}{k}) \bullet 100\% \end{aligned}$

44	Adaptive economic equilibrium (Theorem 12.1)	A market with autonomic interactions between demands *D* and supplies *S* is a self-regulated and self-adaptive system, where any change in demand, supply, or both will be autonomously adjusted via the leverage of price *P* to an equilibrium.	$\left\{\begin{matrix} \left\langle\begin{matrix} D\uparrow\rightarrow \\ S\downarrow\rightarrow \end{matrix}\right\rangle \rightarrow P\uparrow \Rightarrow \left\langle\begin{matrix} D\downarrow\rightarrow \\ S\uparrow\rightarrow \end{matrix}\right\rangle \rightarrow P\downarrow \\ \left\langle\begin{matrix} D\downarrow\rightarrow \\ S\uparrow\rightarrow \end{matrix}\right\rangle \rightarrow P\downarrow \Rightarrow \left\langle\begin{matrix} D\uparrow\rightarrow \\ S\downarrow\rightarrow \end{matrix}\right\rangle \rightarrow P\uparrow \end{matrix}\right\}$ $[\frac{\text{Market conservation}}{\text{Lemma 12.xx}}]+[\frac{\text{Maximizing profits}}{\text{Lemma 12.xx}}]$
45	Formal Economic Model of Software Engineering Cost (FEMSEC) (Theorem 12.3)	On the basis of the workload-driven project organization laws, the expected project cost *C* can be rigorously determined with the optimal labor allocation L_0 and the shortest duration T_{min} by the following 6 steps: 1) Estimate the project size $\overline{S_p}$ 2) Determine the ideal workload W_1 3) Allocate the optimal labor L_0 4) Determine the shortest duration T_{min} 5) Determine the expected workload *W* 6) Determine the expected project cost *C*	$\overline{S_p}=\frac{1}{6}(S_{max}+4S_{exp}+S_{min})[\text{kLOC}]$ $W_1 = \frac{\overline{S_p}}{\rho} \bullet 12 \quad [\text{PM}]$ $L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil \quad [P]$ $T_{\min} = \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0})$ $W = \frac{1}{2} W_1$ $(rL^2{}_0 - rL_0 + 2)$ [PM] $C = L_0 \bullet T_{min} \bullet C_L \ [\$]$
46	Basic essences for evolution (Theorem 13.1)	The *basic evolutional needs* of mankind are to preserve both the species' biological traits via *gene pools*, and the cumulated knowledge via various *information systems*.	
47	Organiza-tional coordination efficiency (Theorem 13.3)	The natural constraints for social organization that forces the architecture of large groups to be evolved and adapted to tree-form hierarchical structures in an organization is the need to maintain acceptable coordinating efficiency at each level of the organization tree.	
48	Time-oriented optimization for large-scale project organization (Theorem 13.4)	*Time-oriented optimization for large-scale project organization* states that in order to further reduce the shortest duration T_{min} of an entire large-scale project constrained by Theorem 8.7, the optimal form of organization is to evenly partition the whole project into *n* lightly-coupled parallel subprojects that may be conducted by independent groups with a shorter duration T^i_{min}, $1 \le i \le n$, so that an average *n*-fold time	$\overline{T^i{}_{\min}} = \frac{1}{n}\sum_{i=1}^{n} T^i{}_{\min}$ $= \frac{1}{n} T_{\min} + \varpi$

		deduction can be gained.	
49	The n-fold error reduction structure (Theorem 13.5)	The *error rate of a work product* can be reduced up to *n* folds from the average error rate of individuals r_e in a coordinative group via *n*-nary *peer reviews* based on the random nature of error distributions and independent nature of error patterns of individuals.	$R_e = \prod_{k=1}^{n} r_e(k)$
50	Power of multi-disciplinary knowledge (Theorem 14.2)	The *ratio of knowledge space* Ω_Σ between the knowledge of an expert with coherently *m* disciplinary knowledge K_Σ and that of a group of *m* experts with separated individual disciplinary knowledge K_m is shown in the mathematical model, where *n* is the number of *average knowledge objects* or concepts in the disciplines.	$\Omega_\Sigma(m,n) = \frac{K_\Sigma}{K_m} = \frac{\mathrm{C}_{m \bullet n}^{2}}{\sum_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{C}_{n}^{2}} = \frac{\frac{(mn)!}{2!(mn\text{-}2)!}}{\frac{m(n)!}{2!(n\text{-}2)!}} \approx \frac{(mn)^2}{mn^2} = m$

14.4.4 FORMAL PRINCIPLES OF SOFTWARE ENGINEERING

Formal principles of software engineering are a systematical elicitation and formalization of new and conventional heuristic principles for software engineering. The essential knowledge on theoretical principles of software engineering developed in this book can be summarized in Table 14.8, highlighted by the 51 Wang's principles. Further details and explanations of these theoretical principles may be referred to previous chapters using the links provided under the title of each principle.

Table 14.8
Formal Principle of Software Engineering

No	Principle	Description	Mathematical model
1	Polymorphous solutions (Theorem 1.6)	The solution space *SS* of software engineering for a given problem is a product of the number of possible design options N_d and the number of possible implementation options N_i.	$SS = N_d \bullet N_i$

2	Formalization of principles (Theorem 2.1)	The empirical principles for software engineering are heuristic and data-based; while the formal principles for software engineering are rigorous and mathematics-based, which are elicited and refined from the empirical principles.	
3	Validation of abstract propositions (Theorem 3.2)	The abstract and information-based propositions and work products, such as a design or a specification of a system, is bounded by logical verifications, mathematical proofs, systematical reviews, behavioral simulations and tests, and/or in field trials.	
4	Compatible intelligent capability (Theorem 3.4)	*Natural intelligence* (NI) and *artificial intelligence* (AI) are compatible by sharing the same mechanisms of intelligent capability.	$\text{AI} \propto \text{NI}$
5	Deductive inference (Theorem 3.6)	Given an arbitrary nonempty set $\mathbf{X}$, let $p(x)$ be a proposition for $\forall x \in \mathbf{X}$, a specific conclusion on $\exists a \in \mathbf{X}$, $p(a)$ can be drawn as in the mathematical models.	$\forall x \in \mathbf{X}, p(x) \vdash \exists a \in \mathbf{X}, p(a)$ or $(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash$ $(\exists a \in \mathbf{X}, p(a) \Rightarrow q(a))$
6	Inductive inference (Theorem 3.7)	If $\exists a, k, succ(k) \in \mathbf{X}$, $p(a)$ and $p(k) \Rightarrow p(succ(k))$ are three valid propositions, then a generic conclusion on $\forall x \in \mathbf{X}$, $p(x)$ can be drawn as in the mathematical models.	$((\exists a \in \mathbf{X}, p(a)) \wedge$ $(\exists k, succ(k) \in \mathbf{X}, (p(k) \Rightarrow p(succ(k)))) \vdash \forall x \in \mathbf{X}, p(x)$ or $((\exists a \in \mathbf{X}, p(a) \Rightarrow q(a)) \wedge (\exists k, succ(k) \in \mathbf{X}, ((p(k) \Rightarrow q(k)) \Rightarrow (p(succ(k)) \Rightarrow q(succ(k))))) \vdash$ $\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)$
7	Abductive inference (Theorem 3.8)	Based on a general implication $\forall x \in \mathbf{X}$, $p(x) \Rightarrow q(x)$, a specific conclusion on $\exists a \in \mathbf{X}$, $p(a)$ can be drawn as in the mathematical models.	$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash$ $(\exists a \in \mathbf{X}, q(a) \Rightarrow p(a))$ or $(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x) \wedge r(x) \Rightarrow q(x)) \vdash (\exists a \in \mathbf{X}, q(a) \Rightarrow (p(a) \vee r(a)))$
8	Analogical inference	Based on a specific predicate $\exists a \in \mathbf{X}$, $p(a)$, a similar specific conclusion can be drawn *iff* $\exists x \in \mathbf{X}$, $p(x)$ as in the	$\exists x \in \mathbf{X}, p(x) \wedge \exists a \in \mathbf{X}, p(a) \vdash \exists b \in \mathbf{X} \wedge b \neq a, p(b)$ or

	(Theorem 3.9)	mathematical models.	$(\exists x \in \mathbf{X}, p(x) \wedge \exists a \in \mathbf{X}, p(a) \Rightarrow q(a)) \vdash (\exists b \in \mathbf{X} \wedge \mathrm{b} \neq a, p(b) \Rightarrow q(b))$
9	Necessary and sufficient conditions of software usage (Theorem 3.10)	Those that warrant the requirements for software solutions are the system behaviors of *repeatability*, *programmability*, and *run-time determinability*.	
10	Principle of abstraction (Theorem 4.2)	Given an arbitrary set *X* and any property *p*, there is a set *A* such that the elements of *A* are exactly those members of *X* which have the property *p*.	$A = \{a \mid a \in X \wedge p(a)\}$
11	Primary types of computational objects (Theorem 4.4)	The *RTPA type system* $\mathfrak{T}$ encompasses 17 primitive types elicited from fundamental computing needs.	$\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @_{e}\mathbf{S}, @_{t}\mathbf{TM}, @_{int}\odot, \circledS_{s}\mathbf{BL}\}$
12	Type equivalence (Theorem 4.5)	Two types $\mathbb{T}_1$ and $\mathbb{T}_2$ are *equivalent, iff* the domain of type $\mathbb{T}_1$ is either identical to or a subset of that of $\mathbb{T}_2$.	$\mathbb{T}_1(x) = \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y)$ or $\mathbb{T}_1(x) \subseteq \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y)$
13	Meta software processes (Theorem 4.6)	The *RTPA meta process system* $\mathfrak{P}$ encompasses 17 fundamental computational operations elicited from the most basic computing needs.	$\mathfrak{P} = \{:=,$ �, $\Rightarrow, \Leftarrow, \nLeftarrow, \gtrdot, \lessdot, \mid\gtrdot, \mid\lessdot, @, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\}$
14	Software composing rules (Theorem 4.7)	The *RTPA process relation system* $\mathfrak{R}$ encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs.	$\mathfrak{R} = \{\rightarrow, \curvearrowright, \mid, \mid\ldots\mid\ldots, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \parallel, \oiint, \mid\mid\mid, \gg, \lightning, \hookrightarrow_{t}, \hookrightarrow_{e}, \hookrightarrow_{i}\}$
15	The primitive computational behaviors (Theorem 5.2)	The *most fundamental computational operations* are logical, arithmetic, and memory access operations on *bits*.	
16	Nature of requirements and specifications (Theorem 5.3)	Requirement elicitation focuses on desired functions of a system δ, while system specification focuses on the entire behavioral space of the system Ω, including both δ and the undesired but potential system transitions represented by $\overline{\delta}$ in the behavioral space.	$S_{\Omega} = \#\delta + \#\overline{\delta}$ $= \#S \bullet \#\Sigma$

17	The weaknesses of automata (Theorem 5.4)	Automata and FSMs as a system composition and modeling method built on event-driven mechanisms are inadequate to model the complete basic computational requirements, particularly the lack of the descriptive power for: a) System architectures and data objects modeling; b) Nonevent-driven transitional process modeling; c) Detailed behavioral descriptions; d) Mathematical operations and processing of complicated languages.	
18	Fundamental computational capabilities (Theorem 5.5)	The essential capabilities for computation are as follows: • A memory for storing bit information; • A simple addressing capability for accessing information in the memory; • Read/write operations for retrieving or updating the memory; • A conditional and quantitative evaluation capability for interpreting the inputted information; • A stored-information-driven mechanism for determining the next step.	
19	Primitive form of information (Theorem 7.1)	The most fundamental form of information that can be represented and processed is binary digit where *k* = *b* = *2*.	$I_b = f : M \to S_b$ $= \lceil \log_b M \rceil$ $= \lceil \log_2 M \rceil \; [bit]$
20	Relationship between a hypothesis and a theory (Theorem 8.1)	The necessary and sufficient conditions for a hypothesis $H_g(C, O, G, P, F)$ to be proven as a theory $\mathcal{T}$ are *iff* it fulfills the following criteria.	$H_g \vdash \mathcal{T}$, *iff* $C \wedge O \wedge G \wedge P \wedge F = \mathbf{T}$
21	Engineering Maturity Model (EMM) (Theorem 8.3)	The applied engineering disciplines have four maturity levels known as the levels of *emergence* (L_1), *art* (L_2), *engineering* (L_3), and *post-engineering* (L_4).	$EMM : L_1 \subseteq L_2 \subseteq L_3 \subseteq L_4$
22	Incompre-ssible workload	A given ideal workload W_1 in software engineering can not be compressed by any kind of labor allocation, and in the best case when there is only one person involved, the minimum workload $W =$	$W \geq W_1 = W_{min}$

	(Theorem 8.5)	$W_1 = W_{min}$ may be reached.	
23	Exchange-ability from labor to time (Theorem 8.8)	The *exchange rate from labor to time* $\gamma_{L\sim T}$ in a coordinative work organization is determined by the ratio between the increment of time ΔT and the increment of labor ΔL.	$\gamma_{L\sim T} = \frac{\Delta T}{\Delta L} = \frac{T_1 \text{ - } T_{min}}{L_0 \text{-} L_1}$ [M/P]
24	Exchange-ability from time to labor (Theorem 8.9)	The *exchange rate from time to labor* $\gamma_{T\sim L}$ in a coordinative work organization is determined by the ratio between the increment of labor ΔL and the increment of time ΔT.	$\gamma_{T\sim L} = \frac{\Delta L}{\Delta T} = \frac{L_0 \text{ - } L_1}{T_1 \text{ - } T_{\min}}$ [P/M]
25	Constraint on group size in coordinative work (Theorem 8.10)	There exists an upper limit of group size S_{max} in coordinative work organization in software engineering. Therefore, large projects must be partitioned into multiple parallel groups that each of the groups obeys the same natural constraint.	$S_{max} = \max(L_0(r)) = 20$ [P]
26	The risk of nonoptimal work organization (Theorem 8.11)	The risks $\mathcal{R}$ due to irrational decisions of work organization are proportional to the coordination rate r in a project. That is, the higher the r, the higher the risk under nonoptimal labor allocation.	$\mathcal{R} \propto r$
27	Cognitive Models of Memory (CMM) (Theorem 9.3)	The architecture of human memory is parallel configured by the Sensory Buffer Memory (SBM), Short-Term Memory (STM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM).	CMM $\triangleq$ SBM $\|$ STM $\|$ LTM $\|$ ABM
28	Generic forms of learnings (Theorem 9.6)	There are sufficiently four categories of learning $\mathcal{L}$ known as those of *knowledge* ($\mathcal{L}_k$), *behaviors* ($\mathcal{L}_b$), *experience* ($\mathcal{L}_e$), and *skills* ($\mathcal{L}_s$).	$\mathcal{L} = (\mathcal{L}_k, \mathcal{L}_b, \mathcal{L}_e, \mathcal{L}_s)$
29	Representa-tion of learning results (Theorem 9.7)	The internal memory in the form of the *OAR* structure can be updated by a conjunction between the existing *OAR* and the newly created sub-*OAR*.	$OAR'\textbf{ST} \triangleq OAR\textbf{ST} \cup sOAR\textbf{ST}$ $= OAR\textbf{ST} \cup (O_s, A_s, R_s)$
30	Principal intelligent advantages (Theorem 9.8)	On the basis of two principal advantages known as the *qualitative* properties (Theorem 9.1) and *quantitative* properties (Theorem 9.2), humans gain the power as the most intelligent species	

		in the world.	
31	Cognitive mechanism of sleeping (Theorem 9.10)	Sleeping is a subconscious process for LTM establishment.	*Cognitive purpose of sleep* *= LTM establishment*
32	Mechanism of LTM establishment (Theorem 9.12)	The entire memory of information represented as an OAR model in the brain is updated by incorporating the sub-OARs formed in STM based on the following selective criteria: a) A new sub-OAR in STM was more frequently used in the previous 24 hours; b) A new sub-OAR in STM was related to the existing OAR in LTM at a higher level of the neural cluster hierarchy; c) A new sub-OAR in STM was given special attention so that it obtained a higher retention weight.	
33	Equivalence between open and closed systems (Theorem 10.3)	An open system *S* and a closed system $\widehat{S}$ in the same context is transformable when their environments Θ_S and $\Theta_{\widehat{S}}$ $(\Theta_{\widehat{S}} = C \not\subset \widehat{S})$ are taken into consideration, respectively.	$\begin{cases} \widehat{S} = \mathrm{S} \sqcup \Theta_{\mathrm{S}} \\ \mathrm{S} = \widehat{S} \sqcup \Theta_{\hat{\mathrm{S}}} \end{cases}$
34	The bottleneck principle of systems (Theorem 10.7)	The output work of a serial system $W(S_s)$ is determined by the least powerful component of the system.	$W(S_s) = \min\ (\mathrm{W}(C_i) \mid C_i \in C_s \wedge 1 \leq i \leq n))$
35	The linear sum principle of systems (Theorem 10.8)	The output work of a parallel system $W(S_p)$ is a sum of the work done by all its components less the overhead of the system ϖ.	$W(S_p) = \sum_{i=1}^{\mathrm{n}} W(C_i)$ - $\varpi,$ $C_i \in C_p,\ \varpi > 0$
36	Orientation of software engineering complexity theories (Theorem 10.13)	The complexity theories of computation and software engineering are different. The former is focused on the problems of *high throughput complexity* that are computing *time efficiency* centered; while the latter puts emphases on the problems of *functional complexity* that are human *cognition time* and *workload* oriented.	

37	Normalized software system architectures (Theorem 10.15)	Components of different subsystems should not be coupled directly, rather than be invoked through their top layer components shared in the same subsystem.	
38	Properties of games (Theorem 11.3)	A formal game G is *deterministic* and *conservative*. That is, once the game $G = (P, D, M, S)$ is set, the properties of G are determined and predictable, but not changeable by any player in the game.	
39	Conditions of win-win decisions (Theorem 11.4)	The states that a win-win decision can be achieved when the following condition of a nonzero-sum game is satisfied, where σ is the sum of the game that is a positive nonzero constant, s_i is the expected score of player i, and n_s is the number of sets of matches in the game.	$\sigma \geq \frac{1}{n_s}\sum_{i=1}^{n} s_i$
40	Property of decision grids (Theorem 11.5)	The decision distance D_t in a decision grid is a constant that is determined by the number of decision trials t_i spent in the time series, where d_r and d_w represent numbers of right and wrong decisions, respectively.	$D_t = t_i = d_r + d_w$
41	Random series of unlimited trials (Theorem 11.6)	Random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under *unlimited trials*.	
42	Random series of limited trials (Theorem 11.7)	Random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under *limited trials*.	
43	Conditions of quality control systems (Theorem 11.8)	The *necessary conditions* for implementing a quality control system for a given product, service, or system are that all attributes of its quality can be: a) Abstractly identified b) Quantitatively defined, and c) Independently measurable.	

44	Predictability of new equilibrium (Theorem 12.2)	A *newly established equilibrium* on price P'_e is determined by the effect P' and feedback effect P'' of the driving forces deviating from the current equilibrium, and the increment of price caused by the shifting of equilibriums is as shown in the mathematical models, where ΔP may be positive or negative that represents a upward or downward shifting of the current equilibrium, respectively.	$P'_e = P'' + \frac{P'-P''}{2} = \frac{P' + P''}{2},\ P' > P'_e$ $\Delta P = P'_e - P_e = \frac{P' + P''}{2} - P_e,\ P' > P'_e$
45	Ultimate objective of software engineering (Theorem 12.4)	Automatic code generation is the only silver bullet to overcome the natural obstacles of the conservative software development productivity, to reduce software development costs, and to improve software quality as a result of reduced human involvement and uncertainty.	
46	Exponential Software Legacy Maintenance Costs (SLMC) (Theorem 12.5)	The ratio of maintenance cost C_m in a software development organization, r_m%, tends to exponentially increase over time t, and it is proportional to the total number of legacy systems N_L that the organization produced.	
47	Strength of motivations (Theorem 13.2)	A *motivation M* is proportional to both the strength of emotion $\|E_m\|$ and the difference between the expectancy of desire E and the current status S, of a person, and is inversely proportional to the cost to accomplish the expected motivation C, where $0 \le \|E_m\| \le 4$, $0 \le (E,S) \le 10$, and $1 \le C \le 10$.	$M = \frac{2.5 \bullet \| E_m \| \bullet (E\text{-}S)}{C}$
48	Mechanism of Software Maintenance Crisis (SMC) (Theorem 14.1)	A software development organization may face a situation known as the *software maintenance crisis*, in which the ratio of the maintenance costs r_m% is approaching 100% of the total costs that the organization spent.	
49	Rigorous levels of empirical and theoretical knowledge (Theorem 15.1)	An *empirical truth* is a truth based on or verifiable by observations, experiments, or experiences. In contrary, a *theoretical proposition* is an assertion based on formal theories or logical inferences.	

50	Necessary and sufficient conditions of IC (Theorem 15.3)	The conditions of IC, C_{IC}, are the possession of *event* B_e, *time* B_t, and *interrupt* B_{int} driven computational behaviors.	$C_{IC} = (B_e, B_t, B_{int})$
51	Necessary and sufficient conditions of AC (Theorem 15.3)	The conditions of AC, C_{AC}, are the possession of *goal* B_g and *inference* B_{inf} driven computational behaviors, in addition to the *event* B_e, *time* B_t, and *interrupt* B_i driven behaviors.	$C_{AC} = (B_g, B_{inf}, B_e, B_t, B_{int})$

It is interesting to compare and contrast the above set of formal principles and those of the heuristic ones as summarized in Table 14.6 and presented in Chapter 2. The comprehensive set of the 50 Wang's laws and 51 Wang's principles form the core of the theoretical framework of software engineering and the foundation towards a matured discipline. They are the crystallization of software engineering theories, which are exploratively elicited, carefully refined, and rigorously formalized from a vast set of empirical knowledge of software engineering and best software industrial practices.

14.5 Impact of the Theoretical Foundations to Software Engineering

The theoretical framework of software engineering developed throughout this book, as summarized in Section 14.4, provides a set of essential knowledge for excellent software engineers. This section introduces a set of cognitive principles of knowledge engineering. Based on them, the efforts and complexities for both knowledge and skill creation and acquisition are analyzed. Then, the expected impacts of the theoretical foundations of software engineering are discussed. A set of student feedback is reported that presents a fresh angle in perceiving the impacts of this book and related courses at both undergraduate and graduate levels.

14.5.1 THE COGNITIVE PRINCIPLES OF KNOWLEDGE ENGINEERING

This subsection creates a set of cognitive models of knowledge acquisitions and analyzes their complexities. This leads to the explanation why multidisciplinary knowledge is necessary and possible to be acquired by individuals. It is also helpful to explain why software engineering problems as a whole need to be investigated via the transdisciplinary approach.

14.5.1.1 The Effort Model of Knowledge Creation and Acquisition

According to the OAR model developed in Section 9.4.2, a knowledge is a relation between two or more abstract objects or concepts in long-term memory, while a behavior is a relation between a concept and an action in the action buffer memory. In other words, knowledge is meant what *to be*, while behaviors are meant what *to do*. Therefore, to some extent, Francis Bacon's assertion (1561-1626) that "*knowledge is power*" may be expressed more accurately as "*intelligence is power*" on the basis of Theorem 9.9, because it is intelligence rather than knowledge that transfers information and motivations into actions and behaviors.

It is amazing that human knowledge creation and development is so difficult where the solving of a hard problem always requires tremendous effort for years, decades, even centuries. However, once the knowledge is created, an ordinary effort may just be needed by individuals to understand and acquire it fairly quickly with no difficulty. This phenomenon in intelligence and knowledge science can be described more formally below.

Lemma 14.1 For a specifically new knowledge K, the *effort spent in its creation* $E_c(K)$ is much greater than that of its acquisition $E_a(K)$, i.e.:

$$E_c(K) >> E_a(K) \qquad (14.3)$$

Eq. 14.3 can be explained by the cognitive informatics theories and the OAR model developed in Section 9.4.2. It is recognized that the creation of knowledge is a process that establishes a novel relation between two or more objects or concepts by searching and evaluating a vast space of possibilities in order to explain a set of natural phenomena or abstract problems. Because the memory capacity of the brain can be as high as $10^{8,432}$ bits as estimated in Section 9.4.5, the complexity in searching for new knowledge is necessarily infinitive, if not a short cut should be discovered by chance or extensive and persistent thoughts. However, the acquisition of knowledge is to simply add a known relation in the LTM of an existing knowledge structure. This is why the speed for acquiring a given knowledge is relatively very high than that of knowledge creation.

It is noteworthy that the speed for acquiring skills could be much slower than that of knowledge acquisition, because of the need for hands on actions and the creation of a permanent internal behavioral model in the action buffer memory.

Lemma 14.2 The *effort of skill acquisition* $E_a(S)$ is much greater than that of knowledge acquisition $E_a(K)$, i.e.:

$$E_a(S) >> E_a(K) \tag{14.4}$$

Another angle for analyzing the effort of creative work in software engineering is by estimating the workload according to the coordinative work organization theory in Section 8.4. For instance, based on the statistical data in authoring this book, the workload can be estimated below with the time spent known as $T \approx 10{,}000$ hrs = 42.0 months, i.e.:

$$\begin{aligned} W_1 &= L \bullet T \\ &= 1 \bullet 42.0 = 42.0 \text{ [PM]} \end{aligned}$$

If this book were authored by multiple experts from 12 individual disciplines as covered in this book and resulting in the same degree of seamless coherency and consistency as that in this book, i.e., not an edited assembly of individual views toward software engineering, the effort according to Theorem 8.4, subject to $r = 30\%$, would be the following:

$$\begin{aligned} W &= W_1(1 + r \bullet \frac{L(L-1)}{2}) \\ &= 42.0\ (1 + 0.3 \bullet \frac{12.0(12.0-1)}{2}) \\ &= 873.6 \text{ [PM]} \end{aligned}$$

The result is about 72.8 person-years when the given interpersonal coordination rate $r = 30\%$, which indicates a mission virtually impossible! This example demonstrates that highly complicated problems may be feasibly and efficiently resolved by a single brain with enhanced and necessary multidisciplinary knowledge, rather than by a group of individuals. That is why a software engineering project should not involve too many architects in early phase no matter how complicated it is. This is also the experience of the author gained in the creation of this book on transdisciplinary foundations of software engineering.

14.5.1.2 The Complexity Model of Knowledge Creation

According to the relational complexity theory of systems as developed in Section 10.3.3, the domain and magnitude of experts' knowledge can be estimated based on the number of abstract objects or concepts and relations among them.

Definition 14.23 The potential *number of relations* among the combination of knowledge from *n* disciplines, $C_r(n)$, is in the order of n^2*,* i.e.:

$$\begin{aligned} C_r(n) &= O(n^2) \\ &= n \bullet (n\text{-}1) \end{aligned} \qquad (14.5)$$

where it is assumed that a pairwise relation *r* is asymmetric, that is, $r(a, b) \neq r(b, a)$, as given in Lemma 10.10.

Example 14.1 Recall that this book elicited and integrated fundamental theories of 12 disciplines. Assuming each discipline has 1,000 concepts in average, the entire knowledge or the total number of consumed concepts that an expert needs to cohesively acquire within all the 12 disciplines, $C_r(n)$, would turn up to be:

$$\begin{aligned} C_r(n) &= n \bullet (n\text{-}1) \\ &\approx 12 \bullet (12\text{-}1) \bullet 10^6 \\ &= 1.32 \times 10^8 \end{aligned} \qquad (14.6)$$

This figure shows that totally about 132 million new relations between multidisciplinary concepts need to be generated in the brain before the written of a book like this is possible.

In other words, the relational complexity of the multidisciplinary knowledge system (Eq. 10.6) is huge enough to enable new concepts, principles, and theories to be created that may not belong to any individual disciplines but on their edges. This reveals the advantages of an expert or a reader who possesses multidisciplinary knowledge toward a set of intricate problems under study, particularly in software engineering.

Constrained by the cognitive, organizational, and resources limitations and their complicated interrelations, most fundamental problems in software engineering theories are not a trivial one. Many of the problems have well been known in the very beginning of the emergence of software engineering 40 years ago; some of them may be traced back to more than 100 years in management science and even earlier in system philosophy. This is why Brooks classified these fundamental problems encountered in software engineering as the essential ones rather than the accidental ones [Brooks, 1975/95]. If only empirical studies are conducted on these problems, perhaps many additional decades are still needed to find some theoretical solutions for them, such as Theorem 8.7 – the 23rd Law of software engineering on the optimal labor allocation and the shortest duration in cooperative work, and Theorem 10.6 – the 34th Law of software engineering on the conservation of system gains.

14.5.1.3 The Cognitive Model of Knowledge Spaces of Multidisciplinary Knowledge

Most hard but interesting problems in research are on the edges of conventional disciplines. Therefore, transdisciplinary and multidisciplinary research are necessary. In addition, the maintaining of a global and holistic view is the key insight for fundamental research, which will be formally stated in Corollary 14.4.

Lemma 14.3 The impact of an expert with coherently *m* disciplinary knowledge $K_{\Sigma m}$ is much greater than those of *m* experts with separated individual disciplinary knowledge K_m, i.e.:

$$K_{\Sigma m} >> \sum_{i=1}^{m} K_m \qquad (14.7)$$

The above lemma can be proven on the basis of the OAR model for internal knowledge representation in the following theorem and related corollaries.

The 50th Law of Software Engineering

Theorem 14.2 The *power of multidisciplinary knowledge* states that the *ratio of knowledge space* Ω_Σ between the knowledge of an expert with coherently *m* disciplinary knowledge $K_{\Sigma m}$ and that of a group of *m* experts with separated individual disciplinary knowledge K_m is:

$$\begin{aligned}\Omega_\Sigma(m,n) &= \frac{K_{\Sigma m}}{K_m} \\ &= \frac{C^2_{m\bullet n}}{\sum\limits_{\mathrm{i=1}}^{\mathrm{m}} C^2_n} = \frac{\dfrac{(mn)!}{2!(mn\text{-}2)!}}{\dfrac{m(n)!}{2!(n\text{-}2)!}} \approx \frac{(mn)^2}{mn^2} \\ &= m\end{aligned} \tag{14.8}$$

where *n* is the number of *average knowledge objects* or concepts in the disciplines.

Example 14.2 Reusing the data and context as given in Example 14.1, i.e., *m* = 12 and *n* = 1,000, with Eq. 14.8, the knowledge space of an expert with coherently 12 disciplinary knowledge can be determined as: $\Omega_\Sigma(m,n)$ = Ω_Σ(12, 1,000) = 12, i.e., 12 times greater than that of the same of the 12 experts with separated individual disciplinary knowledge.

Corollary 14.2 The *ratio of knowledge space* Ω_1 between the knowledge of an expert with coherently *m* disciplinary knowledge $K_{\Sigma m}$ and that of a single expert with one of the individual disciplinary knowledge K_1 is:

$$\begin{aligned}\Omega_1(m,n) &= \frac{K_{\Sigma m}}{K_1} \\ &= \frac{C^2_{m\bullet n}}{C^2_n} = m^2\end{aligned} \tag{14.9}$$

where *n* is the number of average knowledge objects or concepts in a given discipline.

Example 14.3 Similar to the settings of Example 14.2, applying Corollary 14.2 the knowledge space of a single multidisciplinary expert with coherently 12 disciplinary knowledge is $\Omega_1(m, n) = \Omega_1(12, 1{,}000) = 12^2$, i.e., 144 times greater than that of one of the 12 experts with separated individual disciplinary knowledge.

Corollary 14.3 The more the interdisciplinary knowledge one acquires, the larger the knowledge space, and hence the higher the possibility for creation and innovation.

Corollary 14.3 explains the usage of the transdisciplinary approach towards the establishment of the theoretical framework of software engineering. The following corollary explains which is more important in research and software engineering if there is a need to choose one from broadness and depth of individuals' knowledge structures.

Corollary 14.4 In knowledge acquisition and knowledge engineering, *broadness* is more important than *depth*.

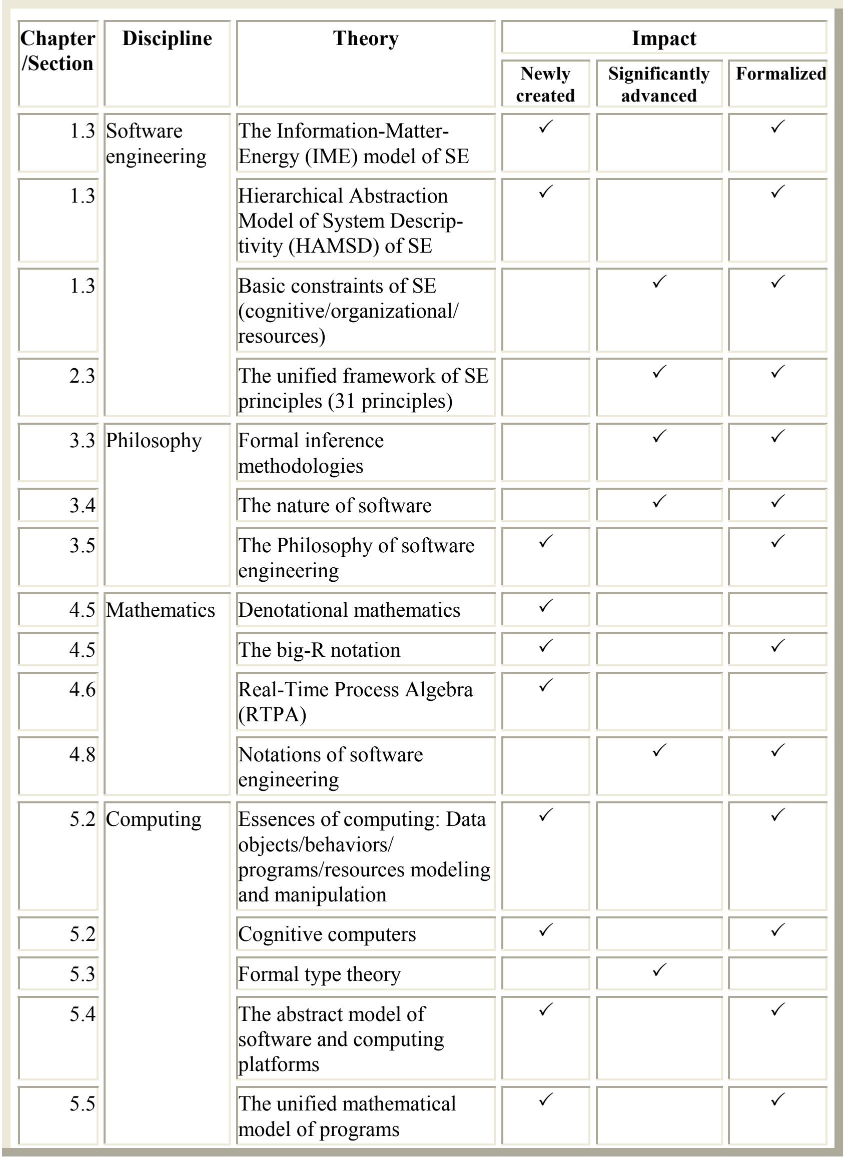
Corollary 14.4 is perfectly in line with the philosophy of holism as discussed in Section 3.2 on philosophies of engineering sciences.

14.5.2 EXPECTED IMPACTS OF WANG'S LAWS AND THEOREMS TO SOFTWARE ENGINEERING

Throughout the development of this book, it is observed that, because of its inherited complexity, wide applications, and both human and machine intelligent dependency, software engineering is not only a discipline that requires multidisciplinary knowledge, but is also an ideal testbed for evaluating existing theories and for developing new theories for the related disciplines.

The closely related disciplines are such as scientific philosophy, mathematics, computer science, engineering science, linguistics, information science, cognitive informatics, system science, management science, economics, sociology, and natural/machine intelligence. This book demonstrates that a wide range of new or enhanced theories, methodologies, and techniques may be developed for those disciplines during the systematical investigations of the theoretical and transdisciplinary foundations of software engineering as shown in Table 14.9. In table 14.9, the expected impacts of the theoretical foundations of software engineering are classified as theories newly created, significantly advanced, and/or formalized from empirical observations.

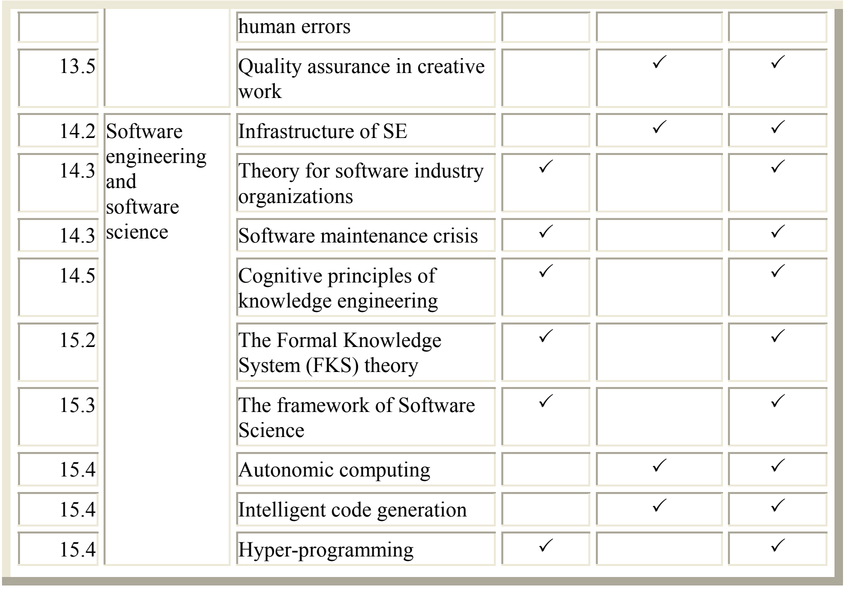
Table 14.9
The Impact of Software Engineering Theories on Related Disciplines



Chapter /Section	Discipline	Theory	Impact		
			Newly created	Significantly advanced	Formalized
1.3	Software engineering	The Information-Matter-Energy (IME) model of SE	✓		✓
1.3		Hierarchical Abstraction Model of System Descriptivity (HAMSD) of SE	✓		✓
1.3		Basic constraints of SE (cognitive/organizational/ resources)		✓	✓
2.3		The unified framework of SE principles (31 principles)		✓	✓
3.3	Philosophy	Formal inference methodologies		✓	✓
3.4		The nature of software		✓	✓
3.5		The Philosophy of software engineering	✓		✓
4.5	Mathematics	Denotational mathematics	✓		
4.5		The big-R notation	✓		✓
4.6		Real-Time Process Algebra (RTPA)	✓		
4.8		Notations of software engineering		✓	✓
5.2	Computing	Essences of computing: Data objects/behaviors/ programs/resources modeling and manipulation	✓		✓
5.2		Cognitive computers	✓		✓
5.3		Formal type theory		✓	
5.4		The abstract model of software and computing platforms	✓		✓
5.5		The unified mathematical model of programs	✓		✓

6.2	Linguistics	The formal model of the Generic English Grammar (GEG)	✓		✓
6.5		Deductive semantics of SE	✓		✓
6.6		Deductive semantics of RTPA	✓		✓
7.1	Information science	Bit: the common root of information science and computer science	✓		✓
7.3		Role of information to mankind evolution	✓		✓
7.4		Informatics laws of software	✓		✓
8.2	Generic engineering	The Engineering Objective Model (EOM)	✓		✓
8.2		The Engineering Maturity Model (EOM)	✓		✓
8.5		Coordinative Work Organization (CWO) theory	✓		✓
8.5		Laws of software engineering organization	✓		✓
8.5		Formal model of the mythical man-month		✓	✓
9.2	Cognitive informatics	Cognitive informatics	✓		✓
9.2		The Cognitive Model of Memory (CMM)	✓		✓
9.3		The Layered Reference Model of the Brain (LRMB)	✓		✓
9.3		The cognitive model of the brain	✓		✓
9.3		Equivalence between NI and AI			✓
9.4		The OAR model of internal knowledge representation	✓		✓
9.6		Cognitive complexity of software	✓		✓
10.3	System science	Mathematical models of abstract systems	✓		✓
10.3		System topology and magnitudes	✓		✓
10.3		System Organization Trees (SOTs)	✓		✓

10.3		System algebra	✓		
10.5		The formal framework of system principles		✓	✓
10.7		Software system complexity theory	✓		✓
11.2	Management Science	Formal principles of management science		✓	✓
11.3		Cognitive process of decision making	✓		✓
11.3		The decision grid (DG) theory	✓		✓
11.3		Formal game theory		✓	✓
11.4		Formal model of quality		✓	✓
11.5		The process infrastructure of SE		✓	✓
12.2	Economics	Fundamental laws of economics		✓	✓
12.2		Formal models of economic equilibrium		✓	✓
12.2		Mathematical model for the invisible hand (Adam Smith)		✓	✓
12.6		Mathematical models of software engineering costs		✓	✓
12.6		The Formal Economic Model of SE Costs (SEMSEC)	✓		✓
12.6		Optimization of SE economic decisions	✓		✓
12.6		The software legacy maintenance cost model	✓		✓
13.2	Sociology	Formalization of sociology principles		✓	✓
13.3		The motivation/attitude-driven behavioral model	✓		✓
13.4		The formal model of social organization		✓	✓
13.4		The formal organization tree (OT)	✓		✓
13.5		Coordinative work organization theory for large-scale SE projects	✓		✓
13.5		Mathematical model of	✓		✓



		human errors			
13.5		Quality assurance in creative work		✓	✓
14.2	Software engineering and software science	Infrastructure of SE		✓	✓
14.3		Theory for software industry organizations	✓		✓
14.3		Software maintenance crisis	✓		✓
14.5		Cognitive principles of knowledge engineering	✓		✓
15.2		The Formal Knowledge System (FKS) theory	✓		✓
15.3		The framework of Software Science	✓		✓
15.4		Autonomic computing		✓	✓
15.4		Intelligent code generation		✓	✓
15.4		Hyper-programming	✓		✓

Details of the newly developed theories have been reviewed in Section 14.4. Further discussions and explanations may be referred to related chapters and sections throughout this book.

14.5.3 STUDENTS' FEEDBACK

Graduate and undergraduate students majoring in software engineering at the University of Calgary have experienced the development of this book and its earlier versions in the form of lecture notes. The following feedback of graduate students from the graduate course on "Theoretical Foundations of Software Engineering" and "Empirical Foundations of Software Engineering" may reflect some of the influences of this work and its approach towards the rigorous treatment of software engineering foundations and theories, and the queries on fundamental laws underpinning software engineering organization and practice.

The following citations are student feedback from the above courses in their own words:

"I think the course provides an excellent understanding of the theoretical foundations of software engineering. The logical organization of the material and the well thought out presentations facilitated learning and made the large amount of information manageable. There are three main themes of learning that I feel I gained from this course:

"First, an understanding of the theoretical foundations of software engineering is invaluable to any new researcher. The opportunity to have a 'guided tour' of the literature and to have highlighted not only the areas of inquiry that fall under each category or study but also to see the 'state of the art' in terms of these fundamental areas was enlightening. It was particularly instructive to see how some of the core areas of software engineering are changing (e.g., computation) while other areas are spawning new avenues of inquiry (e.g., cognitive informatics).

"Second, an appreciation for the multidisciplinary nature of the problematic area was also demonstrated. While some of the fundamental areas expressed this theme better than others it was evident from the readings that the software engineering problematic is an abstract area of inquiry that could benefit from new perspectives from its traditional strongholds but that emerging areas like cognitive informatics need to be multidisciplinary in order to garner the full appreciation needed for software engineering. I find this theme particularly reassuring since I am in a related but separate field of study from software engineering but my research interests overlap many of the questions discussed in this course.

"Finally, the readings from the software engineering literature highlighted a crucial role of the researcher that I was somewhat surprised and particularly pleased to observe. That is, open critique of the field. I think one of the key functions of the academic community is to provide a forum for open debate and critique of practices that are detrimental to the field and of course to provide evidence to the ill effects of such practices and offer alternative approaches based upon solid research. The readings of the Turing award winners were surprisingly critical of many of the current practices but despite these critiques the authors shared an optimism that I feel is well deserved.

"If software engineering can continue to attract the caliber of researchers exemplified by the readings I have no doubt that many of the challenges highlighted in this course will be met. In addition, the opportunities that these challenges represent are extremely exciting for a new researcher like myself since they represent opportunities to make fundamental contributions to the field."

"In terms of impact I think that the most promise is held in the 'cognitive' stream since I think this area of research seems to hold the greatest opportunity in terms of articulating the relationship between energy-matter and information which serves as the basis for software engineering and many other areas of inquiry. This is certainly not meant to imply that similar impacts could not arise from the mathematical based approaches. In fact, I think that the findings in cognitive informatics approaches will drive the development of new forms of mathematical inquiry that better supports the needs of software engineering."

"From this course I learn to judge software standing on a relatively high level in critical thinking. Once I wondered how I put these theories into practice. After reading, thinking, and analyzing those lecture notes again and again, every time I have new findings. These theories indeed exist everywhere in the software industry. They propose the problems yet to be solved in this area; they extend software engineering to a larger scope; they expound essence of software in a philosophical way. Like building a house, whether it has a good foundation is very critical. This course set up a good foundation for my software engineering building. As a programmer, I would think about algorithm complexity and code complexity; as a designer, I would consider using specification language to describe system rigorously; as a project manager, I would specialize in coordination between systems, developers and users. I have acquired a clear sketch of software engineering in a top-down approach."

"In my particular case, as with any knowledge gain, I was once again reminded that "the more you know, the more you realize how little you know". The course showed me that some of the problems or issues the software industry faces are rooted deeper than I thought.

"I always blamed "bad" processes for most of the issues associated with software development. The information gained from the course allows me to see that some of the problems are a result of software being unable to follow the natural laws of the physical world. While I always knew that software development or software engineering activity was like no other discipline, I had no idea in how many ways it does not fit the traditional understanding of a given product's or entity's (our software system) relationship with the rest of the world."

"The classical papers provided an excellent appreciation for the discussions that have occurred and also pointed to emerging issues in the field. I think insights into the emerging aspects of software engineering were also introduced throughout as part of the current 'state-of-the-art' in each of the fundamental areas. I found this particularly useful as a new researcher and I suspect that those practicing software engineering also benefited from knowing the current state of affairs in the field in addition to the historical discussions. Interestingly, many of the fundamentals issues introduced by the classical authors still hold true today albeit in different forms and to varying degrees but it does point to the importance of addressing fundamental issues which solve a host of problems rather than simply addressing one problem that applies to only one situation under certain conditions.

"The first issue of interest was the strong theoretical base that is evidenced in the writings of each of these individuals. I find this somewhat surprising since even now a lot of the software engineering research is aimed at 'practical' problem solving and not more theoretical issues. I find this particularly interesting since while these individuals certainly eventually migrated towards academic careers they also had periods in industry, albeit often in research roles. This highlights a point often lost on many people not involved in research; that there is a need for solid research skills in both academia and industry. This sentiment was echoed in several of the papers in various forms such as a call for relevancy or for the need for closer ties by industry with the academic community. The classical papers also demonstrated one of the benefits of an emerging area of research, which is it attracts researchers from diverse backgrounds. These multiple perspectives and research backgrounds I think served the field well and the benefits of such approaches can be seen in more recent areas of inquiry like cognitive informatics which by design employs a multidisciplinary approach."

"The problems that we face and want to solve today can be traced to the foundations of software engineering. It is helpful for us to tackle the problems via revisiting the classical theories and practices in history. We can get good understanding about the nature of the current problems when we connect them to their headstreams in other disciplines in history and then try to find the solutions based on such understanding. From the point of view of history and tradition, it is possible for us to see some problems that have not appeared in our vision.

"With the perception of the history and tradition of software engineering, we can understand the ideas of pioneers and masters of the art more deeply. The development of theories, practices and technologies of software is no longer a set of broken fragments for us; they are now organic and vivid in our perception as a whole. Some mythical breakthroughs of technologies to us before are now logical and reasonable development of solutions for the original problems. Without this course, we could not have had such understanding and insight.

"One of the most important gains from the course is the methodology of research. Software engineering is not isolated from other disciplines, some of which have been well developed and some that are still not developed, but related tightly with other disciplines. The progress of other disciplines will potentially make contributions to software engineering and vice versa. This gives our researchers an opportunity to address some theoretical or practical problems of software engineering on the edge of investigation. The methodology of research in software engineering could be detailed in the following two steps:

(a) Each current problem is or can be found to have originated from some historical problems. Revisiting the classical theories related with the problem can help generate a different vision and understanding to tackle the problem.

(b) Nearly all the disciplines follow such a rule: the theories get developed when more and more theories and technologies that have already been developed in other disciplines are introduced into this area. So the hot spots of research are always on the edge.

"The knowledge acquired from the course is very useful as it gives the students some insight into the foundations of software engineering and broadens their outlook. The knowledge enables the students to understand the theory of computation and other inter-related issues in software engineering. The different views from leading scientists in the area of computer science made the course a very interesting one.

"Finally, I think the course highlights the wealth of opportunities that exist within software engineering to make fundamental contributions to research. What more could a researcher ask for!?"

14.6 Summary

The **theoretical framework of software engineering** developed in this book reveals that software engineering not only encompasses a wider domain of empirical applications, but also possesses much more theoretical essences that are closer to the root of human knowledge in terms of mathematics, philosophy, cognitive informatics, computation, economics, sociology, and system science.

A vast volume of empirical knowledge has been cumulated in software engineering in the last four decades that is yet to be theoretically processed and refined. The formal documentation of software engineering theories and the fundamental body of knowledge presented in this book are the first attempt to establish the formal and coherent knowledge framework of software engineering towards a matured discipline. The theoretical framework of software engineering presented in this book encompasses the fundamental principles and constraints of software engineering, theoretical foundations of software engineering, and transdisciplinary foundations of software engineering.

On the basis of the first three parts of this book on principles, constraints, theoretical foundations, and transdisciplinary foundations of software engineering, this chapter has put the focuses onto the entire infrastructure of software engineering and discussed the organization of the software industry. This chapter has also summarized the formalized body of knowledge towards software engineering. The impacts of the interdisciplinary foundations for software engineering have been discussed, and students' feedback on this book in the form of lecture notes has been reported.

ARCHITECTURAL SUMMARY OF KNOWLEDGE

Through this chapter, *Retrospect on Software Engineering*, readers have achieved the following strategic goals with the knowledge structure as summarized below.

Chapter 14. Retrospect on Software Engineering

■ Infrastructures of software engineering

- The process infrastructure of software engineering
- Process-based SE (PBSE)
  - The organizational model of PBSE
  - Software engineering process system establishment
  - Software engineering process system assessment
  - Software engineering process system improvement

■ Software industry organization

- The nature of the software industry
- Principles of software industry organization

- Basic principles of software industrial organization
- Separation of software designer, builders, quality assurors, and maintainers in software engineering
- Distributed time-shared development in software engineering

- A perspective on the software maintenance crisis
  - The mathematical model of software maintenance crisis
  - Reasons behind software maintenance crises
  - Solutions to software maintenance crisis

■ Essential knowledge towards excellent software engineers
- Basic constraints of software engineering
- Empirical principles of software engineering
- Laws of software engineering
- Formal principles of software engineering

■ Impact of the theoretical foundations to software engineering
- The cognitive principle of knowledge engineering
  - The effort model of knowledge creation and acquisition
  - The complexity model of knowledge creation
  - The cognitive model of knowledge spaces of multidisciplinary knowledge
- Expected impacts of Wang's laws and theorems to SE
- Students' feedback

SIGNIFICANT FINDINGS OF THIS CHAPTER

• **Process-Based Software Engineering** (PBSE) is an organizational methodology for software engineering, by which the infrastructure of software engineering, encompassing the three process subsystems of organization, development, and management, is integrated by a well-defined process reference model.

• The process reference model can be tailored or adapted to a specific project according to the nature of a project determining by the **project factors** such as application domain, scope, complexity, schedule, experience of project team, reuse opportunities identified, and/or resources availability.

• Tailoring of a PTPM from a comprehensive OPRM makes the software project leaders' tasks greatly simplified. Using this approach, project organization and conduct can be effectively performed within an organization's unified software engineering process infrastructure.

- The **organizational theories** and **methodologies** for the software industry, as important part of **software engineering in the large**, have been almost overlooked in this discipline. Towards a matured software engineering discipline, the nature of the software industry and the fundamental principles of software industrial organization need to be studied.

- The software market is a sector of the information processing market, where **standardization** and **human cognitive familiarity** play an important role in market share. Therefore, international or industrial standards, as well as intellectual properties, are important virtual assets in the software industry.

- Major current **strategic problems** in the software industry are identified as follows:

  a) **Referees are also players:** All the responsibilities in software design, implementation, and quality assurance are carried out by the same organization, even the same engineer or group. As a consequence of this confused allocation of responsibilities, when time, budget, or skills are limited, quality tends to be the first victim in a software engineering project under this form of organization.

  b) **Too high requirements and responsibility are put onto the shoulders of customers:** The fact that is often overlooked in software engineering is that customers may not be able to understand and evaluate the requirements, functionality, quality, reliability, and complete correctness of complex software systems. Therefore in software engineering it is unwise to rely on customers for a complete or thoughtful system requirements. It also unwise to let or to agree by any party that customers should ensure the sole responsibility for testing and evaluating a new software system.

- **Distributed Time-Shared Development (DTSD)** is a software engineering methodology that geographically allocates software development work broadly in distributed time zones with a wide-area Intranet.

- The mechanism of **Software Maintenance Crisis (SMC)** states that a software development organization may face a situation known as the *software maintenance crisis*, in which the ratio of the maintenance costs r_m% is approaching 100% of the total costs that the organization spent.

- The major **solutions** to deal with the SMC problems in software engineering and in the software industry are:

  a) **Enhance technologies** such as: i) to enhance software lifecycle processes to include software maintenance and retirement; ii) to increase depreciation of software systems; iii) To adopt a public agent acting like a library to store all code and documents of commercial software systems.

b) **Software industry reorganization** such as: iv) To create a new affiliated service industry to maintain the legacy systems as that of garages for the automobile industry; and v) To establish software insurance agencies who take responsibility for supporting any interrupted service of vendors.

c) **Honor the responsibility** and liability of the vendor.

- The **power of multidisciplinary knowledge** states that the *ratio of knowledge space* Ω_Σ between the knowledge of an expert with coherently *m* disciplinary knowledge K_Σ and that of a group of *m* experts with separated individual disciplinary knowledge K_m is $\Omega_\Sigma(m,n) = \frac{K_\Sigma}{K_m} \approx m$, where *n* is the number of *average knowledge objects* or concepts in the disciplines.

- The **ratio of knowledge space** Ω_1 between the knowledge of an expert with coherently *m* disciplinary knowledge K_Σ and that of an expert with individual disciplinary knowledge K_1 is $\Omega_1(m,n) = \frac{K_\Sigma}{K_1} \approx m^2$.

- The more the interdisciplinary knowledge one acquires, the larger the knowledge space, and hence the higher the possibility for **creation and innovation** in software engineering.

- In knowledge acquisition and **knowledge engineering**, *broadness* is more important than *depth*.

- Through out the development of this book, it is observed that, because of its inherited complexity, wide applications, and both human and machine intelligence dependency, **software engineering** is not only a discipline that requires multidisciplinary knowledge, but is also an ideal testbed for evaluating existing theories and for developing new theories for the related disciplines.

FUNDAMENTAL THEORIES DEVELOPED IN THIS CHAPTER

Infrastructure of software engineering

- As the scale of software increases continually at an ever faster rate, greater complexity and professional practices become critical, which requires the studies on the infrastructure of software engineering in the form of **process-based software engineering** (PBSE).

- A **software engineering process** is a set of sequential practices that is functionally coherent and reusable for software engineering *organization, development,* and *management*. It is usually referred to as the *software process*, or simply the process.

- A **process reference model** is an established, validated, and proven software engineering process model that consists of a comprehensive set of software processes and reflects the benchmarked best practices in the software industry.

- **The organization model of PBSE:** The common practices in organizing a software engineering process system are given in Fig. 14.3.
  - At the entire **enterprise level**, a common *organization's process reference model* (OPRM) is established.
  - At **project level**, a number of parallel *development* and *management processes* may exist based on the individual project's tailored process model (PTPM), which are derived models of the OPRM reference model. The OPRM process reference model is the key for empirical PBSE. If an OPRM is well established in an organization, the PTPMs at project level can easily be derived. For a PTPM of an individual project, the management and development processes should be one-to-one designed and synchronized.

- **Software Engineering Process System Establishment:** An initial and fundamental step in PBSE is process system establishment. The major aim of process establishment is to build up a software engineering process reference model for a software development organization. When a process system is established and experienced, improvement can be initiated effectively via process assessment and benchmarking.
  - The three basic steps for **deriving a software project process model** are: a) Select and reuse a process system reference model at organization; b) Derive a process model at project level; and c) Apply the derived project process model.

- **Software Engineering Process System Assessment:** From the viewpoint of reference systems there are four types of assessment methods: the *model-based, standard-based, benchmark-based,* and *integrated (model-and-benchmark-based) assessment*.

- **Software Engineering Process System Improvement: P**rocess system improvement is the goal of process assessment, acting on issues found in an assessment and enhancing the effectiveness of processes in the process system. Key categories of process improvement methodologies are *goal-oriented process improvement, benchmark-based process improvement,* and *continuous process improvement.*

Software industry organization

- The **overall organizational methodology** for the software industry is PBSE. Based on a generic software engineering process model such as SEPRM, software engineering activities and processes at personal, project (team), and enterprise levels can be well organized in the three essential aspects of organization, development, and management.

- The **key organization principles** for the software industry are: a) To improve *productivity*, b) To practice *specialization* or division of labor; and c) To deal with the *labor-time interlock constraint.*

- In order to solve the inherited problems, a **separation of roles in the software industry** is necessary. That is, the software industry is ideally split into four sectors known as the organizations of *software **designers***, *software **builders***, *software **quality assurors***, and *software **maintainers*** with totally separated and explicitly designated roles and responsibilities.

- **Distributed Time-Shared Development (DTSD)** is a new approach of division of labor in the time-dimension contrary to division of labor in the functional or specialization dimension. This methodology takes advantages of geographically allocated project teams distributed in different time zones, but interconnected through a wide-area Intranet and supported by remote execution capabilities. Well organized and synchronized DTSD projects may gain time greatly in development, because DTSD provides a virtual 24-hour software development organization with the teams deployed in two or three countries globally.

- **Software Maintenance Crisis (SMC)** is a phenomenon that happens when the demand for software maintenance exceeds the capability that a software development organization can provide, or when the costs of legacy software maintenance predominantly override the investment for new software development.

- There is a need of a sector in the software industry known as the professional software legacy maintainers or the **software garages**.

Essential knowledge towards excellent software engineers

- Throughout this book, the **theoretical and empirical foundations of software engineering** have been explored in a rigorous and transdisciplinary approach. The principles of software engineering are formally documented as a comprehensive set of theorems and laws. This section summarizes the theoretical framework of software engineering principles and laws, which form the fundamental, durable, and enlightening knowledge for researchers and practitioners in software engineering.

  - The essential knowledge on the **14 basic constraints** of software engineering on cognition, organization, and resources is summarized in Table 14.5.

  - The essential knowledge on the **31 empirical principles** of software engineering is summarized in Table 14.6.

  - The essential knowledge on the **50 laws of software engineering** is summarized in Table 14.7.

  - The essential knowledge on **51 formal principles of software engineering** is summarized in Table 14.8.

Impact of the theoretical foundations of software engineering

- A formal and rational documentation of a **comprehensive and essential body of software engineering knowledge** with rigorous treatments.

- **Principles of knowledge engineering:** The relationship between knowledge creation and acquisition is as follows:

  - For a specifically new knowledge K, the **effort spent in its creation** $E_c(K)$ is far more than that of its **acquisition** $E_a(K)$, i.e., $E_c(K) >> E_a(K)$.

  - The **effort of skill acquisition** $E_a(S)$ is far more than that of **knowledge acquisition** $E_a(K)$, i.e., $E_a(S) >> E_a(K)$.

- This book demonstrates that a wide range of **new or enhanced theories, methodologies, and techniques** have been developed not only for software engineering, but also for the closely related disciplines such as scientific philosophy, mathematics, computer science, engineering science, linguistics, information science, cognitive informatics, system science, management science, economics, sociology, and natural/machine intelligence.

Questions and Research Opportunities

14.1 What are the problems in software industry organization? How may the software industry be systematically organized?

14.2 What is Process-Based Software Engineering (PBSE)? Why may process techniques be adopted as the infrastructure of software engineering?

14.3 What are the basic organization principles for the software industry?

14.4 Why is there a need to separate software designers, builders, quality assurors, and maintainers in software engineering? What are the main responsibilities of each of the four sectors?

14.5 What is the Distributed Time-Shared Development (DTSD) technology in software engineering? What is the impact of DTSD on the software industrial organization?

14.6 How is software maintenance crisis discovered and modeled?

14.7 What are the main reasons behind software maintenance crises and potential solutions?

14.8 What are the roles of the special sector in the software industry known as the software legacy maintainers?

14.9 There is an argument that programming has no scientific foundations because both professionals and amateurs can write programs. Do you agree with this observation? Why?

14.10 Why did Brooks consider there is no silver bullet for software development in the 1970s? Are those claims still true?

14.11 Why has software engineering been considered as silver bullet in other disciplines where large-scale software systems are needed?

14.12 Identify some useful theories and techniques provided in this book that would be the potential silver bullets for software engineering.

14.13 Consider what would be the potential silver bullet for software engineering after learning the transdisciplinary theories and methodologies for software engineering presented in this book.

14.14 Summarize and describe how many metaphors of software and software engineering have been explored in this book, and provide a description for each of them.

14.15 It's believed that automatic code generation technologies may replace programmers in the future. Towards archiving this objective in software engineering, what are the potential impacts on the design means (denotational mathematics) and the implementation tools (compilers, programming environments, and testing systems)?

14.16 Software engineering is dependent on multidisciplinary foundations. Summarize the closely related disciplines to software engineering as described in this book.

14.17 Which laws of software engineering are you most interested in? why?

14.18 Which formal principles of software engineering are you most interested in? why?

14.19 According to Theorem 14.2 and Corollary 14.2, explain why the knowledge space of multidisciplinary experts may be m^2 times greater than that of an expert with individual disciplinary knowledge.

14.20 Why is *broadness* more important than *depth* (Corollary 14.3) in knowledge acquisition and knowledge engineering?

14.21 May software engineering methodologies and approaches be exported and applied to other engineering disciplines? Can you provide an example?

14.22 Summarize the structure of knowledge on the transdisciplinary theoretical foundations of software engineering presented in this book in a hierarchical diagram, and describe their interrelationship and possible impacts on software engineering.

14.23 What are the common characteristics of the pioneers in software engineering as reviewed in the classic articles listed in the last question of each chapter?

14.24 Read the following classic article in software engineering:

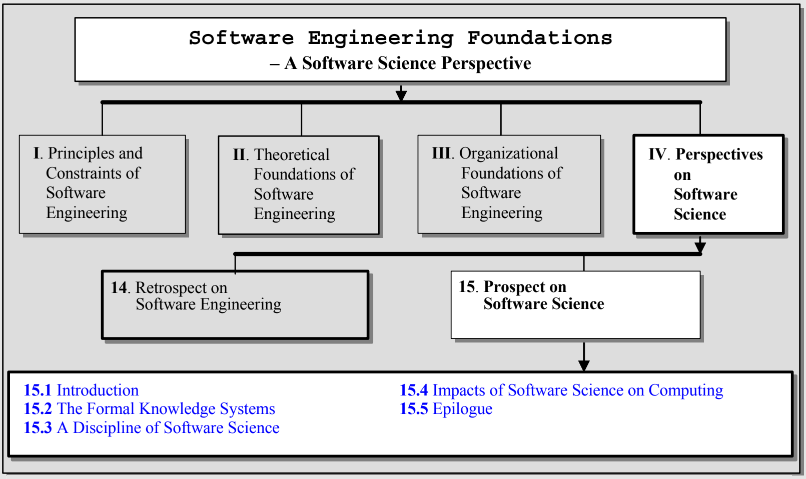
> John Backus (1978), Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, The 1977 Turing Award Lecture, *Communications of the ACM*, 21(8), pp. 613-641.

Discuss the following topics in a group or individually:

- About the author.
- What are the von Neumann architecture and programming styles based on it?
- What were the functional and algebraic styles proposed by the author?
- What conclusions of the article interested you? Why?
- Your arguments or counter-points on any of the conclusions derived in this article.

Chapter 15

PROSPECT ON SOFTWARE SCIENCE


Software Engineering Foundations
– A Software Science Perspective
I. Principles and Constraints of Software Engineering
II. Theoretical Foundations of Software Engineering
III. Organizational Foundations of Software Engineering
IV. Perspectives on Software Science
14. Retrospect on Software Engineering
15. Prospect on Software Science

15. Prospect on Software Science

Knowledge Structure

❍ The formal knowledge systems

- The framework of formal knowledge
- The roles of theoretical and empirical knowledge

❍ A discipline of software science

- Software science: software engineering in the 21st century
- Architecture of software science
- Denotational mathematics for software science

❍ Impact of software science on computing

- Autonomic computing
- Intelligent code generation
- Hyper-programming: New Faces of the Software Architectural Framework

❍ Epilogue

Learning Objectives

- To understand the essences of the formal knowledge systems and its applications in software engineering.

- To know the roles of theoretical and empirical knowledge in software engineering.

- To recognize the emergence of the discipline of software science, and its relationship with software engineering.

- To understand the theoretical structure of software science and the underpinning denotational mathematics.

- To be aware of trends in future developments between the interactions of software science, software engineering, and computing.

"If I have seen farther, it is because I have stood on the shoulders of giants."

Isaac Newton (1643 - 1727)

"Things that were valuable a decade ago will be valuable decades from now. The field is moving too fast to chase them. "

Davis L. Parnas (1997)

15.1 Introduction

Throughout this book, software has been recognized as an entire range of very widely and frequently used objects and phenomena in human knowledge. As a logical consequence, the results of this book provide a foundation leading to the emergence of software science complementing to software engineering. The former is the theoretical inquiry of software and the laws constrain it; while the latter is the empirical study of engineering methodologies and techniques for software development and software engineering organization.

The relationship between software science and software engineering can be analogized to those of theoretical physics and applied physics, or dynamics and mechanical engineering. Without theoretical physics there would be no matured applied physics; without dynamics there would be no matured mechanical engineering. So is software science with software engineering. The ultimate purpose of this book is an attempt to demonstrate that almost all the fundamental problems which could not be solved in the last four decades in software engineering simply stemmed from the lack of coherent theories in the form of software science. The vast cumulated empirical knowledge and industrial practice in software engineering have made this possible to enable the emergence of software science.

Another aim of this concluding chapter is to demonstrate that researchers and practitioners are enabled to rationally predict the future trends in software engineering based on the theoretical foundations about it, the empirical observations on it, and the transdisciplinary knowledge gained from more matured disciplines. Therefore, to a certain extent, the theoretical and empirical theories developed in this book provide a predictability for the future developments and a foundation for explaining unknowns in software engineering.

In the remainder of this chapter, the perspectives on software science and engineering will be presented in three sections. Section 15.2 describes the formal structure of generic knowledge systems, which provides a blueprint for organizing the theoretical framework of software engineering knowledge. Section 15.3 introduces the emerging discipline of theoretical software science as a natural extension of empirical software engineering. Then, Section 15.4 discusses potential impacts of software science on computing and applied software engineering, in the facets of autonomic computing, intelligent software code generation, and hyper-programming.

15.2 The Formal Knowledge System

The entire human knowledge can be classified as either empirical or formal knowledge. The former is the direct knowledge about the physical world, while the latter is the derived knowledge about both the physical and abstract worlds. A formal knowledge system is needed to maintain a stable, efficient, and rigorous inference base, in which only true or false conclusions may be derived and there is no gray area in between the conclusions of a rigorous inference.

This section presents a framework of the formal knowledge system. The taxonomy of human knowledge as a formal system is reviewed. The roles of formal and empirical knowledge are contrasted.

15.2.1 THE FRAMEWORK OF FORMAL KNOWLEDGE

Mathematical thoughts provide a successful paradigm to organize and validate human knowledge, where once a truth or a theorem is established, it is true till the axioms or conditions that it stands for are changed or extended. A proven truth or theorem in mathematics does not need to be argued each time when one applies it as a basis of reasoning. This is the advantage and efficiency of formal knowledge in science and engineering. In other words, if any theory or conclusion may be argued from time-to-time based on a seemed wiser idea or a trade-off, it is an empirical result rather than a formal theory.

The framework of Formal Knowledge System (FKS) of mankind [Wang, 2007a] can be described as shown in Fig. 15.1. The FKS framework shows the interrelationships between a comprehensive set of terms of formal knowledge, where the taxonomy of formal knowledge and their definitions are presented in Table 15.1.

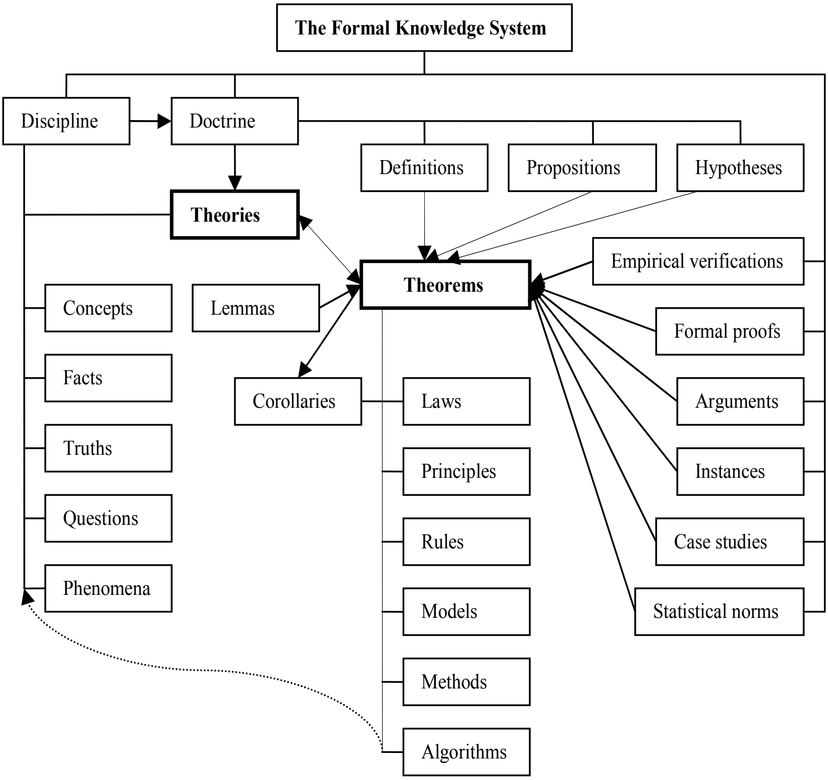
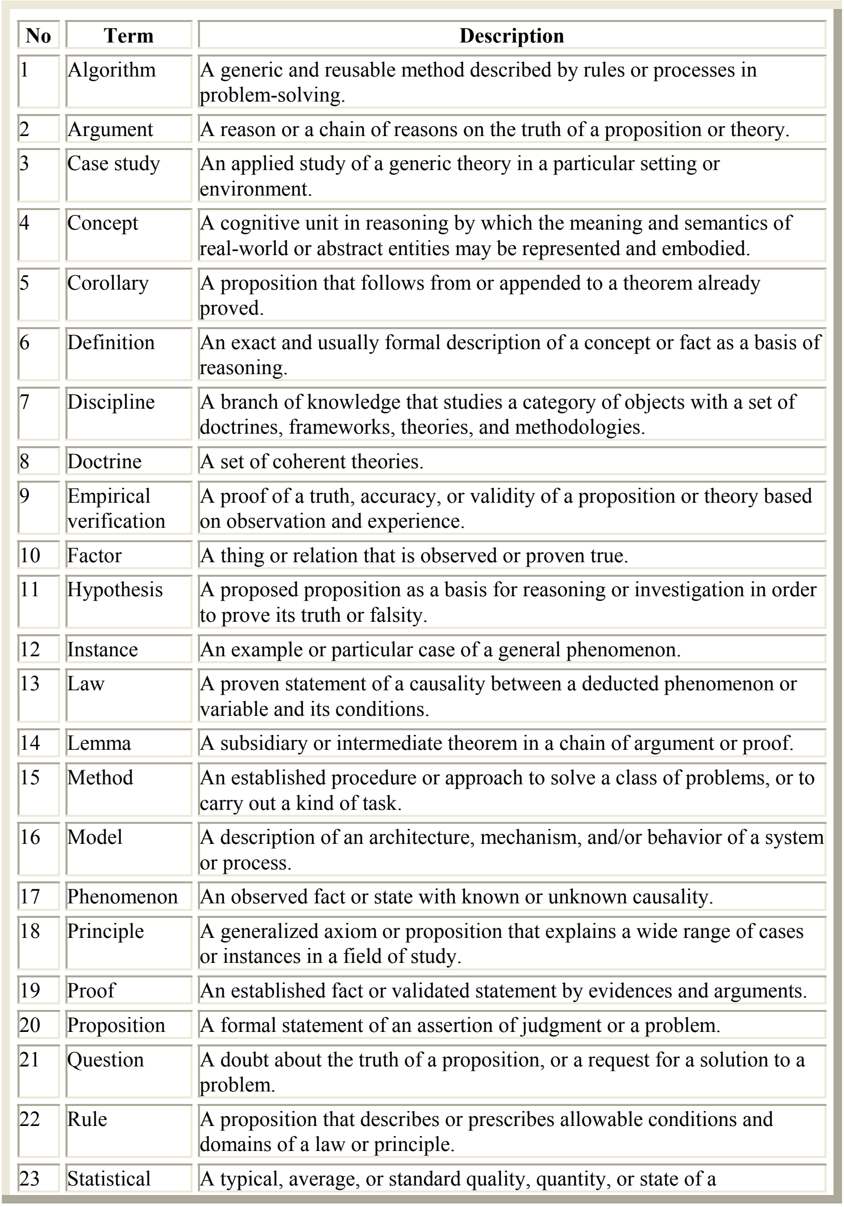


Figure 15.1 The framework of Formal Knowledge System (FKS)

The FKS system is centered by a set of theories. A *theory* is a statement of how and why certain objects, facts, or truths are related. An *empirical truth* is a truth based on or verifiable by observation, experiment, or experience. A *theoretical proposition* is an assertion based on formal theories or logical reasoning, which is a formalization of generic truth and proven empirical knowledge. According to Lemmas 14.1 and 14.2, theoretical knowledge may be easier to acquire when it is existed and proven. However, empirical knowledge is very difficult to be gained without hands-on practice.

According to the FFK model, an immature discipline of science and engineering is characterized as that its body of knowledge is not formalized or is mainly empirical. When there is no theory in a field of human enquiry, the practice in it is risk-prone. Instead of a set of proven theories, the immature disciplines usually document a large set of observed facts, phenomena, and their possible or partially working explanations. In such disciplines, researchers and practitioners might be able to argue every informal conclusion documented in natural languages from time-to-time probably for hundreds of years, until it is formalized and proven rigorously.

Table 15.1
Taxonomy of Formal Knowledge



No	Term	Description
1	Algorithm	A generic and reusable method described by rules or processes in problem-solving.
2	Argument	A reason or a chain of reasons on the truth of a proposition or theory.
3	Case study	An applied study of a generic theory in a particular setting or environment.
4	Concept	A cognitive unit in reasoning by which the meaning and semantics of real-world or abstract entities may be represented and embodied.
5	Corollary	A proposition that follows from or appended to a theorem already proved.
6	Definition	An exact and usually formal description of a concept or fact as a basis of reasoning.
7	Discipline	A branch of knowledge that studies a category of objects with a set of doctrines, frameworks, theories, and methodologies.
8	Doctrine	A set of coherent theories.
9	Empirical verification	A proof of a truth, accuracy, or validity of a proposition or theory based on observation and experience.
10	Factor	A thing or relation that is observed or proven true.
11	Hypothesis	A proposed proposition as a basis for reasoning or investigation in order to prove its truth or falsity.
12	Instance	An example or particular case of a general phenomenon.
13	Law	A proven statement of a causality between a deducted phenomenon or variable and its conditions.
14	Lemma	A subsidiary or intermediate theorem in a chain of argument or proof.
15	Method	An established procedure or approach to solve a class of problems, or to carry out a kind of task.
16	Model	A description of an architecture, mechanism, and/or behavior of a system or process.
17	Phenomenon	An observed fact or state with known or unknown causality.
18	Principle	A generalized axiom or proposition that explains a wide range of cases or instances in a field of study.
19	Proof	An established fact or validated statement by evidences and arguments.
20	Proposition	A formal statement of an assertion of judgment or a problem.
21	Question	A doubt about the truth of a proposition, or a request for a solution to a problem.
22	Rule	A proposition that describes or prescribes allowable conditions and domains of a law or principle.
23	Statistical	A typical, average, or standard quality, quantity, or state of a

	norm	phenomenon or system based on a large set of observations and statistic analyses.
24	Theorem	A generic proposition expressed formally and established by means of accepted truths.
25	Theory	A system of generic and formalized principles, theorems, laws, relations, and models independent of objects to be explained or practices are to be based.
26	Truth	An established, proven, or accepted constant state of a term or proposition in reasoning that is either true or false.

Software engineering is such an immature discipline in which a huge volume of empirical knowledge has been documented and nobody can prove whether this kind of knowledge is universally true or not; what their applicable axioms, conditions, and contexts are; and if they are coherent and complete. The formal documentation of software engineering theories and fundamental body of knowledge in this book is the first attempt to establish a formal theoretical framework of software engineering towards a matured discipline.

The disciplines of mathematics and physics are successful paradigms that adopt the formal knowledge system. The advantages of FKS are its *stability* and *efficiency*. The former is a property of formal knowledge that once it is established and proven, users who refer to it will no longer need to reexamine or reprove it. The latter is a property of formal knowledge that is exclusively true or false that saves everybody's time to argue a proven theory.

15.2.2 THE ROLES OF FORMAL AND EMPIRICAL KNOWLEDGE

In contrasting the nature of empirical knowledge and theoretical knowledge, the following principle on software engineering knowledge can be derived below.

The 49th Principle of Software Engineering

Theorem 15.1 The *rigorous levels of empirical and theoretical knowledge* states that an *empirical truth* is a truth based on or verifiable by observations, experiments, or experiences. In contrary, a *theoretical proposition* is an assertion based on formal theories, logical, or mathematical inferences.

Based on Theorem 15.1, a corollary on application domains of theoretical and empirical knowledge is stated as follows.

Corollary 15.1 The validation scope of *theoretical knowledge* is universal in its domain such as $\forall x \in S \Rightarrow p(x)$; while the validation scope of *empirical knowledge* is based on limited observations such as $\exists x \in S \Rightarrow p(x)$, where S is the domain of a problem x under study, and p a proven proposition or derived theory on x.

The differences of the validated domains between theoretical and empirical knowledge indicates the levels of refinements of different forms of knowledge and their reliability.

Empirical knowledge answers *how*; while theoretical knowledge reveals *why*. Theoretical knowledge is a formalization of generic truth and proven empirical knowledge. Although the discovery and development of a theory or a law may take decades even centuries, its acquisition and exchange are much easier and faster with ordinary effort. However, empirical knowledge is very difficult to be gained. One may consider that a person can learn multiple scientific disciplines such as the fields covered in this book, but few think that a person may be an expert in multiple engineering disciplines such as in all areas of electrical, mechanical, chemical, and computer engineering. The reasons behind this are that each engineering area requires specific empirical knowledge, skills, and tools. All of them need a long period of training and practice to be an expert.

Huge empirical knowledge were created and disappeared over time. For example, there are tons of empirical knowledge on software engineering published each year in the last decades. However, those that would be included in a textbook on software engineering theories as proven and general truth, rather than specific cases partially working on certain given or nonspecified constraints, would be no more than a few handful pages.

According to Corollary 15.1, the major risk of empirical knowledge is its uncertainty when applying in a different environment, even the same environment but at different time, because empirical knowledge and common sense are often error-prone. Consider the following examples:

- In early age of human civilization, people commonly believed that the earth is flat and mankind lived in the center of the universe, until Nicholas Copernicus (1473-1543) proven that these common senses were false in the early 16th century.
- Managers believed that the larger the project, the larger the team required. However, Theorem 8.7 (Law 23 of software

engineering) and Theorem 13.4 (Law 48 of software engineering) reveal that for a given workload, the optimal labor allocation and the shortest project period are constrained by natural laws. That is, putting more than the optimal number of persons into a group is not only counterproductive, but also dramatically increasing the real workload of the project – a major hidden reason of most failures in software engineering project organization.

According to Theorem 3.1, all recurrent objects in nature and their relations are constrained by certain invariant laws, no matter one observed them or not at a given time. This is one of the essences that may be gained in this book. In software engineering, we were told to not use *goto* statement. According to Theorem 4.7, jump or *goto* statement is one of the 17 essential computational operations in process algebra for describing software system behaviors. We were told *extreme* or pairwise *programming* is the latest solution to software engineering. However, according to Theorem 8.11, pairwise working is only efficient and suitable for a certain small scope of software projects because in this approach the interpersonal coordination rate *r* is too high, which may result in a huge additional real workload for a large-scale software project that may be risk prone.

15.3 A Discipline of Software Science

It is recognized that *theoretical software engineering* focuses on foundations and basic theories of software engineering; whilst *empirical software engineering* concentrates on heuristic principles, tools/environments, and best practices by *case studies*, *experiments*, *trials*, and *benchmarking*. Throughout this book it is noteworthy that, because software is the most abstract instructive information, software engineering is one of the most complicated branches of engineering, which requires intensive theoretical investigations rather than only empirical studies. The widely impacted and applicable objects and the complicated theories in software engineering lead to the emergence of a scientific discipline known as software science.

This section provides perspectives on the emerging discipline of software science along with the maturity of software engineering theories and methodologies in fundamental research as presented throughout this book. The architecture and roadmap of software science will be presented.

The theoretical framework, mathematical foundations, and basic methodologies of software science will be briefly introduced.

15.3.1 SOFTWARE SCIENCE: SOFTWARE ENGINEERING IN THE 21ST CENTURY

In the history of science and engineering, a matured discipline always gave birth to new disciplines. This generic evolutionary tendency has been formally described in the EMM model in Theorem 8.3. For instance, theoretical physics was emerged from general and applied physics, and theoretical computing was emerged from computer engineering. So will software science emerge and grow in the field of software, computer, information, knowledge, and intelligent engineering [Wang, 2007a].

Definition 15.1 *Software science* is a discipline of human enquiry that studies the theoretical framework of software as instructive and behavioral information, which can be embodied and executed by generic computers in order to create expected system behaviors and machine intelligence.

The discipline of software science enquiries the common objects in the abstract world such as software, information, data, knowledge, instruction, executable behavior, and their processing by natural and machine intelligence. In other words, software science studies instructive and behavioral information and the mechanism of its translation into system behaviors.

The relationship between software science (SS) and software engineering (SE) can be analogized with those of theoretical physics (TP) and applied physics (AP) as follows:

$$\text{SS} : \text{SE} = \text{TP} : \text{AP} \qquad (15.1)$$

Software science is the theoretical inquiry of software as an entire range of very widely and frequently used objects and phenomena in human knowledge; while software engineering is the empirical study of engineering methodologies and techniques for software development and software industry organization applying theories of software science. Without theoretical physics there would be no matured applied physics; without dynamics there would be no matured mechanical engineering. So do software science with software engineering.

Based on Definition 15.1 and Eq. 15.1, software engineering may be perceived as applied software science. Therefore, the intension of software engineering as provided in Definition 1.6 can be refined as follows.

Definition 15.2 *Software engineering* is an engineering discipline that applies software science theories and methodologies to efficiently, economically, and reliably organize and develop large-scale software systems.

This book has revealed that almost all the fundamental problems that could not be solved in the last four decades in software engineering were simply stemmed from the lack of software science.

It is noteworthy that cognitive informatics perceives information as anything that can be inputted into and processed by the brain; while software science perceives software as any instructive information that can be executed and transformed into computational behaviors by computers. This forges a relationship between cognitive informatics and software science, which indicates that the former is the foundation for natural intelligence science, and the latter is the foundation for artificial intelligence science and software engineering.

15.3.2 ARCHITECTURE OF SOFTWARE SCIENCE

The architecture of software science can be classified into four categories namely theories and methodologies, denotational mathematics, cognitive informatics, and organizational theories as shown in Fig. 15.2.

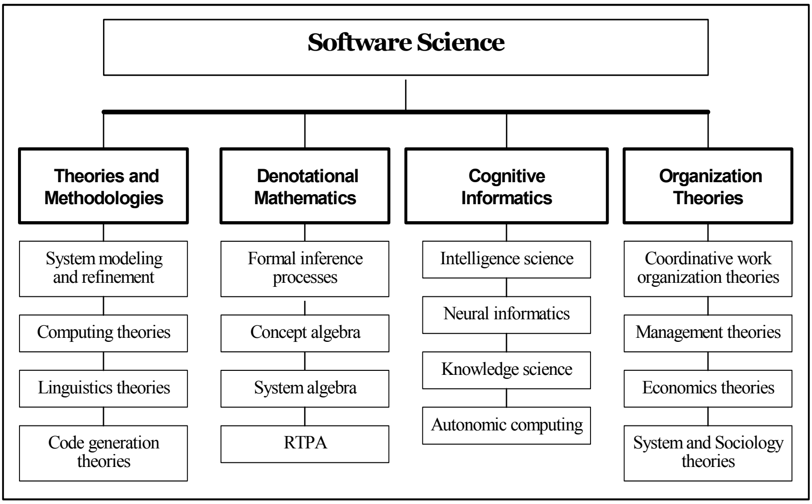


Figure 15.2 The architecture of software science

Theories and methodologies of software science encompass system modeling and refinement methodologies, computing theories, formal linguistic theories, and software code generation theories. Both imperative and autonomic computing theories as well as their engineering applications are explored in this category.

Denotational mathematics for software science is the enquiry for its mathematical foundations in the forms of formal inference methodologies, concept algebra, system algebra, and RTPA. In the contemporary mathematics for software science and software engineering, concept algebra is designed to deal with the *to be* problems and knowledge manipulation. System algebra is developed to formally treat the *to have* problems in terms of dynamic relations and possessions beyond set theory. RTPA is adopted to formalize the *to do* problems such as system architectures, static and dynamic behaviors. Further discussion on denotational mathematics for software science will be presented in Section 15.3.3.

Cognitive informatics for software science encompasses intelligence science, neural informatics, knowledge science, and autonomic computing. Cognitive informatics helps to understand and explain the fundamental mechanisms of natural intelligence and its products in terms of information and knowledge. It also studies the software implementation of intelligent behaviors by autonomic computing. Advances in cognitive informatics will help to overcome the cognitive barriers and inherited complicities in software engineering, which is called the *intellectually manageability* by Dijkstra and the *essential difficulties* by Brooks in software engineering. The entire structure of cognitive informatics may be referred to Chapter 9, and the theory of autonomic computing will be extended in Section 15.4.1.

Organizational theories of software science encompass coordinative work organization theories, management theories, economics theories, and system/sociology theories. The organizational facet of software science studies how large-scale software engineering projects may be optimally organized and what the underpinning laws are at different levels of complexities. The theoretical framework of software engineering organization has been outlined in Chapters 8 through 13 in this book.

15.3.3 DENOTATIONAL MATHEMATICS FOR SOFTWARE SCIENCE

It is recognized that many branches of mathematics were emerged in engineering sciences in order to meet their abstract, rigorous, and expressive needs. These phenomena may be conceived as that new problems require new forms of mathematics. Also, the history of sciences and engineering shows that the maturity of a new discipline mainly characterized by the maturity of its mathematical means that enables rigorous modeling and reasoning in the discipline. Conventional *analytic mathematics* are unable to solve the fundamental problems inherited in software science/software engineering, cognitive informatics, intelligence science, and knowledge science. Therefore, contemporary *denotational mathematical structures* and *means* beyond mathematical logic are yet to be sought.

As identified in Table 6.3, the descriptivity of humans and systems behaviors may be classified into three basic categories known as to *be*, to *have*, and to *do*. All mathematical means and forms, in general, are an abstract and formal description of these three categories of denotational needs and their rules as shown in Table 15.2.

Table 15.2
Denotational Mathematical Means for Software Science

Function	**Category**	**Mathematical Means**	
		Conventional	**Denotational**
Identify *objects* and *attributes*	To *be* (|=)	Logic	Concept algebra
Describe *relations* and *possession*	To *have* (|⊂)	Set theory	System algebra
Describe *status* and *behaviors*	To *do* (|>)	Functions	RTPA

In Table 15.2, the conventional and denotational mathematical means are contrasted. According to Table 15.2, the *generic usage of mathematics* is the means and rules to rigorously and generically express thought and notions at a higher-level of abstraction and rigor.

15.3.3.1 Concept Algebra

A *concept* is a cognitive unit [Ganter and Wille, 1999; Quillian, 1968; Wang, 2006e] by which the meanings and semantics of a real-world entity or an abstract entity may be represented and embodied based on the OAR model [Wang, 2007g].

Definition 15.3 An *abstract concept c* is a 5-tuple, i.e.:

$$c \triangleq (O, A, R^c, R^i, R^o) \tag{15.2}$$

where

- O is a nonempty set of object of the concept, $O = \{o_1, o_2, ..., o_m\}$ = ÞU, where ÞU denotes a power set of a finite or infinite nonempty set of objects.
- A is a nonempty set of attributes, $A = \{a_1, a_2, ..., a_n\}$ = ÞM, where M is a finite or infinite nonempty set of attributes.
- $R^c \subseteq O \times A$ is a set of internal relations.
- $R^i \subseteq C' \times C$ is a set of input relations, where C′ is a set of external concepts.

- $R^o \subseteq C \times C'$ is a set of output relations.

A structural concept model of $c = (O, A, R^c, R^i, R^o)$ can be illustrated in Fig. 15.3, where $c, A, O,$ and R, $R = \{R^c, R^i, R^o\}$, denote the concept, its attributes, objects, and internal/external relations, respectively.

Definition 15.4 *Concept algebra* is a new mathematical structure for the formal treatment of abstract concepts and their algebraic relations, operations, and associative rules for composing complex concepts and knowledge [Wang, 2006e].

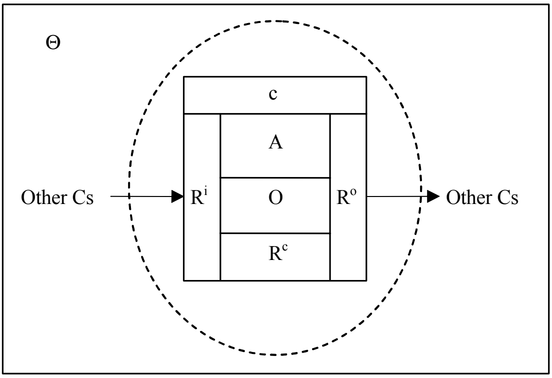


Figure 15.3 The structural model of an abstract concept

Concept algebra deals with the algebraic relations and associational rules of abstract concepts. The associations of concepts form a foundation to denote complicated relations between concepts in knowledge representation. The associations among concepts can be classified into nine categories, such as *inheritance, extension, tailoring, substitute, composition, decomposition, aggregation, specification,* and *instantiation* as shown in Fig. 15.4 [Wang, 2006e]. In Fig. 15.4, $R = \{R^c, R^i, R^o\}$, and all nine associations describe composing rules among concepts, except instantiation that is a relation between a concept and a specific object.

Definition 15.5 A *generic knowledge K* is a relation R_k that mapping a certain concept C into a set of n existing concepts C_i in the brain in the form of OAR, i.e.:

$$K = R_k : C \rightarrow (\mathop{\text{X}}_{i=1}^{n} C_i) \tag{15.3}$$

where $R_k \in \Gamma = \{\Rightarrow, \overset{+}{\Rightarrow}, \bar{\Rightarrow}, \tilde{\Rightarrow}, \uplus, \pitchfork, \Lleftarrow, \vdash, \mapsto\}$.

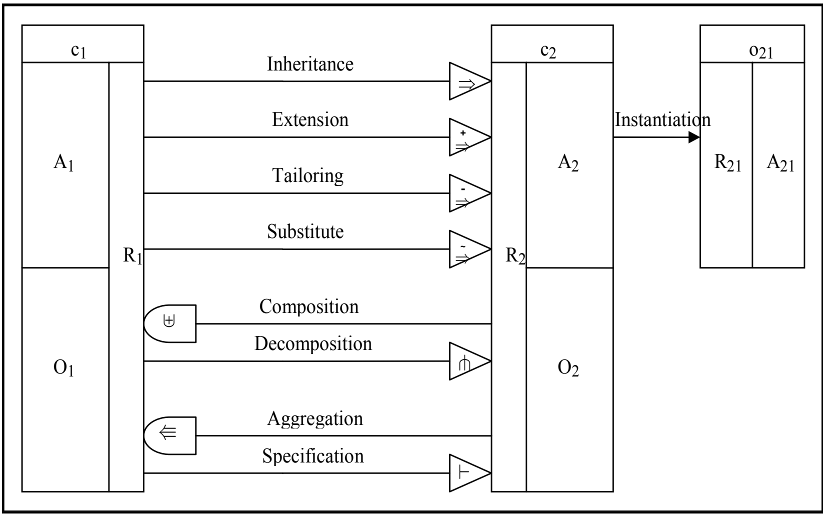


Figure 15.4 Concept association operations in concept algebra

In Definition 15.5 the relation R_k is one of the concept operations defined in concept algebra [Wang, 2006e] that serves as the knowledge composing rules.

Definition 15.6 A *concept network CN* is a hierarchical network of concepts interlinked by the set of nine associations $\Re$ defined in concept algebra, i.e.:

$$CN = R_\kappa : \mathop{\mathsf{X}}_{i=1}^{n} C_i \rightarrow \mathop{\mathsf{X}}_{i=j}^{n} C_j \qquad (15.4)$$

where $R_k \in \Gamma$.

Because the relations between concepts are transitive, the generic topology of knowledge is a hierarchical concept network. The advantages of the hierarchical knowledge architecture K in the form of concept networks are as follows: a) *Dynamic*: The knowledge networks may be updated dynamically along with information acquisition and learning without destructing the existing concept nodes and relational links. b) *Evolvable*: The knowledge networks may grow adaptively without changing the overall and existing structure of the hierarchical network. A summary of the algebraic relations and operations of concepts defined in concept algebra is provided in Table 15.3. Further details may be referred to [Wang, 2006e].

15.3.3.2 System Algebra

Systems are the most complicated entities and phenomena in the physical, information, and social worlds across all science and engineering disciplines. System algebra is a new abstract mathematical structure that provides an algebraic treatment of abstract systems as well as their relations and operational rules for forming complex systems [Wang, 2006d].

System algebra is created for the rigorous treatment of abstract systems and their algebraic relations and operations. A summary of the algebraic relations and operations of abstract systems defined in system algebra is provided in Table 15.3. Further descriptions of system algebra may be referred to Section 10.4 and [Wang, 2006d].

15.3.3.3 RTPA

A key metaphor in system modeling, specification, and description is that a software system can be perceived and described as the *composition* of a set of interacting *processes*. C.A.R. Hoare, R. Milner, and others developed various algebraic approaches to represent communicating and concurrent systems, known as process algebra [Hoare, 1978/85; Milner, 1989]. A *process algebra* is a set of formal notations and rules for describing algebraic relations of software processes. RTPA [Wang, 2002a/02b/03c/06a/07a] extends process algebra to time/event, architecture, and system dispatching manipulations in order to formally describe and specify architectures and behaviors of software systems.

RTPA is a set of formal notations and rules for describing algebraic and real-time relations of software processes. A process in RTPA is a computational operation that transforms a system from a state to another by changing its inputs, outputs, and/or internal variables. A process can be a single meta-process or a complex process formed by using the process combination rules of RTPA known as process relations.

RTPA models 17 meta processes $\mathfrak{P}$ = {:=, �, $\Rightarrow$, $\Leftarrow$, $\nLeftarrow$, ⋗, ⋖, |⋗, |⋖, $\underline{\underline{@}}$, $\triangleq$, $\uparrow$, $\downarrow$, !, $\oslash$, $\boxtimes$, §} and 17 process relations $\mathfrak{R}$ = {$\rightarrow$, $\curvearrowright$, |, |…|, R^*, R^+, R^i, ↺, $\rightarrowtail$, ||, ∯, |||, », ↯, $\hookrightarrow_t$, $\hookrightarrow_e$, $\hookrightarrow_i$}.

Based on RTPA, an important finding about the nature of programs is that according to Theorem 5.7, the generic mathematical model of programs is a finite and nonempty set of cumulatively embedded relational processes between a current statement and all previous ones that formed the semantic context or environment of computing.

The definitions, syntaxes, and formal semantics of RTPA may be referred to Sections 4.6, 4.7, and 6.6, respectively [Wang, 2002a/02b/03c/06a/07a]. A summary of the meta processes and their algebraic operations in RTPA is provided in Table 15.3.

Table 15.3
Taxonomy of Denotational Mathematics for Software Science and Engineering

Operations	Concept Algebra	System Algebra	Real-Time Process Algebra (RTPA)			
			Meta Processes		Relational Operations	
Super/sub relation	$\succ$ / $\prec$	$\sqsupseteq$ / $\sqsubseteq$	Assignment	$:=$	Sequence	$\to$
Related/independent	$\leftrightarrow$ / $\not\leftrightarrow$	$\leftrightarrow$ / $\not\leftrightarrow$	Evaluation	�	Jump	$\curvearrowright$
Equivalent	$=$	$=$	Addressing	$\Rightarrow$	Branch	$\mid$
Consistent	$\cong$		Memory allocation	$\Leftarrow$	Switch	$\mid\ldots\mid\ldots$
Overlapped		Π	Memory release	$\not\Leftarrow$	While-loop	R^{*}
Conjunction	$+$	$\sqcup$	Read	$\gtrdot$	Repeat-loop	R^{+}
Elicitation	$*$		Write	$\lessdot$	For-loop	R^{i}
Comparison	$\sim$		Input	$\mid\gtrdot$	Recursion	$\circlearrowleft$
Definition	$\triangleq$		Output	$\mid\lessdot$	Procedure call	$\rightarrowtail$
Difference		$\boxminus$	Timing	$\underline{\underline{@}}$	Parallel	$\parallel$
Inheritance	$\Rightarrow$	$\Rightarrow$	Duration	$\triangleq$	Concurrence	$\oiint$
Extension	$\overset{+}{\Rightarrow}$	$\overset{+}{\Rightarrow}$	Increase	$\uparrow$	Interleave	$\mid\mid\mid$
Tailoring	$\overset{-}{\Rightarrow}$	$\overset{-}{\Rightarrow}$	Decrease	$\downarrow$	Pipeline	»
Substitute	$\overset{\sim}{\Rightarrow}$	$\overset{\sim}{\Rightarrow}$	Exception detection	!	Interrupt	↯
Composition	$\uplus$	$\uplus$	Skip	$\oslash$	Time-driven dispatch	$\hookrightarrow_{t}$
Decomposition	$\pitchfork$	$\pitchfork$	Stop	$\boxtimes$	Event-driven dispatch	$\hookrightarrow_{e}$
Aggregation/ generalization	$\Lleftarrow$	$\Lleftarrow$	System	§	Interrupt-driven dispatch	$\hookrightarrow_{i}$
Specification	$\vdash$	$\vdash$				
Instantiation	$\mapsto$	$\mapsto$				

The three new structures of denotational mathematics have extended the abstract objects under study in mathematics from basic mathematical entities of numbers and sets to a higher level, i.e., concepts, systems, and behavioral processes. A wide range of applications of the denotational mathematics in the context of software science and engineering has been identified [Wang, 2002b; Wang, 2006d; Wang, 2006e].

Under the overarching structure of denotational mathematics and with the paradigms as shown in Table 15.3, novel mathematical forms and structures, new mathematical entities, engineering applications, and comparative studies on denotational and analytic mathematics will be sought in order to develop a family of denotational mathematical structures for the needs in software science and engineering. This will be described in a succeeding volume of this series of books in software engineering.

15.4 Impacts of Software Science on Computing

This book has revealed that almost all the fundamental problems that could not be solved in the last four decades in software engineering were simply stemmed from the lack of software science. The preceding chapters of this book present the impacts of the multidisciplinary theories and rigorous foundations on software engineering. As a consequence, the discipline of software science is emerging.

On the basis of the software science framework, this section explores how software science and the formalized theories of software engineering may influence the other disciplines synergized in this book, especially computer science and computing methodologies. This will be focused on three important emerging methodologies for computing known as autonomic computing, intelligent software code generation, and hyper-programming [Wang, 2007a].

15.4.1 AUTONOMIC COMPUTING

Recalling the discussions on basic computation models in Section 5.2, autonomic computing is introduced as the latest development in computational machines following automata, Turing machines, and von Neumann machines. The general-purpose computers may do anything unless a specific program is loaded. In other words, they are only a class of general behavioral servos of human instructions [Wang, 2003d/04a/07a]. However, autonomic computers are not only servos, but also instructors and goal-driven controllers [IBM, 2001/06; Pescovitz, 2002; Kephart and Chess, 2003; Murch, 2004; Wang, 2003d/04a/07a/07b/07c/07e/07f].

Autonomic computing is a mimicry and simulation of the natural intelligence possessed by the brain by using generic computers. This indicates that the nature of software in autonomic computing is the simulation and embodiment of human behaviors, and the extension of human capability, reachability, persistency, memory, and information processing speed.

Autonomic computing was first proposed by IBM in 2001, while the history towards autonomic computing, as long as computer science, has been reviewed in Section 5.2.5. IBM perceived that “Autonomic computing is an approach to self-managed computing systems with a minimum of human interference. The term derives from the body's autonomic nervous system, which controls key functions without conscious awareness or involvement [IBM, 2001].” Various studies on autonomic computing have been reported based on this proposal [Pescovitz, 2002; Kephart and Chess, 2003]. The cognitive informatics foundations of autonomic computing have been investigated in [Wang, 2003d/04a/07a/07f].

Based on cognitive informatics theories [Wang, 2002a/03a/07b], autonomic computing is proposed as a new and advanced technology for computing built upon the routine, algorithmic, and adaptive systems as shown in Table 15.4

Table 15.4
Classification of Computing Methodologies and Systems

		Behavior (O)	
		Constant	Variable
Event (I)	Constant	**Routine**	**Adaptive**
	Variable	**Algorithmic**	**Autonomic**
Type of behavior		*Deterministic*	*Nondeterministic*

The first three categories of computing techniques, such as routine, algorithmic, and adaptive computing, as shown in Table 5.4, are imperative. In contrast, the autonomic computing systems do not rely on imperative and procedural instructions, but are dependent on goal, perception, and inference driven mechanisms.

Definition 15.7 An *Imperative Computing* (IC) *system* is a passive system that implements deterministic, context-free, and stored-program controlled behaviors.

Definition 15.8 An *Autonomic Computing* (AC) *system* is an intelligent system that implements nondeterministic, context-dependent, and adaptive behaviors based on goal- and inference-driven mechanisms.

The following subsections explore the theoretical foundations and engineering paradigms of AC.

15.4.1.1 From Imperative Computing to Autonomic Computing

IC relies on stored programs that transfer a computer as a general behavioral implementing machine to different specific intelligent applications. However, AC is based on internal inference engines and goal-driven mechanisms in order to implement autonomic and adaptive computing.

The IC system is a traditional and passive system that implements deterministic, context-free, and stored-program controlled behaviors, where a behavior is defined as a set of observable actions of a given computing system. However, the AC system is an active system that implements nondeterministic, context-dependent, and adaptive behaviors. The AC systems do not rely on instructive and procedural information, but are dependent on internal status and willingness formed by long-term historical events and current rational or emotional goals.

In its AC manifesto, IBM proposed eight conditions setting forth an AC system known as self-awareness, self-configuration, self-optimization, self-maintenance, self-protection (security and integrity), self-adaptation, self-resource-allocation, and open-standard-based [IBM, 2001]. Kinsner pointed out that the above characteristics indicate that IBM perceives AC as a mimicry of human nervous systems [Kinsner, 2007]. In other words, *self-awareness* (consciousness) and *nonimperative* (goal-driven) behaviors are the main characteristics of AC systems [Wang, 2007c/07f].

According to cognitive informatics, the eight characteristics of AC identified by IBM may be sufficient to identify an adaptive system rather than an autonomic system. Because adaptive behaviors can be implemented by IC techniques, but autonomic behaviors may only be implemented by nonimperative and intelligent means. This leads to the formal description of the conditions and basic characteristics of AC, and what distinguish AC systems from conventional IC systems.

The 50th Principle of Software Engineering

Theorem 15.2 The *necessary and sufficient conditions of IC, C_{IC}*, are the possession of *event B_e, time B_t,* and *interrupt B_{int}* driven computational behaviors, i.e.:

$$C_{IC} = (B_e, B_t, B_{int}) \qquad (15.5)$$

As an extension of IC, AC systems are constrained by the following theorem.

The 51st Principle of Software Engineering

Theorem 15.3 The *necessary and sufficient conditions of AC, C_{AC}*, are the possession of *goal B_g* and *inference B_{inf}* driven computational behaviors, in addition to the *event B_e, time B_t,* and *interrupt B_i* driven behaviors, i.e.:

$$\begin{aligned} C_{AC} &= (B_g, B_{inf}, B_e, B_t, B_{int}) \\ &= C_{IC} \uplus (B_g, B_{inf}) \end{aligned} \tag{15.6}$$

Theorem 15.3 reveals the relationship between the computing capabilities of IC and AC systems, which can be stated as follows.

Corollary 15.2 The *behavioral space* of *IC C_{IC}* is a subset of AC *C_{AC}*. In other words, *C_{AC}* is a natural extension of *C_{IC}*, i.e.:

$$C_{IC} \subseteq C_{AC} \tag{15.7}$$

As stated in Theorem 3.4, it is recognized that Artificial Intelligence (AI) is a subset of natural intelligence (NI) [Wang, 2007a]. Therefore, AC may also be referred to natural intelligence and human behaviors. According to the LRMB reference model [Wang et al., 2006], a systematical view towards the formal description and modeling of architectures and behaviors of AC systems is obtained, which explains the functional mechanisms and cognitive processes of the natural intelligence with 39 cognitive processes at six layers known as the *sensation, memory, perception, action, meta,* and *higher cognitive layers* from the bottom up. All fundamental goal-driven and perceptive inferring mechanisms of AC systems can be rigorously described and implemented based on LRMB.

15.4.1.2 Behaviorism Foundations of Autonomic Computing

Behaviorism is a doctrine of psychology and intelligence science that reveals the associations between a given stimulus and an observed response of NI or AI systems developed on the basis of associationism [Sternberg, 1998]. Cognitive informatics classifies human and machine behaviors into four categories known as the *perceptive* behaviors, *cognitive* behaviors, *instructive* behaviors, and *reflective* behaviors [Wang, 2007k]. This section investigates the behavioral spaces and the basic properties of IC and AC.

On the basis of Theorem 15.2 and Definitions 5.65 on the Generic Computing System (GCS), the mathematical model of a generic IC system can be described as follows.

Definition 15.9 The *Imperative Computing System*, §$_{\text{IC}}$, is an abstract logical model of conventional computing platforms denoted by a set of parallel or concurrent computing resources and behaviors as shown in Fig. 15.5.

$$
\begin{array}{ll}
\S_{\text{IC}} \triangleq \text{Imperative-SysID}\mathbf{S} :: & \\
\{\ <\mathop{R}\limits_{i\mathbf{N}=0}^{n_{proc}\mathbf{N}-1} P_i\mathbf{ST}> & \text{// Processes} \\
\ \ \| <\mathop{R}\limits_{addr\mathbf{P}=0}^{n_{MEM}\mathbf{H}-1} \text{MEM}[ptr\mathbf{P}]\mathbf{RT}> & \text{// Memory} \\
\ \ \| <\mathop{R}\limits_{ptr\mathbf{P}=0}^{n_{PORT}\mathbf{H}-1} \text{PORT}[ptr\mathbf{P}]\mathbf{RT}> & \text{// Ports} \\
\ \ \| <\S t\mathbf{TM}> & \text{// The system clock} \\
\ \ \| <\mathop{R}\limits_{k\mathbf{N}=0}^{n_e\mathbf{N}-1} @e_k\mathbf{S} \text{ ↳} P_k> & \text{// } \underline{\text{Event-driven behaviors}} \\
\ \ \| <\mathop{R}\limits_{k\mathbf{N}=0}^{n_t\mathbf{N}-1} @t_k\mathbf{TM} \text{ ↳} P_k> & \text{// } \underline{\text{Time-driven behaviors}} \\
\ \ \| <\mathop{R}\limits_{k\mathbf{N}=0}^{n_{\text{int}}\mathbf{N}-1} @int_k\odot \text{ ↳} P_k > & \text{// } \underline{\text{Interrupt-driven behaviors}} \\
\ \ \| <\mathop{R}\limits_{i\mathbf{N}=0}^{n_V\mathbf{N}-1} V_i\mathbf{RT}> & \text{// System variables} \\
\ \ \| <\mathop{R}\limits_{i\mathbf{N}=0}^{n_S\mathbf{N}-1} \text{ⓢ}S_i\mathbf{BL}> & \text{// System statuses} \\
\} &
\end{array}
$$

Figure 15.5 The imperative computing system model

Fig. 15.5 shows that an IC system §$_{\text{IC}}$ is the executing platform or the operating system that controls all the computing resources of an abstract target machine. The IC system is logically abstracted as a set of process behaviors and underlying resources, such as the memory, ports, the system clock, and system status. An IC behavior in terms of a process P_{k} is controlled and dispatched by the system $\S_{\text{IC}}$, which is triggered by various external, system timing, or interrupt events [Wang, 2007k].

AC extends the conventional behaviors of IC as discussed in the preceding subsection to more powerful and intelligent ones such as goal-driven and inference-driven behaviors. According to Theorem 15.3, with the possessing of all the five forms of intelligent behaviors, AC has advanced closer to the basic intelligent power of human brains from conventional IC.

Definition 15.10 A *goal-driven behavior*, denoted by $\hookrightarrow_{\text{g}}$, is a machine cognitive process in which the *k*th behavior in term of process P_k is triggered by a given goal $@g_k\mathbf{ST}$, i.e.:

$$\mathop{\mathrm{R}}_{k=1}^{n} @g_k\mathbf{ST} \hookrightarrow_{\text{g}} P_k \qquad (15.8)$$

where the goal $@g_k\mathbf{ST}$ is in the system type $\mathbf{ST}$ that denotes a structured description of the goal (Definition 9.30).

Therefore, to some extent, AC is an intelligent goal-driven problem solving machines that searches a solution for a given problem or finds a path to reach a given goal [Rubinstein and Firstenberg, 1995; Chiew and Wang, 2004]. There are two categories of problems in problem solving: a) The *convergent* problems where the goal of problem solving is given but the paths of problem solving may be known or unknown; and b) The *divergent* problems where the goal of problem solving is unknown, but the paths are either known or unknown. The combination of the above cases in problem solving can be summarized in Table 15.5. A special case in Table 15.5 is that when both the goal and path are known, the case is a solved instance of a given problem.

Table 15.5
Classification of Problems and Goals

Type of problem	**Goal**	**Path**	**Type of solution**
Convergent	Known	Unknown	Proof (Specific)
	Known	Known	Instance (Specific)
Divergent	Unknown	Known	Case study (Open-ended)
	Unknown	Unknown	Explorative (Open-ended)

According to Theorem 15.3, *inference capability* is the second extension of AC on top of the capabilities of IC, which is a cognitive process that reasons a possible causality from given premises based on known causal relations between a pair of cause and effect proven true by empirical arguments, theoretical inferences, or statistical regulations.

Definition 15.11 An *inference-driven behavior*, denoted by $↳_{\text{inf}}$, is a machine cognitive process in which the *k*th behavior in terms of process P_k is triggered by a given result of inference process $@inf_k\mathbf{ST}$, i.e.:

$$\mathop{\mathrm{R}}\limits_{k=1}^{n} @inf_k\mathbf{ST} \; ↳_{\text{inf}} \; P_k \tag{15.9}$$

Formal inferences can be classified into the *deductive, inductive, abductive,* and *analogical* categories [Wang, 2007a/07h]. On the basis of the definitions of the behavioral space of AC, a generic AC system may be rigorously modeled below.

Definition 15.12 The *AC System*, $\S_{\text{AC}}$, is an abstract logical model of a computing platform denoted by a set of parallel or concurrent computing resources and behaviors as shown in Fig. 15.6.

15.4.1.3 Cognitive Informatics Foundations of Autonomic Computing

The theory and philosophy behind AC are cognitive informatics [Wang, 2002a/03a/07b/07f]. Cognitive processes of the brain, particularly the perceptive and inference cognitive processes, are the fundamental means for describing AC paradigms, such as robots, software agent systems, and distributed intelligent networks. In recent research in cognitive informatics, *perceptivity* is recognized as *the sixth sense* that serves the brain as the thinking engine and the kernel of the natural intelligence. Perceptivity realizes self-consciousness inside the abstract memories of the brain. Almost all cognitive life functions rely on perceptivity such as *consciousness, memory searching, motivation, willingness, goal setting, emotion, sense of spatiality,* and *sense of motion*.

A fundamental question in cognitive psychology is how consciousness can be the product of physiological processes in the brain. Similarly, the fundamental question for AC is how autonomic behaviors may be generated by nonimperative processes on generic computers. The cognitive models developed in this section reveal, as that of IC is controlled by stored-programs, AC should be controlled by nonprocedural and/or learned cognitive processes by the machines. According to the CMM model (Theorem 9.3), an AC system can be implemented by mimicking the following abstract brain models.

$$
\begin{array}{ll}
\S_{\mathrm{AC}} \triangleq \text{Autonomic-SysID}\mathbf{S} :: & \\
\{ \quad < \mathop{R}\limits_{i\mathbf{N}=0}^{n_{proc}\mathbf{N}-1} P_i\mathbf{ST}> & // \text{ Processes} \\
\quad || < \mathop{R}\limits_{addr\mathbf{P}=0}^{n_{MEM}\mathbf{H}-1} \text{MEM}[ptr\mathbf{P}]\mathbf{RT}> & // \text{ Memory} \\
\quad || < \mathop{R}\limits_{ptr\mathbf{P}=0}^{n_{PORT}\mathbf{H}-1} \text{PORT}[ptr\mathbf{P}]\mathbf{RT}> & // \text{ Ports} \\
\quad || <\S t\mathbf{TM}> & // \text{ The system clock} \\
\quad || < \mathop{R}\limits_{k\mathbf{N}=0}^{n_e\mathbf{N}-1} @e_k\mathbf{S} \hookrightarrow P_k> & // \textit{ Event}\text{-driven behaviors} \\
\quad || < \mathop{R}\limits_{k\mathbf{N}=0}^{n_t\mathbf{N}-1} @t_k\mathbf{TM} \hookrightarrow P_k> & // \textit{ Time}\text{-driven behaviors} \\
\quad || < \mathop{R}\limits_{k\mathbf{N}=0}^{n_{\mathrm{int}}\mathbf{N}-1} @int_k\odot \hookrightarrow P_k > & // \textit{ Interrupt}\text{-driven behaviors} \\
\quad || < \mathop{R}\limits_{k\mathbf{N}=0}^{n_t\mathbf{N}-1} \boldsymbol{@g_k}\mathbf{ST} \hookrightarrow P_k> & // \text{ }\underline{\text{Goal-driven behaviors}} \\
\quad || < \mathop{R}\limits_{k\mathbf{N}=0}^{n_{\mathrm{int}}\mathbf{N}-1} \boldsymbol{@inf_k}\mathbf{ST} \hookrightarrow P_k > & // \text{ }\underline{\text{Inference-driven behaviors}} \\
\quad || < \mathop{R}\limits_{i\mathbf{N}=0}^{n_V\mathbf{N}-1} V_i\mathbf{RT}> & // \text{ System variables} \\
\quad || < \mathop{R}\limits_{i\mathbf{N}=0}^{n_S\mathbf{N}-1} ⓈS_i\mathbf{BL}> & // \text{ System statuses} \\
\} &
\end{array}
$$

Figure 15.6 The autonomic computing system model

Definition 15.13 The *cognitive informatics model of an AC system, ACS*, is equivalent to the high-level logical model of the brain as given in Model 9.6, i.e.:

$$
\begin{aligned}
ACS \triangleq\ & NI\_Sys \\
& || \, CMM \\
=\ & (\ NI\_OS \\
& \quad || \, NI\_App \\
& \ ) \\
& || \, (\ LTM \\
& \quad || \, STM \\
& \quad || \, SBM \\
& \quad || \, ABM \\
& \ )
\end{aligned}
\qquad (15.10)
$$

Eq. 15.10 indicates that although the thinking engine *NI-Sys* is considered the center of the natural intelligence, the memories are essential to enable the *NI-Sys* to properly functioning and to keep internal information and acquired knowledge stable and retrievable.

15.4.1.4 Denotational Mathematics Foundations of Autonomic Computing

As that of IC is based on the mathematical foundation of Boolean algebra, the more intelligent capability of AC should be processed by more powerful mathematical structures known as denotational mathematics in the forms of system algebra [Wang, 2006d], concept algebra [Wang, 2006e], and RTPA [2002a] as described in Section 15.3. The three new structures of contemporary mathematics extend the abstract objects under study in mathematics from basic entities such as numbers and sets to complex ones such as concepts, systems, and behavioral processes as shown in Table 15.2.

It is recognized that the intelligent inference capability of AC systems is based on the cognitive process of abstraction. *Abstraction* is not only a powerful means of philosophy and mathematics, but also a preeminent trait of the human brain identified in cognitive informatics studies [Wang, 2007a/07h]. All formal logical inferences and reasoning, as described in Section 3.3, can only be carried out on the basis of abstract properties shared by a given set of objects under study. Detailed descriptions of the formal cognitive inference processes for AC may be referred to Section 3.3 [Wang, 2007h], which can be used to simulate machine cognitions and the implementation of inference engines for AC systems on the basis of denotational mathematics.

15.4.1.5 Intelligence Science Foundations of Autonomic Computing

Intelligence is perceived as the driving force or the ability to acquire and use knowledge and skills, or to reason in problem solving. It was conventionally perceived that only human beings possess advanced intelligence. However, the development of computers, robots, and autonomic systems indicates that intelligence may also be created or implemented by machines and man-made systems. This is the intelligent behavioral foundation for designing and implementing AC systems.

The Generic Intelligence Model (GIM) and the nature of intelligence have been described in Section 9.3.3. The GIM model and Theorem 9.4 reveal that NI and AI share the same cognitive informatics foundations. In other words, they are compatible. Therefore, on the basis of Theorem 9.4, the studies on NI and AI may be unified into a common framework in the context of AC.

The intelligent behavioral foundations of AC as given in the GIM model provide a new paradigm of AC systems, in which an AC system may not only implement the reflective and instructive intelligence, but also implement the cognitive and perceptive intelligence according to the theory of the intelligent behavioral paradigm.

15.4.2 INTELLIGENT CODE GENERATION

One of the key objectives of software engineering is to increase the productivity of software development by intelligent code generation. That is, strategic software engineering methodologies and technologies should focus on how people may be released from coding rather than bound with it. However, it is recognized that software engineering, as one of the high-tech disciplines, is using the lowest-tech – human labor – in contingent software development [Wang, 2006a]. Software scientists and engineers were busy most of the time to study the approaches to help customers to develop specific software applications. Now, it seems to be the time to turn the focus on the basic needs of the discipline rather than on contingent and individual applications, i.e., theories, methodologies, and universal tools for automatic code generation for the entire range of applications in software engineering.

According to the Engineering Objective Model (EOM) model as stated in Theorem 8.2, productivity is the principal objective and major purpose of any engineering discipline, particularly in software engineering [Bain, 1962; Wang, 2006a]. However, the productivity of software development has remained considerably low in the last four decades [Boehm, 1987; Dale and Zee, 1992; Jones, 1981/1986; Livermore, 2005], because it is found that human creative and cognitive productivity is conservative as stated in Theorem 1.6 and as described in Section 9.5.1 [Wang, 2007a]. Therefore, the improvement of software engineering productivity by technical innovation is the key to achieve other important engineering objectives such as quality and time-to-the-market. The automatic switching system revolutions in the 1940s and 1990s demonstrated how technical innovations have helped to improve productivity in the telecommunication industry [Wang and Patel, 2000]. Thus, it is inevitable that software science and engineering should set its paramount goal on the improvement of productivity in software development by automatic software code generators using cognitive and intelligent methodologies on the basis of rigorous mathematical models of software systems in denotational mathematics [Wang, 2006j].

In software engineering, automatic software generation has been recognized as a tough challenge, because of its inherent complexity and the lack of suitable mathematical means [McDermid, 1991; Brooks, 1975/95; Bjorner, 2000]. The investigation into intelligent and automatic software code generation will focus on theories, methodologies, supporting tools, and environments for code generation. The latest advances in software engineering [McDermid, 1991;Pressman, 1992; Sommerville, 1996; Pfleeger, 1998; Peters and Pedrycz, 2000; Vliet, 2000; Wang and King, 2000a; Wang and Patel, 2000; Broy and Denert, 2002; Wang and Bryant, 2002; Wang, 2000/05a/05d/05f/05g/05h/05i/05j/05k/05l/06a/06c/06f/06g/06h/06i] and *cognitive informatics* [Wang, 2002a/2006c/2007a; Wang and Kinsner, 2006] have provided a rich set of theoretical foundations for the design and implementation of intelligent and automatic program generation systems on the basis of formal system models and semantics [Hoare, 1969; Scott and Strachey, 1971; Wegner, 1972; Ollongren, 1974; Dijktra, 1975/76; Guttag and Horning, 1978; Jones, 1980; Gries, 1981; Bjorner and Jones, 1982; Scott, 1982; Marcotty and Ledgard, 1986; Wikstrom, 1987; Schmidt, 1988/94/96; Goguen et al., 1977/96; Wang, 2006a], particularly the latest development of the *denotational mathematics* known as *system algebra*, *concept algebra*, and RTPA [Wang, 2002a/05a/06d/06e/06f/06j/07a], as well as deductive semantics [Wang, 2006a].

A pilot C++ and Java code generation system based on RTPA has been designed and implemented [Tan, Wang, and Ngolah, 2006]. By the integration of denotational mathematics, deductive semantics, and cognitive models of formal inferences [Wang, 2007h], the mathematical and cognitive means will be adequate to design and implement an autonomic and intelligent software code generation system, which will seamlessly and autonomously transfer the mathematical model of a software system into code. As a result, the outcomes of this program will enable the release of human labor from the late-phase processes in software development.

Therefore, intelligent code generators will lead to the development of cutting-edge techniques for the software industry in order to replace the intensive labor-dependent programming practice in software engineering.

15.4.3 HYPER-PROGRAMMING: NEW FACETS OF THE SOFTWARE ARCHITECTURAL FRAMEWORK

As discussed in Section 10.6.2, the work products of different software engineering processes are different. How to integrate all these abstract work products into a coherent framework in order to improve software descriptivity and integrity is a critical need in software engineering. *Hyper-programming* is a new software engineering methodology that intends to extend the descriptive power of multi-facet software architectures and behaviors, and to extend the scope of documentation to cover all software engineering processes and their workproducts in a coherent framework [Wang, 2005g; Huang and Wang, 2006; Wang et al., 2008].

This section presents the hyper-programming methodology and tool for integrated and coherent software engineering documentation. A hyper-programming tool is designed for automatically creating hyperlinks between system conceptual models in UML, formal models in RTPA, and code in a programming language such as C++ and Java. The three types of design documents for a system in UML, RTPA, and C++ program are stored in a standard HTML file format. When a built-in hyperlink in a system model is clicked, the corresponding HTML page in the integrated file shows up. The hyper-programming method provides a powerful and convenient integration of traditionally separated system design documents by hyperlinks in a coherent environment. Under the support of the hyper-programming tool, programmers can traverse from any point of interested objects to any other one among the conceptual and formal models of systems as well as corresponding code. Therefore, the readability and maintainability of large-scale software systems may be dramatically improved.

15.4.3.1 The Architecture of Hyper-Programming

Definition 15.14 A *hyper-program* is a new type of nonlinear framework for software description and documentation that integrates software architectures, behaviors, code, and related design workproducts into a coherent and multidimensional framework by bidirectional hyperlinks.

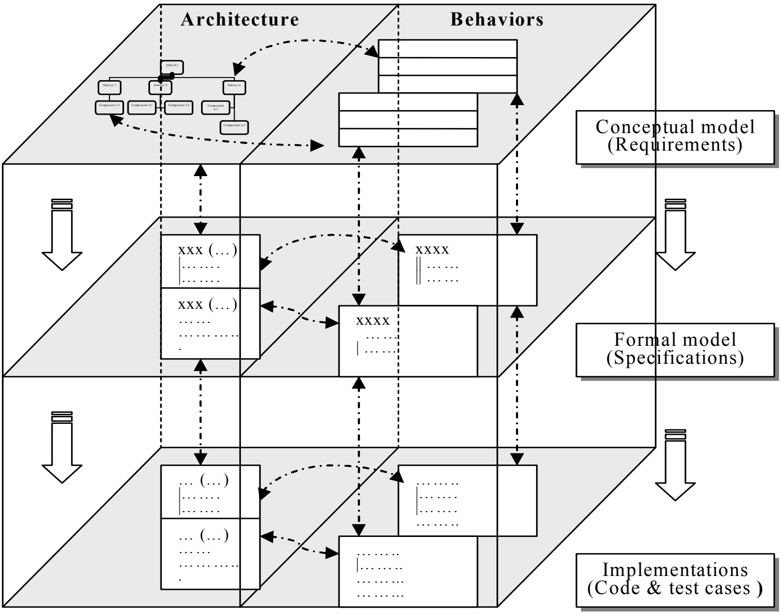


Figure 15.7 The architecture of hyper-programs

The major characteristics of a hyper-program are that its architecture is multidimensional rather than linear in program code documentation, and it integrates all design workproducts in software engineering processes, such as requirements analysis, system specification, system architecture, code, test cases, and system configuration/deployment in a coherent framework as shown in Fig. 15.7.

It is recognized that system architects, programmers, maintainers, and customers need an integrated programming and documentation platform to read and check system consistency among all forms of design and implementation documents and intermediate work products in software engineering [Wang, 2005g]. This leads to the design of hyper-programming methodology [Huang and Wang, 2006; Wang et al., 2008] as shown in Fig. 15.8, where L_{UR}, L_{RU}, L_{RC}, and L_{CR} denote the hyperlinks between UML-RTPA, RTPA-UML, RTPA-C++, and C++-RTPA, respectively.

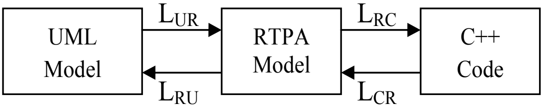


Figure 15.8 The integrated hyper-programming framework

More formally, the hyperlinks can be defined below.

Definition 15.15 A *hyperlink* is a pointer $\curvearrowright$ in a hypertext that refers and transfers a term to another in the scope of the same document or separate documents. For instances:

$$\begin{aligned} L_{UR} &\triangleq E_{UML} \curvearrowright E_{RTPA} \\ L_{RU} &\triangleq E_{RTPA} \curvearrowright E_{UML} \\ L_{RC} &\triangleq E_{RTPA} \curvearrowright E_{C++} \\ L_{CR} &\triangleq E_{C++} \curvearrowright E_{RTPA} \end{aligned} \tag{15.11}$$

where E_{UML}, E_{RTPA}, and E_{C++} are a syntactical entity in UML, RTPA, and C++, respectively.

The hyper-programming system is designed to automatically create hyperlinks between different levels of system documentation from the conceptual model and formal specification to code in the abstraction and refinement hierarchy of system design as stated in Theorem 1.3 [Wang, 2005g]. Hyper-programming implements system documentation integration by hyperlinks among UML class diagrams, RTPA specifications, and C++ source codes (or in other programming languages).

15.4.3.2 Syntactic Relations between RTPA, UML, and C++

This subsection comparatively analyzes the syntaxes of RTPA [Wang, 2002a/02b/03c/07a], UML [Rumbaugh et al, 1998; OMG, 2005; Wang, 2001a], and C++ [Stroustrup, 1986; Horstmann and Budd; 2004]. RTPA is used to formally and explicitly specify software systems in order to enhance the understandability of their architecture, semantics, and behaviours. The architecture of a C++ program can be outlined by RTPA as shown in Fig. 15.9, which provides a formal description of the generic architecture of C++ programs.

$$
\begin{array}{l}
\text{C++Program} \triangleq \{ \\
\qquad\qquad \text{MacroDefinition} \\
\qquad\quad \| \ \text{MainFunction} \\
\qquad\quad \| \ \text{ClassImplementation} \\
\qquad\quad \} \\
\text{C++Program.Mainfunction} \triangleq \{ \\
\qquad\qquad \text{DataStructureDeclaration} \\
\qquad\quad \| \ \text{I/ODeclaration} \\
\qquad\quad \| \ \text{ClassDeclaration} \\
\qquad\quad \| \ \text{SystemFunction} \\
\qquad\quad \} \\
\text{C++Program.Mainfunction.ClassDeclaration} \triangleq \{ \\
\qquad\qquad \mathop{R}\limits_{i=1}^{n} \text{AttributeDeclaration } (\text{i}\mathbf{N}) \\
\qquad\quad \| \ \mathop{R}\limits_{i=1}^{n} \text{MethodDeclaration } (\text{i}\mathbf{N}) \\
\qquad\quad \} \\
\text{C++Program.ClassImplementation} \triangleq \{ \\
\qquad\qquad \mathop{R}\limits_{i=1}^{n} \text{MethodsImplementation } (\text{i}\mathbf{N}) \\
\qquad\quad \}
\end{array}
$$

Figure 15.9 The formal model of a generic C++ program architecture

(a) RTPA vs. C++

The focuses of hyper-programming are class declarations and implementations. Hyper-programming is designed to map between RTPA class specifications and C++ class implementations. Class declaration, method declaration, and method implementation in C++ are mapped to corresponding RTPA syntactical entities in terms of processes.

Class specifications in RTPA have been formalized in [Vu and Wang, 2004; Wang and Huang, 2005]. In RTPA, the type suffix is used to denote type information of a given identifier. For example, *x***N** means the identifier *x* is in the type integer, and an identifier suffixed by **AC** and **CC** denotes a class. For example, *classID***AC** and *classID***CC** are used to denote classes in types of abstract class **AC** and concrete class **CC**, respectively.

(b) RTPA vs. UML

A hyperlinked RTPA specification and C++ code is shown in Fig. 15.10, where the bidirectional hyperlinks between class identifiers *CivicFactory***CC** and *Architecture***ST** in RTPA, and the class name CivicFactory**AC** in C++ source code can be automatically created. Readers interested in the design and implementation of the system can traverse freely among system design models, specifications, and code in an integrated and coherent environment.

The process specifications of static behaviors in RTPA are corresponding to the class method declaration and implementation in C++, respectively. A process declaration in the static behavior section in RTPA is mapped to a class/method declaration in a class, while a process definition in the static behaviors is mapped to a class/method implementation. Because UML is a diagram-based system modeling language [OMG, 2005], a UML diagram seems highly readable. However, its semantics is inaccurate and nonrigorous. Different persons may obtain different information and perceive different meanings from a UML model. UML diagrams may be used to describe the conceptual models of software systems for human communication, particulally for non-professional customers.

The HTML format of UML class diagrams is adopted in hyper-programming for inserting and processing the required hyperlinks. In the HTML format of the UML model, a class diagram is denoted by a group of three rectangle boxes with texts denoting the class name, attributes, and member methods, respectively. The text in the second or third box may be omitted for a specific case.

```
// Formal Specification in RTPA
CivicFactoryCC ≙ {
   [LRU1 | LRC1]          Architecture: ST
                         || StaticBehaviors: ST
                         || DynamicBehaviors: ST
                         }
CivicFactoryCC.ArchitectureST ≙ {
                     [LRU2 | LRC2]       <WheelObj : CivicWheelST>;
                                         <BodyObj : CivicBodylST>;
                                         …
                                        }
CivicFactoryCC.StaticBehaviorsST ≙ {
    CreateFactory (<I:: FactoryNameS>; <O:: >)
     [LRU3 | LRC3]
  || CreateWheel (<I:: NumInstN>; <O:: CivicWheelInstST>)
     [LRU4 | LRC4]
  || CreateBody (<I:: ColorInstS>; <O:: CivicBodyInstST>)
     [LRU5 | LRC5]
}
HondaFactoryCC() : CivicFactoryCC
    [LRU6 | LRC6]       { … }
```

```
// Source Code in C++
class CivicFactoryCC {
            [LCR1]
       public:
           void  CreateFactory(char FactoryNameS);
           CivicWheelInstST  CreateWheel(int NumInstN);
           CivicWheelInstST  CreateBody(char *ColorInstS);
       private:
        [LCR2]
           CivicWheel WheelObj;
                CivicBody   BodyObj;
};
void CreateFactory (char FactoryNameS)
{ …       [LCR3]
}
HondaWheel* CivicFactoryCC::CreateWheel (int NumInstN)
{ …                                    [LCR4]
  return WheelObj.CreateWheel (NumInstN);
}
HondaBody* CivicFactoryCC::CreateBody (char *ColorInstS)
{ …                                   [LCR5]
  return BodyObj.CreateBody(*ColorInstS);
}
class HondaFactoryCC : public CivicFactoryCC
{ …          [LCR6]
}
```

Figure 15.10 Mapping between RTPA models and C++ code

The UML class diagram for a software pattern know as *Civic factory* with hyperlinks is shown in Fig. 15.11, where a hyperlink L_{URi} in the UML model is corresponding to a reverse hyperlink in the RTPA specification L_{RUi}.

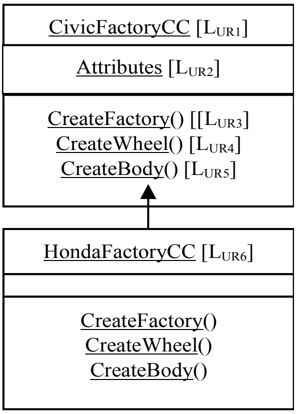

CivicFactoryCC [L_{UR1}]
Attributes [L_{UR2}]
CreateFactory() [[L_{UR3}]
CreateWheel() [L_{UR4}]
CreateBody() [L_{UR5}]
HondaFactoryCC [L_{UR6}]
CreateFactory()
CreateWheel()
CreateBody()

Figure 15.11 Hyperlinks created in a UML class diagram

15.4.3.3 The Framework of the Hyper-Programming Environment

A hyper-programming environment encompasses four major components, as shown in Fig. 15.12, which are the file scanners, the parsers, the hyperlink generator, and the hyperlinked file generator.

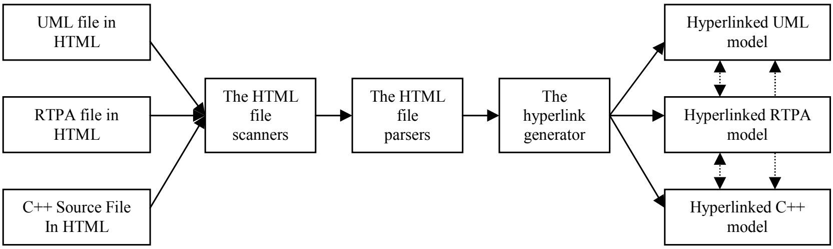


Figure 15.12 The framework of the hyper-programming system

(a) The File Scanners

There are three file scanners in hyper-programming for processing UML, RTPA, and C++ files. The C++ *source file scanner* is a preprocessor that separates a C++ source file into a list of syntactical tokens. The *RTPA file scanner* filters the HTML tags and translates the HTML escape characters into ASCII characters. It also translates the special hexadecimal characters into ASCII characters. The *UML file scanner* analyzes the HTML tags and separates class information from other HTML style tags.

(b) The Parsers

Corresponding to the three file scanners, hyper-programming implements three parsers for each type of the input files in UML, RTPA, and C++ under the support of ANTLR [Parr, 2000]. The *C++ parser* analyzes each token and creates a set of class parser trees. An HTML node in the class tree includes the class name, method name, node type, file position, tag value, and reference tag value. The *RTPA parser* analyzes the inputted tokens and creates a set of class trees. Every node in the tree includes the same information as in the C++ parser. Similarly, the *UML parser* creates equivalent parser trees for the UML file in the HTML format.

Hyperlinks are created based on syntactical equivalence in the three types of documents, such as identical names of classes/processes, methods/processes, and/or variables as shown in Figs. 15.9 and 15.10. For a given hyper node, the file positions include an identifier's start and end positions, which are used to mark the location of an identifier in the token file. When a link needs to be generated, the start point is marked by <a> and the end point is marked by </a> in an HTML tag. If two links like [L_{RUi} | L_{RCi}] need to be created for a certain identifier as shown in Fig. 15.9, the middle position of it is calculated. The first link is marked from the start position to the middle position; and second link is marked from the middle position to the end position. Different cursor styles are employed for the identifier that has been inserted in the bidirectional links. Identifiers in the RTPA file usually have two links: One is linked to UML diagram (L_{RUi}), while the other is for the C++ source code (L_{RCi}).

The tag value of a node is used to generate HTML tag <a> name attribute, where the identifier enclosed by <a> tag can be referred by its name attribute. The reference tag value is used to generate the HTML tag <a> *href* attribute, where a click on the identifier will bring the user to the file position defined by the *href* attribute. Tag and reference tag are cross referenced. Tag value in the node of RTPA parser tree is used as the reference tag value in C++ parser node. Tag values in the node of C++ parser tree are used as reference tag value in RTPA parser node. Tag value in the node of UML class tree is used as the second reference tag value for the node of the RTPA parser.

(c) The Hyperlink Generator

The hyperlink generator utilizes class names in the indexed parser trees to create reference tags. For example, when a class name, method name, or node type is identical in the C++ and RTPA parser trees, the hyperlink generator will copy the tag value of the node in C++ parser tree to the reference tag value in the node of RTPA parser tree and update position indexed list at the same time.

(d) The Hyper-Program Generator

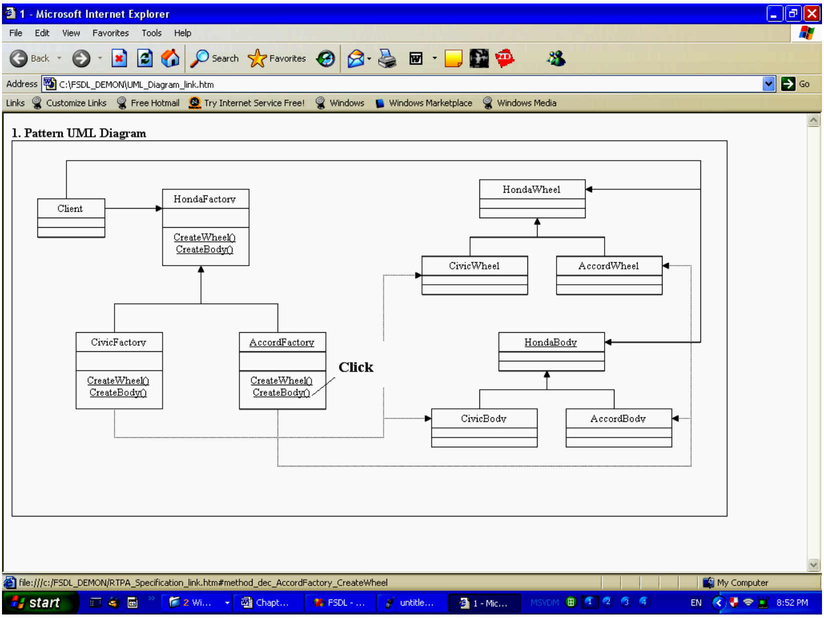
Hyper-programming generates HTML files with built-in hyperlinks. The C++ *hyperlinked file generator* scans a source file and compares the file pointer with file start position stored in the position indexed node list. If the current file position equals to the file start position stored in the list, it inserts the HTML <a> tag and corresponding attributes set for that tag. If the current file position equals to the file end position stored in the previous node it inserts an HTML </a> tag.

Other two types of hyperlinked files in RTPA and UML use the same method.

15.4.3.4 Applications of the Hyper-Programming System

The hyper-programming system has been successfully implemented as an integrated system programming and documentation tool [Huang and Wang, 2006]. An application case study of hyper-programming is demonstrated below.

The *Abstract factory* is one of the popular software design patterns [Gamma et al., 1995; Vu and Wang, 2004; Huang and Wang, 2006]. A virtual *Honda* vehicle factory can be derived on the basis of the abstract factory, which consists of the UML model as shown in Fig. 15.13 with automatically generated hyperlinks. The virtual *Honda* factory builds two models called the *Civic* and *Accord*, respectively. A vehicle in the factory is composed by two components known as the *wheel* and *body*.



1 - Microsoft Internet Explorer

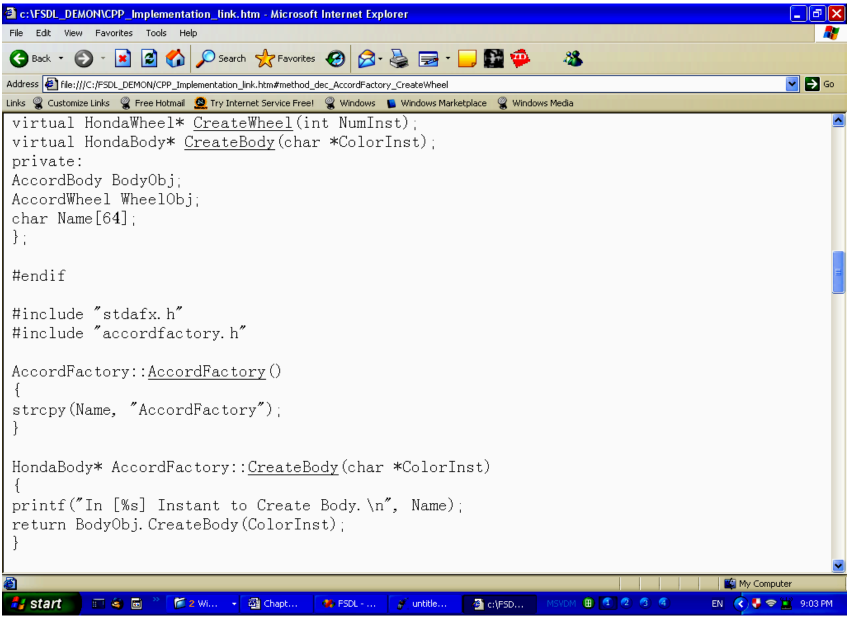
Address C:\FSDL_DEMON\UML_Diagram_link.htm

1. Pattern UML Diagram

file:///c:/FSDL_DEMON/RTPA_Specification_link.htm#method_dec_AccordFactory_CreateWheel

Figure 15.13 UML class diagram with hyperlinks

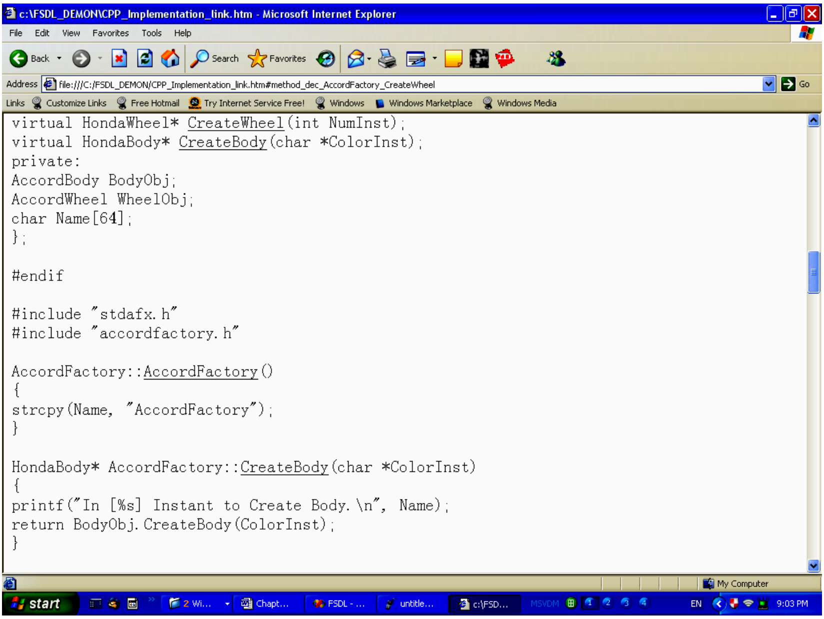
When clicking on the name and member methods of the class, the system jumps to the corresponding syntactical entities in the RTPA specification. For example, if the method *CreateBody* in the class *AccordFactory* is clicked, the page as shown in Fig. 15.11 will show up. If one moves the mouse to the front part of *AccordFactoryCC.CreateBody* text and clicks on it the page listed in Fig. 15.13 will be returned. However, if the click is on the rear part of *AccordFactoryCC.CreateBody,* the C++ source code as implemented in Fig. 15.14 will be displayed. If one clicks the text with hyperlinks in C++ documentation, the corresponding RTPA specification is shown up appropriately to explain the design notions as shown in Fig. 15.15.


c:\FSDL_DEMON\CPP_Implementation_link.htm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address file:///C:/FSDL_DEMON/CPP_Implementation_link.htm#method_dec_AccordFactory_CreateWheel
Go
Links Customize Links Free Hotmail Try Internet Service Free! Windows Windows Marketplace Windows Media
My Computer
start 2 Wi... Chapt... FSDL - ... untitle... c:\FSD... MSVDM EN 9:03 PM

```
virtual HondaWheel* CreateWheel(int NumInst);
virtual HondaBody* CreateBody(char *ColorInst);
private:
AccordBody BodyObj;
AccordWheel WheelObj;
char Name[64];
};

#endif

#include "stdafx.h"
#include "accordfactory.h"

AccordFactory::AccordFactory()
{
strcpy(Name, "AccordFactory");
}

HondaBody* AccordFactory::CreateBody(char *ColorInst)
{
printf("In [%s] Instant to Create Body.\n", Name);
return BodyObj.CreateBody(ColorInst);
}
```

Figure 15.14 C++ source code with hyperlinks

Two pairs of the bidirectional hyperlinks, as shown in Fig. 15.8, have been built among UML, RTPA, and C++ in hyper-programming. The hyper-programming methodology and tool provide a handy and easy environment for designers and users to create and integrate hyper-programming and design documentation in a coherent system.


c:\FSDL_DEMON\CPP_Implementation_link.htm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address file:///C:/FSDL_DEMON/CPP_Implementation_link.htm#method_dec_AccordFactory_CreateWheel
Go
Links Customize Links Free Hotmail Try Internet Service Free! Windows Windows Marketplace Windows Media
My Computer
start 2 Wi... Chapt... FSDL - ... untitle... c:\FSD... MSVDM EN 9:03 PM

```
virtual HondaWheel* CreateWheel(int NumInst);
virtual HondaBody* CreateBody(char *ColorInst);
private:
AccordBody BodyObj;
AccordWheel WheelObj;
char Name[64];
};

#endif

#include "stdafx.h"
#include "accordfactory.h"

AccordFactory::AccordFactory()
{
strcpy(Name, "AccordFactory");
}

HondaBody* AccordFactory::CreateBody(char *ColorInst)
{
printf("In [%s] Instant to Create Body.\n", Name);
return BodyObj.CreateBody(ColorInst);
}
```

Figure 15.15 RTPA specification with hyperlinks

Frederick Brooks (1987) pointed out that there was “no silver bullet” in software engineering. However, recalling what Albert Einstein said that: “Problems that are created by our current level of thinking cannot be solved by that same level of thinking,” it may be perceived that the silver bullet for software engineering lies in software science, which provides the theoretical foundations and fundamental methodologies for software engineering.

In the conclusion of this book, the author believes that readers have obtained a comprehensive and adequate set of theoretical and empirical foundations for software engineering on the basis of software science. Thus, the ideal pyramid structure and relationship between software science, software engineering education, and its practices/applications set forth in the beginning of this book as shown in Fig. 1.1 has been satisfactorily established.

15.5 Epilogue

Throughout this book, I have used the third person mode in presentations, descriptions, discussions, and reasoning in order to maintain an objective view towards the investigation of theoretical and empirical foundations of software engineering. Now, I would like to communicate with readers in the mode of the first person at the end of this book.

In *Art of Scientific Investigation*, W.I. Beveridge (1957) recalled that Claude Bernard (1813-1878) reportedly expressed:

> "Those who do not know the torment of the unknown cannot have the joy of discovery."

I would like to extend Bernard's assertion in order to say that those who do not experience the torment of authoring a large volume book cannot know the cognitive complexity that one may face, the mental tenacity that one may need, the intensive satisfaction that one may gain, and the great joy that one may obtain.

I was greatly unsatisfied about the phenomenon that there was a lack of coherent theoretical foundations for software engineering and most people in this field were used to taking it for granted. The phenomenon that kids are able to programming and sometimes even do it better is an indication that the discipline of software engineering was immature. Based on the development of the coherent theories for software engineering in this book and new challenges discovered at the edge of the exploration, I wish readers will find that software engineering and software science will be better built on the basis of rigorous fundamental researches.

When I checked the electronic manuscript for the final time, I found that I have commutatively spent over 10,000 hours on this book in the last decade, and the final version of this book contains about 356,800 words. It is indeed a persistent effort that leads to the completion of this book, which reminds me what Louis Pasteur (1822-1895) described on his academic motivation [Beveridge, 1957]:

> "Let me tell you the secret that has led me to my goal. My only strength lies in my tenacity."

Once working at Oxford University, C.A.R. Hoare told me that theories are durable while techniques are temporary. He is perfectly in line with Immanuel Kant (1724-1804) who believed: “*There is nothing more practical than a good theory*.” The message is that as long as the fast development of software engineering methodologies remains viable, software engineering techniques will need to evolve as quickly. However, what will be kept stable and durable are the fundamental principles and the formally documented theories and laws of software science and software engineering, which are crystallized during age-long elicitations, evaluations, verifications, and refinements.

In concluding this book, I would like to quote what David L. Parnas expressed in his keynote in FSE/ESEC’97 in Zurich [Parnas, 1997]:

> “Things that were valuable a decade ago will be valuable decades from now. The field is moving too fast to chase them. ”

Bibliography

ABET (1986), *1985 Annual Report,* Accreditation Board for Engineering and Technology (ABET) New York, NY.

Abran, A., P. Bourque, and R. Dupuis (1999), Progress on the Fundamental Principles of Software Engineering, *Proc. 3rd IEEE International Software Engineering Standards Symposium* (ISESS'99), IEEE CS Press, Curitiba, Brazil, May.

Adewumi, A. and Y. Wang (2004), Formal Description of a Generic Graph Model with RTPA, *Proc. 17th Canadian Conference on Electrical and Computer Engineering* (CCECE'04), IEEE CS Press, Niagara Falls, ON, Canada, May, pp. 1537-1540.

Aho, A.V. and J.D. Ullman (1972), *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing*, Prentice Hall, Englewood Cliffs, NJ.

Aho, A.V., R. Sethi, and J.D. Ullman (1985), *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publication Co., New York.

Albrecht, A.J. and J.E. Gaffney (1983), Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, IEEE Transactions on Software Engineering, 9(6), pp.639-648.

Alexander, C. et al. (1977), *A Pattern Language,* Oxford Univ. Press, NY.

Anthony, R. N. (1965*), Planning and Control Systems: A Framework for Analysis,* Harvard University Graduate School of Business Administration, Cambridge, MA.

Arbib, M.A. (1969), Theories of Abstract Automata, Prentice Hall, Englewood Cliffs, NJ.

Arbib, R., A. Michael, and J.L. Rhodes (1968), Complexity and Graph Complexity of Finite State Machines and Finite Semi-Groups, in M.A. Arbib *ed.*, Algorithmic Theory of Machines, Languages and Semi-Groups, Academic Press, NY, pp. 127-145.

Aristotle, *Morality and Human Nature* (1925), The Nicomachean Ethics, W.D. Ross trans., Oxford University Press, Oxford.

Arnold, A. and I. Guessarian (1996), *Mathematics for Computer Science, Prentice Hall International*, London, UK.

Arnowitz, J., M. Arent, and N. Berger (2006), Effective Prototyping for Software Makers, Morgan Kaufmann.

Aron, J, D. (1983), *The Program Development Process, Part 2 – The Programming Team*, Addison-Wesley, Reading, MA.

Ashby, W.R. (1956), *An Introduction to Cybernetics*, Chapman & Hall.

Ashby, W.R. (1958a), General Systems Theory as a New Discipline, *General Systems Yearbook*, 3(1).

Ashby W.R. (1958b), Requisite Variety and Implications for Control of Complex Systems, *Cybernetica*, 1, pp. 83-99.

Ashby W.R. (1962), Principles of the Self-Organizing System, in: Principles of Self-Organization, von Foerster H. and Zopf G. (eds.), Pergamon, Oxford, p. 255-278.

Ashby, W.R. (1970), Information Flows Within Coordinated Systems, in J. Rose *ed.*, Progress in Cybernetics, Vol. 1, Gordon and Breach, London, pp. 57-64.

Ashby, W.R. (1972), Systems and Their Informational Measures, in G.J. Klir *ed.*, *Trends in General Systems Theory*, Wiley, NY, pp. 78-97.

Ashby, W.R. (1973), Some Peculiarities of Complex Systems, *Cybernetic Medicine*, 9:2, pp. 1-6.

Ashenhurst, R.L. and S. Graham (1987)*, ACM Turing Award Lectures, The First Twenty Years: 1966 - 1985*, Anthology Series, ACM Press, Addison-Wesley Publishing Company, NY, pp. 458 - 466.

Aspray, W., R. Keil-Slawik, and D.L. Parnas (1996), *History of Software Engineering*, Dagstuhl Seminar Series Report #9635, Germany, August.

Backus, J. (1978), Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, The 1977 Turing Award Lecture, *Communications of the ACM*, 21(8), pp. 613-641.

Baddeley, A. (1990), *Human Memory: Theory and Practice*, Allyn and Bacon, Needham Heights, MA.

Baeten, J.C.M. and J.A. Bergstra (1991), Real Time Process Algebra, *Formal Aspects of Computing*, 3, pp.142-188.

Bain, D. (1962), *The Productivity Prescription: The Manager's Guide to Improving Productivity and Profits*, McGraw-Hill, NY.

Baker, F. T. (1972), Chief Programmer Team Management of Production Programming, *IBM Systems Journal*, 11(1), pp.56-73.

Basili, V.R. (1980a), *Models and Metrics for Software Management and Engineering,* IEEE Computer Society Press, Los Alamitos, CA.

Basili, V.R. (1980b), *Qualitative Software Complexity Models: A Summary in Tutorial on Models and Methods for Software Management and Engineering,* IEEE Computer Society Press, Los Alamitos, CA.

Basili, V.R. and R.W. Selby (1991), Paradigms for Experimentation and Empirical Studies in Software Engineering, *Reliability Engineering and System Safety*, 32(1-2), pp. 171-193.

Basili, V.R., R.W. Selby, and D.H. Hutchens (1986), Experimentation in Software Engineering, IEEE Transactions of Software Engineering, 12(7), pp.733-743.

Bate, R. et al. (1993), A System Engineering Capability Maturity Model, Version 1.1, *CMU/SEI-95-MM-03,* Software Engineering Institute, Pittsburgh, PA, 841993.

Bauer, F.L (1972), Software Engineering, *Information Processing*, 71.

Bauer, F.L. (1976), Software Engineering, in Ralston, A. and Meek, C. L. (eds.), *Encyclopedia of Computer Science*, Petrocelli/Charter, New York.

Bayana, S. (2006), *Learning to Deal with COTS*, ProQuest/UMI.

Bell, D.A. (1953), *Information Theory*, Pitman, London.

Berger, J. (1990), *Statistical Decision Theory – Foundations, Concepts, and Methods*. Springer-Verlag.

Bertsekas, D.P. (1995), *Dynamic Programming and Optimal Control*, Volume 1, Belmont, MA., Athena Scientific.

Beveridge, W.I. (1957), *The Art of Scientific Investigation*, Random House Trade Paperbacks, London, UK.

Bhandarhar, D. and D.W. Clark (1991), Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization, Communications of the ACM, Sept., pp.310-319

Binet, A. (1905), New Methods for the Diagnosis of the Intellectual Level of Subnormals, *L'Annee Psychologique*, 12, pp.191-244.

Bjorner, D. (2000), Pinnacles of Software Engineering: 25 Years of Formal Methods, *Annals of Software Engineering,* 10, Kluwer Academic Publishers, USA, Nov., pp.11-66.

Bjorner, D. and C.B. Jones (1982), *Formal Specification and Software Development*, Prentice Hall, Englewood Cliffs, NJ.

Boehm, B.W. (1976), Software Engineering, *IEEE Transactions on Computers,* 25(12), pp.1226-1241.

Boehm, B.W. (1981), *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ.

Boehm, B.W (1983), Seven Basic Principles of Software Development, *Journal of System and Software,* 3(1), March.

Boehm, B.W. (1984), Software Engineering Economics, *IEEE Trans. on Software Engineering*, 10(1), pp. 4-12.

Boehm, B.W., T.E. Gray, and T. Seewaldt (1984), Prototyping Versus Specifying: A Multiproject Experiment, IEEE Trans. on Software Engineering, 10(3), pp. 290-302.

Boehm, B.W. (1987), Improving Software Productivity, *IEEE Computer*, 20(9), pp.43.

Boehm, B.W. (1988), A Spiral Model for Software Development and Enhancement, *IEEE Computer*, 21(5), May, pp.61-72.

Boehm, B.W et al. (2000), *Software Cost Estimation with COCOMO II,* Prentice Hall, Englewood Cliffs, NJ.

Boehm, B. and P. Bose (1994), A Collaborative Spiral Software process Model based on Theory W*, Proc. 3rd International Conference on the Software Process,* IEEE Computer Society Press, Reston, VA, October, pp.59-68.

Booch, G. (1986), Object-Oriented Development*, IEEE Transactions on Software Engineering,* IEEE Computer Society Press, 12(2).

Boucher, A. and R. Gerth (1987), A Timed Model for Extended Communicating Sequential Processes, *Proc. ICALP'87*, Springer LNCS 267.

Boulding, K. (1974), Economics and General Systems, *Int. J. Gen. Sys.,* 1(1), pp. 67-73.

Boulding, K. (1956), General Systems Theory - The Skeleton of Science, *General Systems Yearbook*, 1, pp. 11-17.

Bourque, P. and A. Abran (1996), An Experimental Framework for Software Engineering Research, Proc. Forum on Software Engineering Standards Issues (SES'96), IEEE SESC, Montreal, Canada, Oct.

Bowen, J.P., A. Fett, and M.G. Hinchey *eds.* (1998), *Proc. the Z Formal Specification Notation,* Lecture Notes in Computer Science, Vol. 1493, Springer-Verlag, Berlin.

Brillouin, L. (1953), Negentropy Principle of Information, *J. of Applied Physics*, 24(9), pp. 1152-1163.

Brinch, H.P. (1973), *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ.

Brinch-Hansen, P. (1971), Short-Term Scheduling in Multiprogramming Systems, *Proc. the Third ACM Symposium on Operating Systems Principles*, Oct., pp.103-105.

Bronson, R. and G. Naadimuthu (1997), *Schaum's Outline of Theory and Problems of Operations Research*, 2nd ed., McGraw-Hill, NY.

Brooks, R.A. (1970), New Approaches to Robotics, *American Elsevier*, 5, NY, pp. 3-23.

Brooks, F.P. Jr. (1975), *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley Longman, Inc., Boston.

Brooks, F.P. Jr. (1987), *No Silver Bullet: Essence and Accidents of Software Engineering,* IEEE Computer, 20(4), pp. 10-19.

Brooks, F.P. Jr. (1995), *The Mythical Man-Month: Essays on Software Engineering,* Anniversary ed., Addison-Wesley Longman, Inc., Boston.

Brookes, S.D., C.A.R. Hoare, and A.W. Roscoe (1984), A Theory of Communicating Sequential Processes, *Journal of ACM*, 31(7), pp.560-599.

Broy, M. and E. Denert *eds.* (2002), *Software Pioneers,* Springer, Berlin.

Broy, M., C. Pair, and M. Wirsing (1984), A Systematic Study of Models of Abstract Data Types, *Theoretical Computer Science*, 33, pp.139-1274.

Brue, S.L. (2001), *Economics*, McGraw-Hill.

Buckland, J.A. et al. (1991)*, Total Quality Management in Information Systems,* QED Information Sciences, Wellesley, Dedham, MA.

Bunge, M. (1978), General Systems Theory Challenge to Classical Philosophy of Science, *Int. J. Gen. Sys.,* 4(1).

Bunge, M. (1981), Systems All the Way, *Nature and Systems*, 3(1), pp. 37-47.

Cannan E. ed. (1994), *The Wealth of Nations*, A. Smith (1776), Modern Library, Co., London.

Cardelli, L. (1987), Basic Polymorphic Typechecking*, Science of Computer Programming,* 8(2).

Cardelli, L. and P. Wegner (1985), On Understanding Types, Data Abstraction and Polymorphism*. ACM Computing Surveys,* 17(4), pp.471-522.

Carlsson, C. and E. Turban (2002), DSS: Directions for the next decade, Decision Support Systems, *33*, pp. 105-110.

Casti J.L. and A. Karlqvist eds. (1986), *Complexity, Language, and Life: Mathematical Approaches*, International Institute for Applied Systems Analysis, Laxenburg, Austria.

CCITT (1988), Recommendation Z.100 – *Specification and Description Language SDL,* Blue Book, Vol.20 – 24, ITU, Geneva.

Chaffin, D.B. and G.B. Andersson (1984), *Occupational Biomechanics*, Wiley & Sons Co.

Chaitin, G.J. (1977), Algorithmic Information Theory, *IBM J. Res. Develop.*, 21(4), pp. 350-359.

Chaitin, G.J. (2004), *Algorithmic Information Theory*, Cambridge Univ. Press, UK.

Chan, C., W. Kinsner, Y, Wang, and D. Miller eds. (2004), *Proc. the Third IEEE International Conference on Cognitive Informatics* (ICCI'04), Victoria, Canada, August, IEEE Computer Society Press, Los Alamitos, CA.

Chiew, V. and Y. Wang (2002), Software Engineering Process Benchmarking*, Proc. 4th International Conference on Product Focused Software Process Improvement* (PROFES'02), Springer LNCS 2559, Rovaniemi, Finland, Dec., pp.519-531.

Chiew, V. and Y. Wang (2004), Formal Description of the Cognitive Process of Problem Solving, *Proc. 3rd IEEE International Conference on Cognitive Informatics* (ICCI'04), IEEE CS Press, Canada, August, pp. 74-83.

Chomsky, N. (1956), Three Models for the Description of Languages, *I.R.E. Transactions on Information Theory*, 2(3), pp.113-124.

Chomsky, N. (1957), *Syntactic Structures*, Mouton, The Hague.

Chomsky, N. (1959), On Certain Formal Properties of Grammars, *Information and Control*, 2, pp.137-167.

Chomsky, N. (1962), Context-Free Grammar and Pushdown Storage, *Quarterly Progress Report*, MIT Research Laboratory, 65, pp.187-194.

Chomsky, N. (1965), *Aspects of the Theory of Syntax*, MIT Press, Cambridge, MA.

Chomsky, N. (1982), *Some Concepts and Consequences of the Theory of Government and Binding*, MIT Press, Cambridge, MA.

Chorafas, D.N. (1998), *Agent Technology Handbook*, McGraw-Hill, NY.

Christensen, L.B. (1997), *Experimental Methodology*, 7th ed., Allyn and Bacon, Needham Heights, MA.

Coleman, J.S. (1990), *Fundamentals of Social Theory*, The Belknap Press of Harvard Univ Press, Cambridge, MA.

Comer, D.E. (2000), *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture*, 4th ed., Prentice Hall International, Inc.

Comer, D.E. and D.L. Stevens (1996), *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Application*, Prentice Hall International, Inc.

Corsetti E., A. Montanari, and E. Ratto (1991), Dealing with Different Time Granularities in Formal Specifications of Real-Time Systems, *The Journal of Real-Time Systems*, 3(2), Kluwer Academic Publishers, June, pp.191-215.

Cries, D. (1981), *The Science of Programming*. Springer-Verlag, New York.

Crosby, P.B. (1977), *Quality is Free: The Art of Making Quality Certain*, McGraw-Hill, NY.

Curtis, B., H. Krasner, V.Y. Shen, and N. Iscoe (1987), On Building Software Process Models under the Lamppost, *Proc. 9th International Conference on Software Engineering,* IEEE Computer Society Press, Monterey, CA., pp. 96-103.

Cutnell, J.D. and K.W. Johnson (1998), *Physics*, 4th ed., John Wiley & Sons, Inc., NY.

Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare (1972), *Structured Programming*, Academic Press, NY.

Dale, C. J. and Zee, H. (1992), Software Productivity Metrics: Who Needs Them? *Information and Software Technology,* 34(11), 731-738.

David, J.C.M. (2002), Information Theory, Inference and Learning Algorithms, Cambridge Univ. Press, UK.

Davis, A.M. (1994), Fifteen Principles of Software Engineering, *IEEE Software*, Nov., pp.94-96.

Dawson, M.R.W. (1998), *Understanding Cognitive Science,* Blackwell Publishing Ltd., Oxford, UK.

Day, J.D. and H. Zimmermann (1983), *The OSI Reference Model,* Proc. of the IEEE, 71, Dec., pp. 1334-1340.

Debenham, J.K. (1989), *Knowledge Systems Design,* Prentice Hall, NY.

Deming, W.E. (1982),*Quality, Productivity and Competitive Position*, MIT Press, Center for Advanced Engineering Study, Cambridge, MA.

Deming, W.E. (1986), *Out of the Crisis*, MIT Press, Cambridge, MA.

Derrick, J. and E. Boiten (2001), *Refinement in Z and Object-Z: Foundations and Advanced Applications*, Springer-Verlag, London.

Descartes, R. (1979), *Meditations on First Philosophy*, D. Cress trans., Indianapolis: Hackett Publishing Co. Inc.

Dierks, H. (2000), *A Process Algebra for Real-Time Programs*, LNCS Vol. 1783, Springer, Berlin, pp. 66-76.

Dijkstra, E.W. (1965), Programming Considered as a Human Activity, in W. A. Kalenich (ed.), *Proc. IFIP Congress 65*, Spartan Books, Washington, D.C.

Dijkstra, E.W. (1968a), The GOTO Statement Considered Harmful, *Communications of the ACM*, 11(3), March, pp. 147-148.

Dijkstra, E.W. (1968b), The Structure of the THE Multiprogramming System, *Communications of the ACM*, May, 11(5), pp. 341-346.

Dijkstra, E.W. (1972), The Humble Programmer, 1972 Turing Award Lecture, *Communications of the ACM*, 15(10), pp.859-866.

Dijkstra, E.W. (1975), Guarded Commands, Nondeterminacy, and the Formal Derivation of Programs, *Communications of the ACM*, 18(8), pp.453-457.

Dijkstra, E.W. (1976), *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ.

Donnelly, J.H. Jr., J.L. Gibson, and J.M. Ivancevich (1998), *Fundamentals of Management*, 10th ed., McGraw-Hill Co. Inc., Boston.

Doren, C.V. (1992), *A History of Knowledge: Past, Present, and Future*, Ballantine Books, NY.

Dorling, A., Y. Wang et al. (1999), Reference Model Extensions to ISO/IEC TR 15504-2 for Acquirer Processes, *ISO/IEC JTC1/SC7/WG10*, Curitiba, Brazil, May, pp. 1-34.

Dougherty, D.M. and D.B. Stephens (1984), The Lasting Quality of PERT, *R&D Management*, Jan., pp.47-56.

Dromey, G. (1995), A Model for Software Product Quality, *IEEE Trans. on Software Engineering*, 21(2), pp. 146-162.

DTI (1987), *The TickIT Guide,* Department of Trade and Industry, London.

Dunn, M. and G. Epstein ed. (1977), *Modern Uses of Multiple-Valued Logic*, Springer, Berline.

Dunn R.H. and R.S. Ullman (1994), *TQM for Computer Software, 2nd ed.,* McGraw-Hill, Inc., NY.

Dupuis, R., P. Bourque, A. Abran, S. Wolff, and J. Moore (1999), *Progress Report on the Fundamental Principles of Software Engineering,* 1999, IEEE SESC.

Dutta, S., S. Kulandaiswamy, and L.V. Wassenhove (1998), Benchmarking European Software Management Best Practices, *Comm. ACM*, 41(6), June, pp.77-86.

Eagly, A.H. and S. Chaiken (1992), *The Psychology of Attitudes*, Harcourt Brace, San Diego.

Earhart, S.V. *ed.* (1986), *AT&T UNIX Programmer's Manual*, Holt, Rinehart, and Winston, New York, NY.

Edwards, W. and B. Fasolo (2001), Decision Technology, *Annual Review of Psychology*, 52, pp.581-606.

EFQM (1993), *Total Quality Management - the European Model for Self-Appraisal 1993,* Guidelines for Identifying and Addressing Total Quality Issues, European Foundation for Quality Management, Brussels, Belgium.

Eide, A.R., R.D. Jenison, L.H. Mashaw, and L.L. Northup (1979), *Engineering Fundamentals and Problem Solving*, McGraw-Hill Book Co., New York, NY.

Eigen, M. and P. Schuster (1979), *The Hypercycle: A Principle of Natural Self-Organization*, Springer, Berlin.

Elder, G.H. Jr. (1975), Age Differentiation and the Life Course, *Annual Review of Sociology*, 1, Palo Alto, CA.

Ellis, D. O. and J.L. Fred (1962), *Systems Philosophy*, Prentice Hall, Englewood Cliffs, NJ.

Ellis, W.D. *ed.* (1938), *A Source Book of Gestalt Psychology*, Routledge & Kegan Paul, London.

Embry, D.E. (1986), SHERPA: A Systematic Human Error Reduction and Prediction Approach, Proc. the International Topical Meeting on Advances in Human Factors in Nuclear Power Systems, Knoxville, Tennessee.

Emerson, E.A. (1990), Temporal and Modal Logic, in J. van Leeuwen ed., *Handbook of Theoretical Computer Science, Vol. B: Formal Model and Semantics*, Elsevier, pp. 995-1072.

Fabrycky, W. J., M. Ghare, and P.E. Torgersen (1984), *Applied Operations Research and Management Science*, Prentice Hall, N.J.

Fayad, M. and D. Schmidt (1997), Object-Oriented Application Frameworks. *CACM*, 40(1)0.

Fayad, M.E., D.C. Schmidt, and R.E. Johnson *ed.* (1999), *Building Application Frameworks: Object-Oriented Foundations of Frameworks Design*, John Wiley & Sons Inc., NY.

Fayol, H. (1929), *General and Industrial Management*, Translated by C. Storrs, Sir Isaac Pitman and Sons, London.

Fazio, R.H. (1986), How do Attitudes Guide Behavior? In R.M. Sorrentino and E.T. Higgins *eds.*, *The Handbook of Motivation and Cognition: Foundations of Social Behavior*, Guilford Press, NY.

Fecher, H. (2001), A Real-Time Process Algebra with Open Intervals and Maximal Progress, *Nordic Journal of Computing*, 8(3), pp.346-360.

Fenton, N.E. and Pfleeger, S.L. (1997)*, Software Metrics – A Rigorous and Practical Approach,* 2nd ed., PWS Publishing, London.

Fenton, N.E. (1991), *Software Metrics: A Rigorous Approach*, Chapman & Hall, London.

Feynman, R.P. and L.M. Brown (2000), *Selected Papers of Richard Feynman*, World Scientific Pub. Co., Singapore.

Finfer, M. (1989), Panel Report: What Makes a Good Software Engineer? *Proc. 1998 Conference on Tri-Ada*, ACM Press, NY, pp.367-370.

Fischer, K.W., P.R. Shaver, and P. Carnochan (1990), How Emotions Develop and How They Organize Development, *Cognition and Emotion*, 4, pp.81-127.

Fishbein, M. and I. Ajzen (1975), *Belief, Attitude, Intention, and Behavior: An Introduction to Theory and Research*, Addison-Wesley, Reading, MA.

Ford, J. (1986), *Chaos: Solving the Unsolvable, Predicting the Unpredictable, in Chaotic Dynamics and Fractals*, Academic Press.

Frank, R.H. (1997), *Microeconomics and Behaviors*, 3rd ed, McGraw-Hill Co. Inc., NY.

Freud, S. (1895), Project for a Scientific Psychology, in J. Strachey (1966) ed, *The Standard Edition the Complete Psychological Works of Sigmund Freud*, 1, Hogarth Press, London.

Fried, G.H. and G.J. Hademenos (1999), *Schaum's Outline of Theory and Problems of Biology*, 2nd ed., McGraw-Hill, NY.

Friedman, K. (1996), *The Decision Tree*, Heart Publishing Co.

Gabrieli, J.D.E. (1998), Cognitive Neuroscience of Human Memory, *Annual Review of Psychology*, 49, pp. 87-115.

Gaines, B.R. (1979), General Systems Research, *General Systems Yearbook*, 24, pp. 1-9.

Gaines, B.R. (1972), Axioms for Adaptive Behavior, *Int. J. of Man-Machine Studies*, 4, pp. 169-199.

Gaines, B.R. (1976), On the Complexity of Causal Models, *IEEE Trans. On Sys., Man, and Cyb*., 6, pp. 56-59.

Gaines, B.R. (1977), System Identification, Approximation and Complexity, *Int. J. Gen. Sys*., 3(145), pp. 145-174.

Gaines, B.R. (1978), Progress in General Systems Research, in G.J. Klir *ed.*, *Applied General Systems Research*, Plenum, NY, pp. 3-28.

Gaines, B.R. (1983), Precise Past - Fuzzy Future, *Int. J. Man-Machine Studies*, 19, pp. 117-134.

Gaines, B.R. (1984), Methodology in the Large: Modeling All There Is, *Systems Research*, 1(2), pp. 91-103.

Gamma E., R.Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading, MA.

Ganter, B. and R. Wille (1999), *Formal Concept Analysis*, Springer, Berlin.

Gantt, H. L. (1919), *Organizing for Work*, Harcourt Brace and Howe, NY.

Garvin, D.A. (1987), Competing in the Eight Dimensions of Quality, *Harvard Business Review*, Vol. 1987, Nov./Dec.

Garvin, D.A. (1991), How the Baldrige Award Really Works, *Harvard Business Review*, Vol. 1991, Nov./Dec., pp.80-93.

Gerber, R., E.L. Gunter, and I. Lee (1992), Implementing a Real-Time Process Algebra, in Archer, M., J.J. Joyce, K.N. Levitt and Phillip J. Windley eds., *Proc. the International Workshop on the Theorem Proving System and its Applications*, IEEE Computer Society Press, Los Alamitos, CA, USA, August, pp.144-154.

Gersting, J. L (1982), *Mathematical Structures for Computer Science,* W. H. Freeman & Co., San Francisco.

Giarrantans, J. and G. Riley (1989), *Expert Systems: Principles and Programming*, PWS-KENT Pub. Co., Boston.

Gilb, T. (1988), *Principles of Software Engineering Management,* Addison-Wesley, Reading, MA.

Gilb, T. and D. Graham (1993), *Software Inspection*, Addison-Wesley, Reading, MA.

Gisselquist, R. (1998), Engineering in Software, *Communications of the ACM,* 41(10), Oct., pp. 107-108.

Gleason, H.A. Jr. (1961), *An Introduction to Descriptive Linguistics*, Holt, Rinehart and Winston, Toronto.

Gleason, J.B. (1997), *The Development of Language, Introduction to Descriptive Linguistics*, 4th ed., Allyn and Bacon, Boston, MA.

GMOD (1992), *V-Model: Software Lifecycle Process Model, General Report No. 250,* German Ministry of Defense.

Goguen, J.A. (1978), Some Design Principles and Theory for OBJ-0, A Language for Expressing and Executing Algebraic Specifications of Programs, *Proc. of the International Conference on Mathematical Studies of Information Processing*, Kyoto, pp.425-473.

Goguen, J.A., J.W. Thatcher, E.G. Wagner, and J.B. Wright (1977), Initial Algebra Semantics and Continuous Algebras, *Journal of the ACM*, 24(1), January, pp. 68-59.

Goguen, J.A. and G. Malcolm (1996), *Algebraic Semantics of Imperative Programming*, MIT Press.

Goldberg, A. and D. Robson (1983)*, Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company.

Goldman, S. (1953), *Information Theory*, Prentice Hall, Englewood Cliffs, NJ, USA.

Gray, B. (1989), *Collaborating: Finding Common Ground for Multiparty Problems*, San Francisco, Jossey-Bass.

Gregory, M.S. (1971), *History of Development of Engineering*, Longman Group Ltd., London.

Gries, D. (1981), *The Science of Programming*, Spinger-Verlag, Berlin.

Grune, D., H.E. Bal, C.J.H. Jacobs, and K.G. Langendoen (2000), *Modern Compiler Design*, John Wiley & Sons, England.

Gunter, C.A. (1992), Semantics of Programming Languages: Structures and Techniques, in M. Garey and A. Meyer ed., *Foundations of Computing*, MIT Press.

Gunter, C.A. and J.C. Mitchell, ed. (1994), *Theoretical Aspects of Object-Oriented Programming.* MIT Press.

Gustavsson, A. (1989), Maintaining the Evaluation of Software Objects in an Integrated Environment, *Proc. 2nd International Workshop on Software Configuration Management*, ACM, Princeton, NJ, October, pp.114-117.

Guttag, J.V. (1975), *The Specification and Application to Programming of Abstract Data Types*, PhD Thesis, University of Toronto.

Guttag, J.V. (1977), Abstract Data Types and the Development of Data Structures, *Communications of the ACM*, 20(6), pp.396-404.

Guttag, J.V. and J.J. Horning (1978), The Algebraic Specification of Abstract Data Types, *Acta Informatica*, 10, pp.27-52.

Guttag, J.V. (2002), Abstract Data Types, Then and Now, in M. Broy and E. Denert eds., *Software Pioneers*, Springer, Berlin, pp.443-452.

Haase, V., R. Messmarz, G. Koch, H.J. Kugler, and P. Decrinis (1994), BOOTSTRAP: Fine-Tuning Process Assessment, *IEEE Software,* 11, July, 25-35.

Haken, H. (1977), *Synergetics*, Springer-Verlag, NY.

Haken, H. (1983), *Advanced Synergetics – Instability Hierarchies of Self-Organizing Systems and Devices*, Springer-Verlag, Berlin.

Haken, H., A. Wunderlin, S. Yigitbasi (1995), An Introduction to Synergetics, *Open Systems and Information Dynamics*, 3(1), pp. 97-130.

Hagstrom, J. N. (1988), Computational Complexity of PERT Problems, *Networks,* 18, pp.139–147.

Hall, A. D. (1967), *A Methodology for Systems Engineering*, Van Nostrand Reinhold, New York.

Hall, A.S. and R.E. Fagan (1956), Definition of System, *General Systems Yearbook*, 1, pp. 18-28.

Halstead, M.H. (1977), *Elements of Software Science,* Elsevier North – Holland, New York.

Harauz, J. (1997), Workshop Report on System Engineering Principles and Scenarios, *Summary Report of the Second International Symposium on Software Engineering Standards* (ISESS'97), IEEE SESC, Sept.

Hardy, C. and B. Latane (1986), Social Loafing on a Cheering Task, *Social Science*, 71, pp.165-172.

Hardy, C. and N. Phillips (1998), Strategies of Engagement: Lessons from the Critical Examination of Cooperation and Conflict in an Interorganizational Design, *Organization Science*, 9(2), Feb., pp.217-230.

Harnish, R.M. (2002), Minds*, Brain, Computers: An Historical Introduction to the Foundations of Cognitive Science*, Blackwell Publishers, Ltd., Oxford, UK.

Harre, R. (2002), Cognitive Science: A Philosophical Introduction, *SAGE Publishing Ltd., London, UK.*

Hartmanis, J. and R.E. Stearns (1965), On the Computational Complexity of Algorithms, *Trans. AMS,* 117, pp. 258-306.

Hartmanis, J. (1994), On Computational Complexity and the Nature of Computer Science, 1994 Turing Award Lecture, *Communications of the ACM*, Vol.37, No.10, pp.37-43.

Harvey, R. L. (1994), *Neural Network Principles*, Prentice Hall International, Englewood Cliffs, NJ.

Hastie, R. (2001), Problems for Judgment and Decision Making, *Annual Review of Psychology*, Vol. 52, pp.653-683.

Hayes, I.J. (ed.) (1987), *Specification Case Studies*, Prentice Hall, London.

Hennessy, J.L. and D.A. Patterson (1996), Computer *Architecture: A Quantitative Approach,* 2nd ed., Morgan Kaufmann, CA.

Hermes, H. (1969), *Enumerability, Decidability, Computability*, Springer-Verlag, New York.

Hester, S.D., D.L. Parnas, and D.F. Utter (1981), Using documentation as a Software Design Medium, *Bell System Technical Journal*, 60(8), Oct., pp. 1941-1977.

Higman, B. (1977), *A Comparative Study of Programming Languages*, 2nd ed., MacDonald.

Hoare, C.A.R. (1969), An Axiomatic Basis for Computer Programming, *Communications of the ACM*, 12(10), pp.576-580.

Hoare, C.A.R. (1973), Hints on Programming Language Design, *Proc. ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 31-40.

Hoare, C.A.R. (1975), Software Engineering, *Computer Bulletin*, 2(6), Dec., pp. 6-7.

Hoare, C.A.R. (1978), Communicating Sequential Processes, *Communications of the ACM*, 21(8), pp.666-677.

Hoare, C.A.R. (1980), The Emperor's Old Clothes, The 1980 Turing Award Lecture, *Communications of the ACM,* 24(2), pp.75-83.

Hoare, C.A.R. (1985), *Communicating Sequential Processes,* Prentice Hall International, Englewood Cliffs, NJ.

Hoare, C.A.R. (1986), *The Mathematics of Programming*, Clarendon Press, Oxford, UK.

Hoare, C.A.R. and Jones, C.B. (eds.) (1989), *Essays in Computing Science*, Prentice Hall, Englewood Cliffs, NJ.

Hoare, C.A.R. and N. Wirth (1966), A Contribution to the Development of ALGOL, *Communications of the ACM*, 9(6), pp.413-431.

Hoare, C.A.R., I.J. Hayes, J. He, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin (1987), Laws of Programming, *Communications of the ACM*, 30(8), August, pp. 672-686.

Hoffman, D.M. and D.M. Weiss *eds.* (2001), Software Fundamentals: Collected Papers by David L. Parnas, Addison-Wesley, Inc., Boston.

Hopcroft, J.E. and J.D. Ullman (1979), *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley Publishing Co., MA.

Horowitz, E. (1984), *Fundamentals of Programming Languages*, 2nd ed., Computer Science Press, Rockville, MD.

Horstmann, C. and T. Budd (2004), *Big C++,* John Wiley & Sons, Inc., Danvers, MA.

Huang, J. and Y. Wang (2006), Design of an Integrated Hyper Specification Documentation Tool, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 370-379.

Hull, C.L. (1943), *Principles of Behavior: An Introduction to Behavior Theory*, Oxford Univ. Press, NY.

Humphrey, W. (1996), Using a Defined and Measured Personal Software Process, *IEEE Software*, May, pp. 77-88.

Humphrey, W. S. (1995), *A Discipline for Software Engineering*, SEI Series in Software Engineering, Addison-Wesley, Reading, MA.

Humphrey, W.S. (1988), Characterizing the Software Process: A Maturity Framework, *IEEE Software*, 5(2), March, pp. 73-79.

Humphrey, W.S. (1989), *Managing the Software Process*, Addison-Wesley Longman, Reading, MA.

Hurley, P.J. (1997), *A Concise Introduction to Logic*, 6th ed., Wadsworth Pub. Co., Belmont, CA.

Huseman, R. C., and E.W. Miles (1988), Organizational Communication in the Information Age, *Journal of Management*, 14, pp.181-204.

Huxham, C. (1996), Advantage or Inertia? Making Cooperation Work, in Paton, R., Clark, G., Jones, G., Lewis, J. and Quinlan, P. (eds.), *The New Management Reader,* Routledge, London, New York.

IBM (1996), Software Development Performance and Practices in Europe: A Benchmark of Software Development in Europe – Self Assessment Questionnaire, V.2.0, *IBM Eurocoordination*, pp. 1-11.

IBM (1997), IBM European Benchmark of Software Development Practices, pp.1-240.

IBM (2001), *IBM Autonomic Computing Manifesto*, http://www.research.ibm.com/autonomic/.

IBM (2006), Autonomic Computing White Paper: *An Architectural Blueprint for Autonomic Computing*, 4th ed., June, pp. 1-37.

IBM and IVF (1997), 1997 National Benchmarking Survey of Performance and Practices in Swedish Software Producing Units, V.2.0, pp.1-15.

IEEE (1983), *Software Engineering Standards, 1983 Collection*, IEEE Computer Society Press, Los Alamitos, CA.

IEEE (1988), *Software Engineering Standards, 1988 Collection*, IEEE Computer Society Press, Los Alamitos, CA.

IEEE (1998), *Software Engineering Code of Ethics and Professional Practice* (v.5.2), Recommended by the IEEE/ACM Joint Task Force on Software Engineering Ethics and Professional Practices.

IEEE 610.12 (1991), IEEE STD 610.12 – 1990, IEEE Standard Glossary of Software Engineering Terminology, Corrected Edition, Feb.

IEEE/ACM (2001), Software Engineering Body of Knowledge (SWEBOK), V.0.95, May, pp. 1-213.

IEEE/ACM (2003), Computing Curricula – Software Engineering (CCSE), http://sites.computer.org/ccse/.

IPA (1970), *The International Phonetic Alphabet*, http://www.arts.gla.ac.uk/IPA/ipa.html.

ISO 9000-1 (1994), *Quality Management and Quality Assurance Standards (Part 1) - Guidelines for Selection and Use*, International Organization for Standardization, Geneva.

ISO 9000-2 (1994), *Quality Management and Quality Assurance Standards (Part 2) – Generic Guidelines for Application of ISO 9001, ISO 9002 and ISO 9003,* International Organization for Standardization, Geneva.

ISO 9000-3 (1991), *Quality Management and Quality Assurance Standards (Part 3) - Quality Management and Quality Assurance Standards (Part 3) - Guidelines to Apply ISO 9001 for Development, Supply and Maintenance of Software,* International Organization for Standardization, Geneva.

ISO 9000-4 (1993), *Quality Management and Quality System (Part 4) - Guidelines for Dependability Programme Management*, International Organization for Standardization, Geneva.

ISO 9001 (1989), *Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation, and Servicing*, International Organization for Standardization, Geneva.

ISO 9001 (1994), Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation, and Servicing, Revised Edition, International Organization for Standardization, Geneva.

ISO 9002 (1994). *Quality Systems - Model for Quality Assurance in Production, Installation and Servicing*, International Organization for Standardization, Geneva.

ISO 9003 (1994), *Quality Systems - Model for Quality Assurance in Final Inspection and Test*, International Organization for Standardization, Geneva.

ISO 9004-1 (1994), *Quality Management and Quality System Elements (Part 1) – Guidelines,* International Organization for Standardization, Geneva.

ISO 9004-2 (1991), *Quality Management and Quality System Elements (Part 4) - Guidelines for Quality Management and Quality Systems Elements for Services*, International Organization for Standardization, Geneva.

ISO 9004-4 (1993), *Quality Management and Quality System Elements (Part 2) - Guidelines for Quality Improvement,* International Organization for Standardization, Geneva.

ISO 9126 (1991), *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use*, International Organization for Standardization, Geneva.

ISO/IEC 12207 (1995), *Information Technology – Software Life Cycle Processes,* International Organization for Standardization, Geneva.

ISO/IEC 15504 (2000), *Software Process Assessment - Parts 1 ~ 9,* ISO/IEC, Geneva.

ISO/IEC 15288 (1999), *Information Technology – Life Cycle Management – System Life Cycle Processes*, ISO/IEC JTC1/SC7 N2184, Geneva, pp.1-42.

Jackson, J.M. and S.G. Harkins (1985), Equity in Effort: An Explanation of the Social Loafing Effect, *Journal of Personality and Social Psychology,* 49, pp.1199-1206.

James, W. (1890), *Principles of Psychology*, New York, Holt.

James, W.M. (1998), *Software Engineering Standards: A User's Road Map,* IEEE Computer Society Press, Los Alamitos, CA.

Janis, I. (1971), Groupthink, *Psychology Today*, Nov., pp.43-46.

Jeffrey, A. (1992), Translating Timed Process Algebra into Prioritized Process Algebra, in J. Vytopil ed., *Proc. the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS Vol. 571, Springer-Verlag, Nijmegen, The Netherlands, pp.493-506.

Jenner, M.J. (1995), *Software Quality Management and ISO 9001*, John Wiley & Sons, Inc., New York.

Jennings, N.R. (2000), On Agent-Based Software Engineering, *Artificial Intelligence*, 177(2), pp. 277-296.

Jensen, J. (1990), *Morphology: Word Structure in Generative Grammar,* John Benjamins, Amsterdam.

Jensen, K. (1978), *Pascal User Manual and Report*, 2nd ed., Springer Verlag.

Johansson, H.J., and P. Mchugh (1993), *Business Process Reengineering*, John Wiley & Sons, Engl.

Johnson, S.C. (1975), Yacc – Yet Another Compiler Compiler, AT&T Bell Laboratories, *Computing Science Technical Report No.32*, AT&T Bell Labs., Murray Hill, NJ.

Jones, C. (1981), *Programming Productivity* – Issues for the Eighties, IEEE Press, Silver Spring, MD.

Jones, C. (1986), *Programming Productivity*, McGraw-Hill Book Co., NY.

Jones, C. (1996), *Patterns of Software System Failure and Success*, International Thomson Computer Press, Boston, MA.

Jones, C.B. (1980), *Software Development: A Rigorous Approach*, Prentice-Hall International, London.

Juran, J. M. (1988), *Juran on Planning for Quality*, Macmillan, New York.

Juran, J.M. (1989), *Juran on Leadership for Quality,* The Free Press, New York.

Juran, J.M., L.A. Seder, and F.M. Gryna eds. (1962)*, Quality Control Handbook,* 2nd ed., McGraw-Hill, New York.

Juran, J.M. and F.M. Gryna (1980), *Quality Planning and Analysis*, McGraw-Hill, New York.

Kandel, E.R., J.H. Schwartz, and T.M. Jessell *eds.* (2000), *Principles of Neural Science,* 4th ed., McGraw-Hill, New York.

Kant, I. (1956), *Ethics Founded on Reason*, in Groundwork of the Metaphysics of Morals, London, Unwin Hyman Ltd.

Kephart, J. and D. Chess (2003), The Vision of Autonomic Computing, IEEE *Computer*, 26(1), Jan, pp. 41-50.

Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Gary, and M.A. Adler (1986), *Software Complexity Measurement*, Chapter 28, ACM Press, New York, pp. 1044 -1050.

Keen, P.G. and M.S. Morton (1978)*, Decision Support Systems: An Organizational Perspective,* Addison-Wesley, Reading, MA.

Kelley, J.E. (1961), Critical-Path Planning and Scheduling: Mathematical Basis. *Operations Research,* 9, pp. 296–320.

Kemere, C.F. (1998), Progress, Obstacles, and Opportunities in Software Engineering Economics, *Communications of the ACM,* August, 41(8).

Kephart, J. and D. Chess (2003), The Vision of Autonomic Computing, *IEEE Computer*, 26(1), Jan, pp. 41-50.

Kerr, N.L. (1983), Motivation Losses in Small Groups: A Social Dilemma Analysis, *Journal of Personality and Social Psychology*, 45, pp.819 – 828.

Khaden, R. and A. Schultzki (1983), Planning and Forecasting Using a Corporate Model, *Managerial Planning*, Jan./Feb.

Kinsner, W., D. Zhang, Y. Wang, and J. Tsai *eds.* (2005), *Proc. the Fourth IEEE International Conference on Cognitive Informatics* (ICCI'05), Irvine, California, USA, IEEE Computer Society Press, Los Alamitos, CA., July.

Kinsner, W. (2007a), Towards Cognitive Machines: Multiscale Measures and Analysis, *The International Journal on Cognitive Informatics and Natural Intelligence* (IJCINI), 1(1), pp. 28-38.

Kinsner, W. (2007b), Is Entropy Suitable to Characterize Data and Signals for Cognitive Informatics? *The International Journal on Cognitive Informatics and Natural Intelligence* (IJCINI), 1(2), pp. 34-57.

Kleene, S.C. (1952), *Introduction to Metamathematics*, North Holland, Amsterdam.

Kleene, S.C. (1956), Representation of Events by Nerve Nets, in C.E. Shannon and J. McCarthy eds, *Automata Studies*, Princeton Univ. Press, pp. 3-42.

Klir, G.J. *ed.* (1972), *Trends in General Systems Theory*, John Wiley, New York.

Klir, G.L. (2001), *Facets of Systems Science*, 2nd ed., Kluwer Academic/Plenum Publishers, New York.

Klir, R.G. (1988), Systems Profile: the Emergence of Systems Science, *Systems Research*, 5(2), pp. 145-156.

Klusener, A.S. (1992), Abstraction in Real Time Process Algebra, in J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg eds., *Proc. Real-Time: Theory in Practice*, LNCS, Springer, Berlin, pp. 325-352.

Knuth, D.E. (1965), On the Translation of Languages from Left to Right, *Information and Control*, 8, pp.607-639.

Knuth, D.E. (1974), Structured Programming with GOTO Statements, *ACM Computing Surveys*, 6(4), December, pp. 261 - 302.

Knuth, D.E. (1984), Literate Programming, *The Computer Journal,* 27(2), pp.97-111.

Koch, G.R. (1993), Process Assessment: The 'BOOTSTRAP' Approach, *Information and Software Technology*, 35(6/7), Butterworth-Heinemann Ltd., Oxford, June/July, pp.387-403.

Kolb, D.A. and A.L. Frohman (1970), An Organization Development Approach to Consulting, *Sloan Management Review*, 12(1), Fall, pp.51-65.

Kolmogorov, A.N. (1965), Three Approaches to the Quantitative Definition of Information, *Problems of Information Transmission*, 1(1), pp. 1-7.

Komorita, S.S. and J.M. Barth (1985), Components of Reward in Social Bargaining, *Journal of Personality and Social Psychology*, 48, pp. 364-373.

Kotulak, R. (1997), *Inside the Brain*, Andrews McMeel Publishing Co., Kansas City, MO.

Kramosil, I. (2001), *Probabilitic Analysis of Belief Functions*, Kluwer Academic/Plenum Publishers, NY.

Kravits, D.A. and B. Martin (1986), Ringelmann Rediscovered: The Original Article, *Journal of Personality and Social Psychology*, 50, pp. 936-941.

Krohn, K.B. and Rhodes, J.L. (1963), Algebraic Theory of Machines, in J. Fox *ed.*, *Mathematical Theory of Automata*, Polytechnic Press, Brooklyn, NY, pp. 341-384.

Kuhn, T. (1970), *The Structure of Scientific Revolutions,* The Univ. of Chicago, Chicago.

Kuvaja, P. and A. Bicego (1994b), BOOTSTRAP – A European Assessment Methodology, *Software Quality Journal*, June.

Kyburg, H.E. (1984), *Theory and Measurement*, Cambridge University Press, Cambridge, UK.

Labrosse, J.J. (1999), *MicroC/OS-II, The Real-Time Kernel*, 2nd ed., R&D Books, Gilroy, CA, December.

Laplante, P.A. (1977), *Real-Time Systems Design and Analysis*, 2nd ed., IEEE Press.

Latane, B., K.D. Williams, and S.G. Harkins (1979), Many Hands Make Light the Work: The Cause and Consequences of Social Loafing, *Journal of Personality and Social Psychology*, 37, pp.822-832.

Lawrence Jr., J.A. and B. Pasternack (2002), *Applied Management Science: A Computer Integrated Approach for Decision Making*, 2nd ed., Wiley.

Leahey, T.H. (1980), *A History of Psychology: Main Currents in Psychological Thoughts*, 4th ed., Prentice Hall, Upper Saddle River, N.J.

Lehman, M.M. (1985), *Program Evolution: Processes of Software Change*, Academic Press, London.

Lesk, M.E. (1975), Lex – A Lexical Analyzer Generator, AT&T Bell Laboratories, *Computing Science Technical Report No.39*, Murray Hill, NJ.

Leveson, N.G. (1995), *Software: System Safety and Computers*, Addison Wesley.

Leveson, N.G. (1997), Software Engineering: A Look Back and a Path to the Future, *Communications of the ACM*, Feb., pp. 1-5.

Lewin, Z.K. (1948), *Resolving Social Conflicts: Selected Papers on Group Dynamics*, in G.W. Lewin ed., Harper and Row, NY.

Lewis, B. and D. Berg (1998), *Multithreaded Programming with Threads*, Sun Microsystems Press, Upper Saddle River, NJ.

Lewis, H.R. and Papadimitriou, C.H. (1998), *Elements of the Theory of Computation,* 2nd ed., Prentice Hall International, Englewood Cliffs, NJ.

Linton, R. (1936), *The Study of Man*, Appleton Century Crofts, NY.

Lipschutz, S. (1964), *Schaum's Outline of Theory and Problems of Set Theory and Related Topics*, McGraw-Hill, Inc., New York.

Lipschutz, S. and M. Lipson (1997), *Schaum's Outline of Theories and Problems of Discrete Mathematics*, 2nd ed., McGraw-Hill Inc., New York, NY.

Liskov, B. and S. Zilles (1974), Programming with Abstract Data Types, *ACM SIGPLAN Notices*, 9, pp.50-59.

Little, J.D.C. (1961), A Proof of the Queuing Formula L=λW, *Operations Research*, 9.

Liu, J. (2000), *Real-Time Systems*, Prentice Hall, Upper Saddle River, NJ.

Livermore, J. (2005), *Measuring Programmer Productivity*, http://home.sprynet.com/~jgarriso/jlpaper.htm.

Locke, J. and J.A. St. John (1902), *The Philosophical Works of John Locke.*

Louden K.C. (1993), *Programming Languages: Principles and Practice*, PWS-Kent Publishing Co., Boston.

Macionis, J.J., J.N. Clarke, and L.M. Gerber (1997), *Sociology,* 2nd ed., Prentice Hall Allyn and Bacon Canada, ON.

Mackintosh, N.J. (1998), *IQ and Human Intelligence*, Oxford University Press, USA.

MaCulloch, W.S. and Pitts, W. (1943), A Logic Calculus of the Ideas Imminent in Nervous Activity, *Bull. Math. Biophysics*, 5, pp.115-133.

Makridakis, S. and E.R. Weinstraub (1971), On the Synthesis of General Systems, *General Systems Yearbook*, 16, pp. 43-54.

Makridakis, S. and C. Faucheux (1973), Stability Properties of General Systems, *General Systems Yearbook*, 18, pp. 3-12.

Mandrioli, D. and C. Ghezzi (1987), *Theoretical Foundations of Computer Science*, John Wiley & Sons, New York.

Maquet, P. (2001), The Role of Sleep in Learning and Memory, *Science*, 294(5544), November, pp. 1048 – 1051.

Marcotty, M. and H. Ledgard (1986), *Programming Language Landscape*, 2nd ed., SRA, Chicago.

Marieb, E.N. (1992), *Human Anatomy and Physiology*, 2nd ed., The Benjamin/Cummings Publishing Co., Inc., Redwood City, CA.

Marshall, A. (1938), *Principles of Economics*, The Macmillan Co., London.

Marshall, T. (1989), Worth the RISC, *Byte*, 14(2), pp. 245.

Martin-Lof, P. (1975), An Intuitionistic Theory of Types: Predicative Part, in H. Rose and J. C. Shepherdson eds., *Logic Colloquium 1973*, NorthHolland.

Maslow, A.H. (1962), *Towards a Psychology of Being*, Van Nostrand, Princeton, NJ.

Maslow, A.H. (1970), *Motivation and Personality*, 2nd ed, Harper & Row, NY.

Matlin, M.W. (1998), *Cognition*, 4th ed., Harcourt Brace College Publishers, Orlando, FL.

McCabe, T.H. (1976), A Cyclomatic Complexity Measure, *IEEE Trans. Software Engineering,* 2(6), pp.308-320.

McCarthy, J.L. (1987), Generality in Artificial Intelligence, The 1971 Turing Award Lecture, *Communications of the ACM*, 30(12), pp. 1029-1035.

McConnell, S. (1999), Software Engineering Principles, *IEEE Software*, 16(2), March/April, pp.1-4.

McConnell, S. (2000), Ten Best Influences on Software Engineering, *IEEE Software,* 17(1), Jan./Feb.

McCue, G.M. (1978), IBM's Santa Teresa Laboratory: Architectural Design for Program Development, *IBM Systems Journal*, 17(1), pp. 4-25.

McCulloch, W.S. and W.H. Pitts (1943), A Logic Calculus of the Ideas Immanent in Nervous Activity, *Bulletin of Mathematical Biophysics*, USA, Vol. 5.

McCulloch, W.S. (1965), *Embodiments of Mind*, MIT Press, Cambridge, MA.

McCulloch, W.S. (1993), *The Complete Works of Warren S. McCulloch*, Intersystems Pub., Salinas, CA.

McDermid, J.A. *ed.* (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann Ltd., Oxford, UK.

Meek, B.L. (1991), Early High-Level Languages, Chapter 43, in J. McDermid ed., *Software Engineer's Reference Book*, Butterworth Heinemann Ltd., Oxford, UK.

Melton, A. *ed.* (1996), *Software Measurement*, Int. Thomson Computer Press, London,

Meyer, B. (1990), *Introduction to the Theory of Programming Languages*, Prentice Hall, Englewood Cliffs, NJ.

Meystel, A.M. and J.S. Albus (2002), *Intelligent Systems, Architecture, Design, and Control,* John Wiley & Sons, Inc.

Milenkovic, M. (1992), *Operating Systems: Concepts and Design*, 2nd ed., McGraw-Hill, New York.

Mill, J.S. (1843), *A System of Logic,* University Press of the Pacific, Honolulu.

Mill, J.S. (1861), *Defence of Utilitarianism*, in Utilitarianism.

Miller, G.A. (1956), The Magical Number Seven, Plus or Minus two: Some Limits of Our Capacity for Processing Information, *Psychological Review*, 63, pp. 81-97.

Mills, H.D. (1975), The New Math of Computer Programming, *Communications of the ACM*, 18(1), Jan., pp. 43-48.

Mills, H.D., M. Dyer, and R.C. Linger (1987), Cleanroom Software Engineering, *IEEE Software*, 4(5), Sept., pp.19-25.

Mills, H.D., D. O'Neill, R.C. Linger, M. Dyer, and R.E. Quinnan (1980), The Management of Software Engineering, *IBM System Journal*, 24(2), pp.414-477.

Milner, R. (1980), *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag.

Milner, R. (1989), *Communication and Concurrency*, Prentice Hall, Englewood Cliffs, NJ.

Mitchell, J.C. (1990), Type systems for programming languages. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed. North Holland, pp.365-458.

Mitchell, J.C. (1996), *Foundations for programming languages*, MIT Press.

Mooney, J.D. (1947), *The Principles of Organization*, Harper and Row, NY.

Murch, R. (2004), *Autonomic Computing*, Person Education, London.

Murty, K. (1983), *Linear Programming*. John Wiley & Sons, New York.

Myerson, R.B. (1997), *Game Theory: Analysis of Conflict*, Harvard Univ. Press.

NASA (2000), http://shemehs.larc.nasa.gov/Gifs/humor-cat.gif .

Naur, P. *ed.* (1963), Revised Report on the Algorithmic Language Algol 60, *Communications of the ACM*, 6(1), pp.1-17.

Naur, P. and B. Randell (eds.) (1969)*, Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO.

Naur, P. (1978), The European Side of the Last Phase of the Development of Algol, *ACM SIGPLAN Notices*, 13, pp.15-44.

Negoita, C.V. (1989), Review: Fuzzy Sets, Uncertainty, and Information, *Cybernetes*, 18(1), pp. 73-74.

Ngolah, C.F., Y. Wang, and X. Tan (2004), The Real-Time Task Scheduling Algorithm of RTOS+, *IEEE Canadian Journal of Electrical and Computer Engineering*, 29(4), pp. 237-243.

Ngolah, F.C. and Y. Wang (2005), An Operational Semantics of RTPA, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1810-1813.

Ngolah, F.C., Y. Wang and X. Tan (2005), An RTPA Supporting Environment for Java Code Generation, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 717-720.

Ngolah, C.F., Y. Wang, and X. Tan (2006), Implementing the Real-Time Processes of RTPA Using Real-Time Java, *Proc. 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, Ottawa, ON, Canada, May 8 -10, pp. 1609-1612.

Nicollin, X. and J. Sifakis (1991), An Overview and Synthesis on Timed Process Algebras, *Proc. 3rd International Computer Aided Verification Conference*, pp. 376-398.

Nielson, M.A. and I.L. Chuang (2000), *Quantum Computing and Quantum Information*, Cambridge Univ. Press, UK.

Nordstrom, B., K. Petersson, and J.M. Smith (1990), *Programming in Martin-Lof's Type Theory*, Oxford Science Publications.

O'Grady, W. and J. Archibald (2000), *Contemporary Linguistic Analysis: An Introduction*, 4th ed., Pearson Education Canada Inc., Toronto.

Ollongren, A. (1974), *Definition of Programming Languages by Interpreting Automata*, Academic Press, New York.

OMG (2002), *IDL Syntax and Semantics*, July, pp. 1-74.

OMG (2005), *Unified Modeling Language (UML): Superstructure*, Object Management Group, version 2.0, August.

Osborne, M. and A. Rubinstein (1994), *A Course in Game Theory*, MIT Press.

Osgood, C. (1953), *Method and Theory in Experimental Psychology*, Oxford Univ. Press, UK.

Pagan, F.G. (1981), *Semantics of Programming Languages: A Panoramic Primer*, Prentice Hall, Englewood Cliffs, NJ.

Park, C.S., R. Pelot, K.C. Porteous, and M.J. Zuo (2001), *Contemporary Engineering Economics*, 2nd ed., Addison Wesley Longman, Toronto, ON, Canada.

Park, R.E. (1922), *The Immigrant Press and Its Control*, Harper, NY.

Parkinson, C.N. (1957), *Parkinson's Law and Other Studies in Administration,* Ballantine Books, NY.

Parnas, D.L. (1971), Information Distribution Aspects of Design Methodology, *Proc. 1971 IFIP Congress*, Booklet TA-3, pp.26-30.

Parnas, D.L. (1972), On the Criteria to be Used in Decomposing Systems into Modules, *Communications of ACM,* 15(12), Dec., pp.1053-1058.

Parnas, D.L., J.E. Shore, and D. Weiss (1976), Abstract Types Defined as Classes of Variables, *Proc. of Conference on Data Abstraction, Definition, and Structure*, Salt Lake City, March, pp.22-24.

Parnas, D.L. (1978), Some Software Engineering Principles, *Infotech State of the Art Report on Structured Analysis and Design,* Infotech International, pp.1-10.

Parnas, D.L. (1979), Design Software for Ease of Extension and Contraction, *IEEE Trans. Software Engineering*, 5(3), March, pp. 128-138.

Parnas, D.L. and P.C. Clements (1986), A Rational Design Process: How and Why to Fake It, *IEEE Trans. on Software Engineering*, 12(2), pp. 251-257.

Parnas, D.L. (1994a), Professional Responsibilities of Software Engineering, *Proc. IFIP World Congress 1994,* Vol.II, August, pp.332-339.

Parnas, D.L. (1994b), Software Aging, *Proc. 16th International Conference on Software Engineering*, Sorento, Italy, May, pp.279-287.

Parnas, D.L. (1995), On ICSE's "Most Influential Papers", *ACM Software Engineering Notes,* 20(3), July, pp.29-32.

Parnas, D.L. (1996), Teaching Programming as if it were Engineering, in C. Neville Dean and M.G. Hinchey eds., *Teaching and Learning Formal Methods,* Academic Press, pp.43-55.

Parnas, D.L. (1997), Software Engineering: An Unconsummated Marriage, *Communications of the ACM*, 40(9), Sept., pp. 128.

Parnas, D.L. (1998), Software Engineering Programs are not Computer Science Programs, *Annals of Software Engineering*, 6, pp. 19-37.

Parr, T. (2000), *ANTLR Reference Manual*, http://www.antlr.org/.

Pasquero, J. (1991), Supraorganizational Cooperation: The Canadian Environmental Experiment, *Journal of Applied Behavioral Science*, 27(2), pp.38-64.

Patel, S., D. Patel, and Y. Wang *eds.* (2003), *Proc. 2nd IEEE International Conference on Cognitive Informatics* (ICCI'03), IEEE Computer Society Press, July, 227pp.

Pattee, H.H. (1978), Complementarity Principle in Biological and Social Structures, *Journal of Social and Biological Structures*, 1.

Pattee, H.H. (1986), Universal Principles of Measurement and Language Functions in Evolving Systems, in J.L. Casti and A. Karlqvist eds. (1986), *Complexity, Language, and Life: Mathematical Approaches*, Springer-Verlag, Berlin, pp. 268-281

Paulk, M.C., B. Curtis, M.B. Chrissis et al. (1991), Capability Maturity Model for Software, Version 1.0, Software Engineering Institute, *CMU/SEI-91-TR-24,* August.

Paulk, M.C., B. Curtis, M.B. Chrissis, and C.V. Weber (1993), Capability Maturity Model, Version 1.1, *IEEE Software*, 10(4), July, pp.18-27.

Paulk, M.C., C.V. Weber, and B. Curtis (1995), *The Capability Maturity Model: Guidelines for Improving the Software Process*, SEI Series in Software Engineering, Addison-Wesley, Reading, MA.

Payne, D.G. and M.J. Wenger (1998), *Cognitive Psychology,* Houghton Mifflin Co., Boston, NY.

Pedrycz, W. (1981), On Approach to the Analysis of Fuzzy Systems, *Int. J. of Control*, 34, pp. 403-421.

Perry, D., N. Staudenmayer, and L. Votta (1994), Finding Out What Goes on in a Software Development Organization, Special Issue on Measurement Based Process Improvement, *IEEE Software*, 11(4), July.

Pescovitz, D. (2002), Autonomic Computing: Helping Computers Help Themselves, *IEEE Spectrum*, 39(9), pp.49–53.

Peter, L.J. and R. Hull (1969), *The Peter Principle: Why Things Always Go Wrong?* William Morrow, NY.

Peter, R. (1967), *Recursive Functions*, Academic Press, New York.

Peters, J.F. and W. Pedrycz (2000), *Software Engineering: An Engineering Approach*, John Wiley & Sons, Inc., NY.

Peterson, J.L. and A. Silberschantz (1985), *Operating System Concepts*. Addison-Wesley, Reading, MA.

Pfleeger, S.L. (1998), *Software Engineering: Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ.

Pierce, B.C. (2002), *Types and Programming Languages*, MIT Press.

Pinel, J.P.J. (1997), Biopsychology, 3rd ed., Allyn and Bacon, Needham Heights, MA.

Pinneau, S.R. (1961), *Changes in Intelligence Quotient: Infancy to Maturity*, Houghton Mifflin, Boston.

Planck, M. (1936), *Philosophy of Physics*, WW Norton & Co., Inc.

Plato (1961), *Knowledge and True Belief*, F.M. Cornford trans., The Theattetus, England, Champman & Hall Ltd.

Plato (1975), *Critique of the Divine Command Theory*, G.M.A. Grube trans., *The Euthyphro, in the Trial and Death of Socrates, Indianapolis*, Hackett Publishing Co.

Pressman, R.S. (1992), *Software Engineering: A Practitioner's Approach,* 3rd ed., McGraw-Hill International Editions, New York.

Prigogine, I. and I. Stengers (1984), *Order Out of Chaos: Man's New Dialog with Nature*, Bantam Books.

Prigogine, I. and I. Stengers (1997), *The End of Certainty, Time, Chaos, the New Laws of Nature,* Free Press, 1997.

Pnueli, A. (1977), The Temporal Logic of Programs, Proc. 18th IEEE Symposium on Foundations of Computer Science, IEEE, pp.46-57.

Qian, X.S., J.Y. Yu, and R.W. Dai (1990), A New Scientific Field – Open Complex Giant Systems and Methodologies, *Nature*, 13(1), pp. 3-11.

Quatrani, T. (1999), *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley Professional.

Rabin, M.O. and D. Scott (1959), Finite Automata and their Decision Problems, *IBM Journal of Research and Development,* 3, pp. 114-125.

Radnor, M. et al. (1970), Implementation in Operations Research and R&D in Government and Business Organization, *Operations Research*, 18(6), pp.976-991, Nov./Dec.

Ramaswami, V. and P.E. Wirth (1997), *Teletraffic Contributions for the Information Age*, Elsevier Science.

Rapoport, A. (1962), Mathematical Aspects of General Systems Theory, *General Systems Yearbook*, 11, pp. 3-11

Reason, J. (1987), Generic Error-Modeling System (GEMS): A Cognitive Framework for Locating Common Human Error Forms, in J. Rasmussen et al. eds., *New Technology and Human Error*, Wiley, NY.

Reason, J. (1990), *Human Error*, Cambridge Univ. Press, Cambridge, UK.

Reed, G.M. and A.W. Roscoe (1986), A Timed model for Communicating Sequential Processes, *Proc. ICALP'86*, LNCS 226, Springer-Verlag, Berlin.

Reisberg, D. (2001), *Cognition: Exploring the Science of the Mind*, 2nd ed., W.W. Norton & Co., NY.

Reza, F.M. (1961), *An Introduction to Information Theory*, Dover Publications, Inc., NY.

Richardson, A.R. (1966), *Business Economics*, Macdonald & Evans, Braintree, MA.

Ritzer, G. (1983), *Sociological Theory*, Alfred A. Knopf, NY.

Ritzer, G. (1993), *The McDonaldization of Society: An Investigation into the Changing Character of Contemporary Social Life*, Pine Forge Press, Thousand Oaks, CA.

Roberts, N.C. and R.T. Bradley (1991), Stakeholder Cooperation and Innovation: A Study of Public Policy Initiation at the State Level, *Journal of Applied Behavioral Science*, 27(2), pp.209-227.

Rosen, R. (1977), Complexity as a Systems Property, *Int. J. of Gen. Systems*, 3(4), pp. 227-232.

Rosenzmeig, M.R., A.L. Leiman, and S.M. Breedlove (1999), *Biological Psychology: An Introduction to Behavioral, Cognitive, and Clinical Neuroscience,* 2nd ed., Sinauer Associates, Inc., Publishers, Sunderland, MS.

Royce, W.W. (1970), Managing the Development of Large Software Systems: Concepts and Techniques, *Proc. WESCON*, August, USA.

Rubinstein, M.F. and I.R. Firstenberg (1995), *Patterns of Problem Solving*, 2nd ed., Prentice Hall, Inc., Englewood Cliffs, NJ.

Rumbaugh, J. et al. (1991), *Object-Oriented Modeling and Design*, Prentice Hall.

Rumbaugh, J., I. Jacobson, and G. Booch (1998*), The Unified Modeling Language Reference Manual,* ACM Press, New York.

Russell, B. (1961), *Basic Writings of Bertrand Russell*, George Allen & Unwin Ltd., London.

Russell, B. (1948), *Other Minds are Known by Analogy from One's Own Case*, in Human Knowledge: Its Scopes and Limits, London, Unwin Hyman Ltd.

Sabloniere, P. (2002), GRID in E-Business, Keynote speech, *Proc. 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, Montpellier, France, Sept., pp.4.

Salvendy, G. (2006), *Handbook of Human Factors and Ergonomics*, 3rd ed., Wiley.

Sapir, E. (1929), The Status of Linguistics as a Science, *Language*, 5, pp. 207-214.

Schael, T. (1998), Workflow Management Systems for Process Organizations, Second Edition*, Lecture Notes in Computer Science, 1096,* Springer-Verlag, Berlin.

Schedrovitzk, G.P. (1962), Methdological Problems of Systems Research, *General Systems Yearbook*, 11, pp. 27-53.

Schein, E.H. (1961), Management Development as a Process at Influence, *Industrial Management Review*, 2(2), Spring, pp.59-77.

Schilpp, P.A. (1946), The Philosophy of Bertrand Russell, *American Mathematical Monthly*, 53(4), p.210.

Schmenner, R.W. and M.L. Swink (1998), On Theory in Operations Management, *Journal of Operations Management*, 17, pp.97–113.

Schmidt, D. (1988), *Denotational Semantics: A Methodology for Language Development*, W. C. Brown Publishers, Dubuque, IA.

Schmidt, D.A. (1994), *The Structure of Typed Programming Languages.* MIT Press.

Schmidt, D. (1996), Programming Language Semantics, *ACM Computing Surveys,* 28(1), March.

Schneider, S.A. (1989), *Correctness and Communication in Real-Time Systems,* D. Phil. Thesis, Oxford University.

Schneider, S.A. (1991), An Operational Semantics for Timed CSP, *Programming Research Group Technical Report PRG-1-91*, Oxford University.

Schonberger, R.J. (1981), Why Projects are 'Always' Late: A Rationale Based on Manualsimulation of a PERT/CPM Network. *Interfaces,* 11, pp. 66-70.

Scott, D. (1982), Domains for Denotational Semantics, In *Automata, Languages and Programming IX*, Springer-Verlag, Berlin, pp. 577-613.

Scott, D.S. and C. Strachey (1971), Towards a Mathematical Semantics for Computer Languages, *Programming Research Group Technical Report PRG-1-6,* Oxford University.

Sepulveda, J.A., W.E. Souder, and B.S. Gottfried (1984), *Schaum's Outline of Theories and Problems of Engineering Economics*, McGraw-Hill Inc., NY.

SESC (1996), *Proc. Forum on Software Engineering Standards Issues* (SES'96), IEEE SESC, Montreal, Canada, Oct.

SESC (1997), *Proc. 2nd IEEE International Software Engineering Standards Symposium* (ISESS'97), IEEE SESC, Walnut Creek, California, June.

SESC (1999), *Proc. 3rd IEEE International Software Engineering Standards Symposium* (ISESS'99), IEEE CS Press, Curitiba, Brazil, May.

Shannon, C.E. (1948), A Mathematical Theory of Communication, *Bell System Technical Journal*, 27, pp.379-423 and 623-656.

Shannon, C.E. (1949a), Communication in the Presence of Noise*, Proc. IRE*, 37, pp.10-21.

Shannon, C.E. (1949b), Communication Theory of Secrecy Systems*, Bell System Technical Journal*, 28, pp.656-715.

Shannon, C.E. (1951), Prediction and Entropy of Printed English*, Bell System Technical Journal*, 30(27), pp.50-64.

Shannon, C.E. (1959), Coding Theorems for a Discrete Source with a Fidelity Criterion, *IRE National Convention Record*, Part 4, pp. 142-163.

Shannon, C.E. and W. Weaver (1949), *The Mathematical Theory of Communication*, Illinois University Press, Urbana, IL, USA.

Shannon, C.E. *ed.* (1956), *Automata Studies*, Princeton U. Press, Princeton.

Shao, J. and Y. Wang (2003), A New Measure of Software Complexity based on Cognitive Weights, *Canadian Journal of Electrical and Computer Engineering*, 28(2), pp.69-74.

Shewhart, W.A. (1939), *Statistical Method from the Viewpoint of Quality Control,* The Graduate School, George Washington University, Washington, D.C.

Siever, E., A. Weber, S. Figgins, and A. Oram (2003), *Linux in a Nutshell*, 4th ed., O'Reilly & Associates.

Silberschatz, A., P. Galvin and G. Gagne (2003), *Applied Operating System Concepts*, John Wiley & Sons, Inc., Danvers, MA.

Simon H.A. (1957), *Models of Man: Social and Rational*, Wiley, London.

Simon, H.A. (1960), *The New Science of Management Decision*, Harper & Row, New York.

Simon, H. (1965), Architecture of Complexity, *General Systems Yearbook*, 10, pp. 63-76.

Sintzoff, M. (1989), The Scientific Engineering of Software, *Communications of the ACM*, 32(2), pp. 258.

Skarda, C.A. and W.J. Freeman (1987), How Brains Make Chaos into Order, *Behavioral and Brain Sciences*, 10.

Skilling, J. (1989), Classic Maximum Entropy, in J. Skilling *ed.*, *Maximum Entropy and Bayesian Methods*, Kluwer, New York, pp. 45-52.

Skinner, B.E. (1948), *Determinism Rules Out Freedom*, in Walden Two, Macmillan Publishers.

Slavin, S.L. (1988), *Economics: A Self-Teaching Guide*, John Wiley & Sons, Inc., NY.

Sloane, A. (1993), *Computer Communications: Principles and Business Applications*, McGraw-Hill Book Co.

Slonneg, K. and B. Kurts (1995), *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Pub. Co.

Smith, A. (1776), *An Inquiry into the Nature and Causes of the Wealth of Nations*, Volumes 1 and 2, W. Strahan & T. Cadell, London.

Smith, R.E. (1993), *Psychology*, West Publishing Co., New York.

Snyder, A. (1987), Inheritance and the Development of Encapsulated Software Components, *in* Shriver and Wagner, eds., *Research Directions in Object-Oriented Programming,* MIT Press, Cambridge, MA, pp.165-188.

Soanes, C. and A. Stevenson (2003), *Oxford Dictionary of English*, 2nd ed., Oxford University Press, UK.

Sober, E. (1995), *Core Questions in Philosophy: A Text with Readings*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ.

Solso, R.L. ed. (1999), *Mind and Brain Science in the 21st Century*, The MIT Press, Cambridge, MS.

Sommerville, I. (1996), *Software engineering*, 5th ed., Addison-Wesley, 1996.

Sparks, S., K. Benner, and C. Faris (1996), Managing Object-Oriented Framework Reuse, *IEEE Computer*, September, pp 52-62

Spencer, A. (1991), *Morphological Theory*, Blackwell, Cambridge, MA.

Spivey, J.M. (1988), *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press.

Spivey, J.M. (1992), *The Z Notation: A Reference Manual (2nd ed.)* Prentice-Hall, London.

SQPL (1990), *SQPA: Software Quality and Productivity Analysis at Hewlett Packard,* Hewlett Packard Software Quality and Productivity Laboratory, HP Report.

Squire, L.R., B. Knowlton, and G. Musen (1993), The Structure and Organization of Memory, *Annual Review of Psychology*, 44, pp.453 – 459.

Stallings, W. (1987), *Computer Architecture and Organization*, Macmillan, NY.

Stallings, W. (2000), *Local and Metropolitan Area Networks*, 6th ed., Prentice Hall Inc.

Sternberg, R.J. (1998), *In Search of the Human Mind*, 2nd ed., Harcourt Brace & Co., Orlando, FL.

Steven, A. (1980), *Decision Support Systems: Current Practice and Continuing Challenges*, Addison-Wesley, Reading, MA.

Stickgold, R., J.A. Hobson, R. Fosse, and M. Fosse (2001), Sleep, Learning, and Dreams: Off-Line Memory Reprocessing, *Science*, 294(5544), November, pp. 1048-1051.

Stillings, N., M.H. Feinstein (1987), *Cognitive Science: An Introduction,* MIT Press, Cambridge, MA.

Stoy, J.E. (1997), *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, The MIT Press, Cambridge, MA.

Stroustrup, B. (1982), Classes: An Abstract Data Type Facility for the C Language, *ACM Software Engineering Notes*, 17, pp. 42-52.

Stroustrup, B. (1986), *The C++ Programming Language*, Addison-Wesley Publishing Co., MA.

Stroustrup, B. (1987), What is 'Object-Oriented Programming?' *Proc. European Conference on OO Programming (ECOOP'87)*, Paris.

Stubbs, D.F. and N.W. Webre (1985), *Data Structures with Abstract Data Types and Pascal,* Brooks/Cole Publishing Co., Monterey, CA.

Taguchi, G. (1986), *Introduction to Quality Engineering: Designing Quality into Products and Processes*, The Organization, Tokyo.

Taibi, T. and D.C.L. Ngo (2003), Formal Specification of Design Patterns - A Balanced Approach, *Journal of Object Technology*, 2(4), pp. 127-140.

Takahara, Y. and T. Takai (1985), Category Theoretical Framework of General Systems, *Int. J. Gen. Sys*., 11(1), pp. 1-33.

Tan, X. and Y. Wang (2003), Specification of Abstract Data Types using Real-Time Process Algebra, *Proc. 16th Canadian Conference on Electrical and Computer Engineering* (CCECE'03), IEEE CS Press, Montreal, Canada, May, pp.1293-1296.

Tan, X. and Y. Wang (2005), A Denotational Semantics of RTPA, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 2003-2006.

Tan, X. and Y. Wang (2006), Transforming RTPA Mathematical Models of System Behaviors into C++, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 362-369.

Tan, X., Y. Wang, and C.F. Ngolah (2004a), A Novel Type Checker for Software System Specifications in RTPA, *Proc. 17th Canadian Conference on Electrical and Computer Engineering* (CCECE'04), IEEE CS Press, Niagara Falls, ON, Canada, May, pp. 1549-1552.

Tan, X., Y. Wang, and C.F. Ngolah (2004b), Specification of the RTPA Grammar and Its Recognition, *Proc. 3rd IEEE International Conference on Cognitive Informatics* (ICCI'04), IEEE CS Press, Canada, August, pp. 54-63.

Tan, X., Y. Wang, and F.C. Ngolah (2005), Implementation of the Kernel Techniques of Real-Time Process Algebra (RTPA), *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp.1019-1022.

Tan, X., Y. Wang, and C.F. Ngolah (2006), Design and Implementation of an Automatic RTPA Code Generator, *Proc. 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, Ottawa, ON, Canada, May 8 -10, pp. 1605-1608.

Tanenbaum, A.S. (1994), *Distributed Operating Systems*, Prentice Hall Inc.

Tanenbaum, A.S. and A. Tanenbaum (2001), *Modern Operating Systems*, Prentice Hall Inc.

Tarski, A. (1944), The Semantic Conception of Truth, *Philosophic Phenomenological Research*, 4, pp.13-47.

Tayler, F.W. (1911), *Principles of Scientific Management*, Harper and Row, NY.

Terman, L.M. and M. Merrill (1961), *Stanford-Binet Intelligence Scale, Manual for the Third Revision*, Houghton Mifflin.

Thomas, I. (1994), Software Processes and Business Processes, *Proc. 3rd International Conference on Software Processes* (ICSP3), Reston, VA, Oct.

Thomas, L. (1974), *The Lives of a Cell: Notes of a Biology Watcher*, Viking Press, NY.

Thomas, R., L.R. Rogers, and J.L. Yates (1986), *Advanced Programmer's Guide to Unix System V*, Osborne McGraw-Hill, Berkeley, CA.

Tomassi, P. (1999), *Logic,* Routledge, London and New York.

Tribus, M. (1961), Information Theory as the Basis for Thermostatistics and Thermodynamics, *Journal of Applied Mechanics*, 28, pp.108.

Tripp, L. (1996), Underlying Principles for IEEE Software Engineering Standards, Business Planning Group, IEEE Software Engineering Standards Committee, *Proc. 1st IEEE International Software Engineering Standards Symposium* (ISESS'96).

Turing, A.M. (1936), On Computable Numbers, with an Application to the Entscheidungs Problem, *Proc. London Mathematical Society*, 2(42), pp.230-265, and 2(43), pp.544-546, 1036.

Turing, A.M. (1950), Computing Machinery and Intelligence, *Mind,* 59, pp. 433-460.

Ure, A. (1835), *The Philosophy of Manufactures,* Chas. Knight, London.

van Heijenoort, J. (1997), *From Frege to Godel, A Source Book in Mathematical Logic 1879-1931*, Harvard Univ. Press, Cambridge, MA.

Vereijken, J.J. (1995), A Process Algebra for Hybrid Systems, in A. Bouajjani and O. Maler eds., *Proc. 2^nd^ European Workshop on Real-Time and Hybrid Systems*, Grenoble, France, June.

Vliet, H.V. (2000), *Software Engineering: Principles and Practice,* 2nd ed., John Wiley & Sons Ltd., West Sussex, England.

von Bertalanffy, L. (1952), *Problems of Life: An Evolution of Modern Biological and Scientific Thought,* C.A. Watts, London.

von Neumann, J. (1946), The Principles of Large-Scale Computing Machines, *Annals of History of Computers*, 3(3), pp. 263-273.

von Neumann, J. (1958), *The Computer and the Brain*, Yale Univ. Press, New Haven.

von Neumann, J. (1963), *General and Logical Theory of Automata,* A.H. Taub ed., Collected Works, Vol. 5, Pergamon, pp. 288-328.

von Neumann, J. and A.W. Burks (1966), *Theory of Self-Reproducing Automata*, Univ. of Illinois Press, Urbana, IL.

von Neumann, J. and Morgenstern, O. (1980), *Theory of Games and Economic Behavior*, Princeton Univ. Press.

Votta, L.G., and A. Porter (1995), Experimental Software Engineering: A Report on the State of the Art, *Proc. International Conference on Software Engineering* (ICSE'95), Seattle, Washington USA, pp.277-279.

Vu, N.-C. and Y. Wang (2004), Specification of Design Patterns using Real-Time Process Algebra (RTPA), *Proc. of 2004 Canadian Conference on Electrical and Computer Engineering (CCECE'04),* IEEE CS Press, Niagara Falls, ON, Canada, May, pp. 1545-1548.

Wald, A. (1950), *Statistical Decision Functions*, John Wiley & Sons.

Wang, Y. (2007a), The Theoretical Framework of Cognitive Informatics, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(1), Jan., pp. 1-27.

Wang, Y. (2007b), Cognitive Informatics: Exploring Theoretical Foundations for Natural Intelligence, Neural Informatics, Autonomic Computing, and Agent Systems, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(1), Jan., pp. (e)1-10.

Wang, Y. (2007c), Exploring Machine Cognition Mechanisms for Autonomic Computing, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(2), March, pp. (e)1-10.

Wang, Y. (2007d), On Laws of Work Organization in Human Cooperation, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(2), March, pp. 1-15.

Wang, Y. (2007e), Perspectives on Autonomic Computing, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(3), July, pp. (e)1-9.

Wang, Y. (2007f), Towards Theoretical Foundations of Autonomic Computing, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(3), July, pp. 1-16.

Wang, Y. (2007g), The OAR Model of Neural Informatics for Internal Knowledge Representation in the Brain, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(3), July, pp. 64-75.

Wang, Y. (2007h), The Cognitive Processes of Formal Inferences, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(4), Dec., pp. 75-86.

Wang, Y. (2007i), Formal Description of the Cognitive Processes of Perceptions with Emotions, Motivations, and Attitudes, *The International Journal of Cognitive Informatics and Natural Intelligence* (IJCINI), IPI Publishing, USA, 1(4), Dec., pp. 1-13.

Wang, Y. (2007j), *Cognitive Informatics: A Transdisciplinary Field Exploring Natural and Artificial Intelligence*, IPI Publishing Co., USA, to appear.

Wang, Y. (2007k), Cognitive Informatics Foundations of Natural Intelligence, keynote speech, *Proc. 6th IEEE International Conference on Cognitive Informatics* (ICCI'07), IEEE CS Press, July, to appear.

Wang, Y. (2007*l*), Formal Linguistics and the Deductive Grammar, *Proc. 6th IEEE International Conference on Cognitive Informatics* (ICCI'07), Lake Tahoe, California, USA, IEEE CS Press, July, to appear.

Wang, Y. (2006a), On the Informatics Laws and Deductive Semantics of Software, *IEEE Transactions on Systems, Man, and Cybernetics (C),* 36(2), March, pp.161-171.

Wang, Y. (2006b), Cognitive Informatics - Towards the Future Generation Computers that Think and Feel, Keynote speech, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), Beijing, China, IEEE CS Press, July, pp. 3-7.

Wang, Y. (2006c), Cognitive Complexity of Software and Its Measurement, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 226-235.

Wang, Y. (2006d), On Abstract Systems and System Algebra, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 332-343.

Wang, Y. (2006e), On Concept Algebra and Knowledge Representation, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 320-331.

Wang, Y. (2006f), On the Big-R Notation for Describing Iterative and Recursive Behaviors, *Proc. 5th IEEE International Conference on Cognitive Informatics* (ICCI'06), IEEE CS Press, Beijing, China, July, pp. 132-140.

Wang, Y. (2006g), A Mathematical Model for Explaining the Mythic Man-Month, *Proc. 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, Ottawa, ON, Canada, May 8 – 10, pp. 2358-2361.

Wang, Y. (2006h), A Unified Mathematical Model of Programs, *Proc. 19th Canadian Conference on Electrical and Computer Engineering* (CCECE'06), Ottawa, ON, Canada, May 8 – 10, pp. 2346-2349.

Wang, Y. (2006i), On Constraints and Counter-Measures for Software Engineering, *Proc. 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, Ottawa, ON, Canada, May 8 – 10, pp. 2354-2357.

Wang, Y. (2006j), Cognitive Informatics and Contemporary Mathematics for Knowledge Representation and Manipulation, *Proc. 1st Int. Conf. on Rough Sets and Knowledge Technology* (RSKT'06), Chongqing, China, Lecture Note on AI, LNAI 4062, pp. 69-78.

Wang, Y. (2005a), On the Mathematical Laws of Software, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1086-1089.

Wang, Y. (2005b), A Novel Decision Grid Theory for Dynamic Decision Making, Proc. 4th IEEE International Conference on Cognitive Informatics (ICCI'05), IEEE CS Press, Irvine, California, USA, August, pp. 308-314.

Wang, Y. (2005c), Cognitive Informatics and Internal Signal Processing in the Brain, Proc. 26th Annual Meeting of the Canadian and Applied Industrial Mathematics Society (CAIMS'05), Univ. of Manitoba, Winnipeg, June, pp. 20-21.

Wang, Y. (2005d), Economic Models of Software Engineering and the Software Maintenance Crisis, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1814-1817.

Wang, Y. (2005e), Mathematical Models and Properties of Games, Proc. 4th IEEE International Conference on Cognitive Informatics (ICCI'05), IEEE CS Press, Irvine, California, USA, August, pp. 294-300.

Wang, Y. (2005f), On Cognitive Properties of Human Factors in Engineering, Proc. 4th IEEE International Conference on Cognitive Informatics (ICCI'05), IEEE CS Press, Irvine, California, USA, August, pp. 174-182.

Wang, Y. (2005g), On the Cognitive Foundations and Abstract Means of System Designs, Proc. 2nd International Conference on Design Education, Innovation, and Practice, Kananaskis, AB., Canada, July 18-20, 2005, pp. T3.1-3.8.

Wang, Y. (2005h), The Development of the IEEE/ACM Software Engineering Curricula, *IEEE Canadian Review,* 51(2), May, pp.16-20.

Wang, Y. (2005i), On the Organization Laws of Software Engineering, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1818-1821.

Wang, Y. (2005j), Psychological Experiments on the Cognitive Complexities of Fundamental Control Structures of Software Systems, Keynote Speech, *Proc. 4th IEEE International Conference on Cognitive Informatics* (ICCI'05), IEEE CS Press, Irvine, California, USA, August, pp. 4-5.

Wang, Y. (2005k), Sociological Models of Software Engineering, *Proc. the 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1806-1809.

Wang, Y. (2005*l*), System Science Models of Software Engineering, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1802-1805.

Wang, Y. (2004a), On Autonomic Computing and Cognitive Processes, Keynote Speech, *Proc. 3rd IEEE International Conference on Cognitive Informatics* (ICCI'04), IEEE CS Press, Canada, August, pp. 3-4.

Wang, Y. (2004b), On Cognitive Informatics Foundations of Software Engineering, *Proc. 3rd IEEE International Conference on Cognitive Informatics* (ICCI'04), IEEE CS Press, Canada, August, pp. 22-31.

Wang, Y. (2004c), On the Engineering Foundations of Software Engineering, *Proc. 6th Canadian Conference on Computer and Software Engineering Education* (C3SEE'04), U of C Press, Calgary, Canada, March, pp.105-116.

Wang, Y. (2003a), Cognitive Informatics: A New Transdisciplinary Research Field, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* 4(2), pp.115-127.

Wang, Y. (2003b), On Cognitive Informatics, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* 4(2), pp.151-167.

Wang, Y. (2003c), Using Process Algebra to Describe Human and Software Behaviors, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* 4(2), pp.199-213.

Wang, Y. (2003d), *Cognitive Informatics Models of Software Agent Systems and Autonomic Computing,* Keynote Speech, *Proc. 1st International Conference on Agent-Based Technologies and Systems* (ATS'03), Univ. of Calgary Press, Calgary, Canada, August, pp.25.

Wang, Y. (2003e), *On Cognitive Mechanisms of the Eyes: the Sensor vs. the Browser of the Brain,* Keynote Speech, *Proc. 2nd IEEE International Conference on Cognitive Informatics* (ICCI'03), IEEE CS Press, London, UK, August, pp.225.

Wang, Y. (2003f), The Measurement Theory for Software Engineering, *Proc. 16th Canadian Conference on Electrical and Computer Engineering* (CCECE'03), IEEE CS Press, Montreal, Canada, May, pp.1321-1324.

Wang, Y. (2002a), The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal*, Baltzer Science Publishers, Oxford, 14, Oct., pp. 235-274.

Wang, Y. (2002b), The Real-Time Process Algebra (RTPA) and Its Applications, Invited Speech, *Proc. 10th Anniversary Colloquium of the United Nations University / International Institute for Software Technology* (UNU/IIST'02), Lisbon, Portugal, March, pp.72.

Wang, Y. (2002c), A New Mathematics for Describing Notions and Thoughts in Software Design, *Proc. First IEEE International Conference on Cognitive Informatics* (ICCI'02), Calgary, AB., Canada, IEEE CS Press, August, pp.193-202.

Wang, Y. (2002d), On Cognitive Informatics, Keynote Speech, *Proc. First IEEE International Conference on Cognitive Informatics* (ICCI'02), Calgary, AB., Canada, IEEE CS Press, August, pp.34-42.

Wang, Y. (2002e), The Latest Development in Cognitive Informatics, Keynote Speech, *Proc. 8th International Conference on Object-Oriented Information Systems* (OOIS'02), Montpellier, France, Sept., LNCS 2452, Springer, pp.5.

Wang, Y. (2002f), On Software Engineering Measurement Deployment in GQM, *Proc. 2nd ASERC Workshop on Quantitative and Soft Computing Based Software Engineering* (QSSE'02), Banff, AB, Canada, Feb., pp.1-8.

Wang, Y. (2002g), On the Information Laws of Software, Keynote Speech, *Proc. First IEEE International Conference on Cognitive Informatics* (ICCI'02), Calgary, AB., Canada, IEEE CS Press, August, pp.132-144.

Wang, Y. (2002h), On Formal Methods Education in Software Engineering, *Proc. 5th Canadian Conference on Computer and Software Engineering Education* (C3SE2'02), Winnipeg, Canada, May, pp.71-79.

Wang, Y (2002i), On Teaching Theoretical Foundations of Software Engineering, *Proc. 5th IEEE Canadian Conference on Computer and Software Engineering Education* (C3SE2'02), Winnipeg, Canada, May, pp.49-57.

Wang, Y. (2001a), Formal Description of the UML Architecture and Extendibility, *The International Journal of the Object*, Hermes Science Publications, Paris, 6(4), pp.469-488.

Wang, Y. (2001b), Software Engineering Standards: Review and Perspectives, Chapter 28, in S.K. Chang ed., *Handbook of Engineering and Knowledge Engineering,* World Scientific Publishing Co., pp.277-304.

Wang, Y. (2001c), A Process-Oriented Infrastructure for Software Engineering Education, *Proc. 4th Canadian Conference on Computer Engineering Education* (CCCEE'01), IEEE and Univ. of New Brunswick, NB, Canada, May, pp.119-126.

Wang, Y. (2001d), A Software Engineering Measurement Framework (SEMF) for Teaching Quantitative Software Engineering, *Proc. 4th Canadian Conference on Computer Engineering Education* (CCCEE'01), IEEE and Univ. of New Brunswick, NB, Canada, May, pp.88-101.

Wang, Y. (2001e), A Web-Based European Software Process Benchmarking Server, *Proc. 23rd IEEE International Conference on Software Engineering* (ICSE'01), Toronto, May, pp.439 - 440.

Wang, Y. (2001f), Formal Description of Object-Oriented Software Measurement and Metrics in SEMS, *Proc. 7th International Conference on Object-Oriented Information Systems* (OOIS'01), Calgary, August, Canada, pp.123-132.

Wang, Y. (2001g), Software Engineering: Hard Problems, Extending Domains, and Fundamental Objectives, *Proc. 4th Canadian Conference on Computer Engineering Education* (CCCEE'01), IEEE and Univ. of New Brunswick, Canada, May, pp.127.

Wang, Y. (2000), Progress and Trends in Software Engineering, Keynote Speech, *Proc. 2000 Conference of IEEE Sweden*, Stockholm, May, pp.1-16.

Wang, Y. (1996), A New Sorting Algorithm: Self-Indexed Sort, *ACM SIGPLAN*, 31(3), March, USA, pp. 28-36.

Wang Y., G. King, I. Court, M. Ross, and G. Staples (1997), On Testable Object-Oriented Programming, *ACM Software Engineering Notes (SEN)*, July, 22(4), pp.84-90.

Wang Y., G. King, D. Patel, I. Court, G. Staples, M. Ross, and S. Patel (1998a), On Built-in Test and Reuse in Object-Oriented Programming, *ACM Software Engineering Notes (SEN)*, 23(4), pp.60-64.

Wang Y., G. King, A. Dorling, D. Patel, I. Court, G. Staples, and M. Ross (1998b), A Worldwide Survey on Software Engineering Process Excellence, *Proc. of 2oth IEEE International Conference on Software Engineering* (IEEE ICSE'98), Kyoto, April, IEEE Press, pp.439-442.

Wang Y., G. King, A. Doling, and H. Wickberg (1999a), A Unified Framework of the Software Engineering Process System Standards and Models, *Proc. 4th IEEE International Software Engineering Standards Symposium* (IEEE ISESS'99), IEEE CS Press, Brazil, May, pp.132-141.

Wang Y., G. King, A. Dorling, M. Ross, G. Staples, and I. Court (1999b), A Worldwide Survey on Best Practices Towards Software Engineering Process Excellence, *ASQ Journal of Software Quality Professional*, 2(1), Dec., pp. 34-43.

Wang Y., H. Wickberg, and A. Dorling (1999c), Establishment of a National Benchmark of Software Engineering Practices, *Proc. 4th IEEE International Software Engineering Standards Symposium* (IEEE ISESS'99), IEEE CS Press, Brazil, May, pp.16-25.

Wang, Y., G. King, D. Patel, S. Patel, and A. Dorling (1999d), On Coping with Software Dynamic Inconsistency at Real-Time by the Built-in Tests, *Annals of Software Engineering: An International Journal,* Baltzer Science Publishers, Oxford, 7, pp.283-296.

Wang, Y., I. Chouldury, D. Patel, S. Patel, A. Dorling, H. Wickberg, and G. King (1999e), On the Foundations of Object-Oriented Information Systems, *The International Journal of the Object*, 5(1), Feb, pp.9-27.

Wang, Y. and G. King (2000a), *Software Engineering Processes: Principles and Applications*, CRC Book Series in Software Engineering, Vol.1, CRC Press, USA.

Wang, Y. and G. King (2000b), A New Approach to Benchmark-Based Process Improvement, *Proc. 2000 European Software Process Improvement* (EuroSPI'00), Copenhagen, Nov., pp.1.40-1.49.

Wang Y., G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross (2000), On Built-in Tests Reuse in Object-Oriented Framework Design, *ACM Journal on Computing Surveys*, 32(1es), March, pp.7-12.

Wang, Y. and D. Patel (2000), Comparative Software Engineering: Review and Perspectives, *Annals of Software Engineering: An International Journal*, Baltzer Science Publishers, Oxford, 10, Dec., pp.1-10.

Wang, Y. and H. Leung (2001), A Benchmark-Based Adaptable Software Process Model, *Proc. 2001 IEEE EuroMicro Conference on Software Process and Product Improvement* (EuroMicro'01), Warsaw, Poland, September, pp.216-224.

Wang, Y. and A. Bryant (2002), Process-Based Software Engineering: Building the Infrastructure, *Annals of Software Engineering: An International Journal,* 14, Oct., pp. 9-37.

Wang, Y. and N.C. Foinjong (2002), Formal Specification of a Real-Time Lift Dispatching System, *Proc. 15th IEEE Canadian Conference on Electrical and Computer Engineering* (CCECE'02), Winnipeg, Manitoba, Canada, May, Vol.2, pp.669-674.

Wang, Y., R.H. Johnston, and M.R. Smith, eds. (2002), *Proc. 1st IEEE International Conference on Cognitive Informatics* (ICCI'02), IEEE Computer Society Press, July.

Wang, Y. and D. Gafurov (2003), The Cognitive Process of Comprehension, *Proc. 2nd IEEE International Conference on Cognitive Informatics* (ICCI'03), IEEE CS Press, London, UK, August, pp.93-97.

Wang, Y., D. Liu, and Y. Wang (2003), Discovering the Capacity of Human Memory, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* 4(2), pp.189-198.

Wang, Y. and C.F. Noglah (2003), Formal Description of Real-Time Operating Systems using RTPA, *Proc. 16th Canadian Conference on Electrical and Computer Engineering* (CCECE'03), IEEE CS Press, Montreal, Canada, May, pp.1247-1250.

Wang, Y. and J. Shao (2003), Measurement of the Cognitive Functional Complexity of Software, *Proc. 2nd IEEE International Conference on Cognitive Informatics* (ICCI'03), IEEE CS Press, London, UK, August, pp.67-74.

Wang, Y. and Y. Zhang (2003), Formal Description of an ATM System by RTPA, *Proc. 16th Canadian Conference on Electrical and Computer*

Engineering (CCECE'03), IEEE CS Press, Montreal, Canada, May, pp.1255-1258.

Wang, Y. and D. Liu (2004), Statistical Analysis of International Software Engineering Programs, *Proc. 6th Canadian Conference on Computer and Software Engineering Education* (C3SEE'04), U of C Press, Calgary, Canada, March, pp.249 - 290.

Wang, Y., and S. Patel (2004), On Modeling Object-Oriented Information Systems, *The International Journal of Software and System Modeling,* 3(4), pp.258-261.

Wang, Y., W. Kinsner, and M. Smith *eds.* (2004), *Proc. 2004 Canadian Conference on Computer and Software Engineering Education* (C3SEE'04), Univ. of Calgary Press, Calgary, Canada.

Wang, Y. and J. Huang (2005), Formal Models of Object-Oriented Patterns using RTPA, *Proc. 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1822-1825.

Wang, Y. and W. Kinsner (2006), Recent Advances in Cognitive Informatics, *IEEE Transactions on Systems, Man, and Cybernetics* (C), 36(2), March, pp.121-123.

Wang, Y. and Y. Wang (2006), Cognitive Informatics Models of the Brain, *IEEE Transactions on Systems, Man, and Cybernetics* (C), 36(2), March, pp.203-207.

Wang, Y., Y. Wang, S. Patel, and D. Patel (2006), A Layered Reference Model of the Brain (LRMB), *IEEE Transactions on Systems, Man, and Cybernetics* (C), 36(2), March, pp.124-133.

Wang, Y. and G. Ruhe (2007), The Cognitive Process of Decision Making, *The International Journal of Cognitive Informatics and Natural Intelligence* (JCINI), IPI Publishing, USA, 1(2), March, pp. 73-85.

Wang, Y., J. Huang, and Y. Tian (2008), On Hyper-Programming, *The International Journal of Cognitive Informatics and Natural Intelligence* (JCINI), IPI Publishing, USA, 2(3), July, to appear.

Warner, W.L. and J.O. Low (1947), *The Social System of the Modern Factory*, Yankee City Series, Vol. 4, Yale Univ. Press, New Haven, CN.

Wasserman, A. (1996), Toward a Discipline of Software Engineering*, IEEE Software,* Nov., pp.23-31.

Weber, M. (1947), *The Theory of Social and Economic Organization*, A.M. Henderson and T. Parsons trans., Oxford Univ. Press, NY.

Wegner, P. (1972), The Vienna Definition Language, *ACM Computing Surveys*, Vol.4, No.1, pp.5-63.

Weinberg, G.M. (1971), *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York.

Westen, D. (1999), *Psychology: Mind, Brain, and Culture*, 2nd ed., John Wiley & Sons, Inc., NY.

Whorf, B.L. (1941), The Relation of Habitual Thought and Behavior to Language, in *Language, Thought, and Reality*, The Technology Press of MIT, Cambridge, MA.

Wickens, C.D., S.E. Gordon, and Y. Liu (1998), *An Introduction to Human Factors Engineering*, Addison Wesley Longman, Inc., NY.

Widrow, B. and M.A. Lehr (1990), 30 Years of Adaptive Neural Networks: Perception, Madeline, and Backpropagation, *Proc. of the IEEE*, Sept., 78(9), pp. 1415-1442.

Wiener, N. (1948), *Cybernetics or Control and Communication in the Animal and the Machine*, MIT Press, Cambridge, MA.

Wiener, R. and L.J. Pinson (2000), *Fundamentals of OOP and Data Structures in Java*, Cambridge University Press.

Wiggins, J.A., B.B. Eiggins, and J.V. Zanden (1994), *Social Psychology*, 5th ed., McGraw-Hill, Inc., NY.

Wikstrom, A. (1987), *Functional Programming using Standard ML*, Prentice Hall, Englewood Cliffs, NJ.

William, B. (1991), *Creating Value for Customers: Design and Implementing a Total Corporate Strategy*, John Wiley & Sons, New York.

Williams, R.M. Jr. (1970), *American Society: A Sociological Interpretation*, 3rd ed., Alfred A. Knopf, NY.

Wilson, L.B. and R.G. Clark (1988), *Comparative Programming Language*, Addison-Wesley Publishing Co., Wokingham, UK.

Wilson, R.A. and F.C. Keil (2001), The MIT *Encyclopedia of the Cognitive Sciences,* MIT Press.

Wirth, N. (1974), On the Design of Programming Languages, *Proc. 1974 IFIP Congress*, North-Holland, Amsterdam, pp.386-393.

Wirth, N. (1976), *Algorithm + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, NJ.

Wittig, A.F. (2001), *Schaum's Outlines of Theory and Problems of Introduction to Psychology*, 2nd ed., McGraw-Hill, NY.

Wood, D.J. and B. Gray (1991), Towards a Comprehensive Theory of Cooperation, *Journal of Applied Behavioral Science*, 27(2), 139-162.

Woodcock, J. and J. Davies (1996), *Using Z: Specification, Refinement, and Proof*, Prentice Hall International, London.

Wright, P.H. (2002), *Introduction to Engineering*, 3rd ed., John Wiley & Sons, Inc., New York, NY.

Wundt, W.M. (1873), *Principles of Physiological Psychology*, Partially translated by S. Diamond, in R.W. Rieber. Ed. (1980), W. Wundt and the Making of a Scientific Psychology, Plenum, New York.

Yao, Y.Z. and Y. Wang (2004), A New Approach to Test Cases Generation Based on Real-Time Process Algebra, *Proc. 17th Canadian Conference on Electrical and Computer Engineering* (CCECE'04), IEEE CS Press, Niagara Falls, ON, Canada, May, pp. 1515-1518.

Zachary, W., R. Wherry, F. Glenn, and J. Hopson (1982), Decision situations, decision processes, and decision functions: Towards a theory-based framework for decision-aid design. *Proc. of 1982 Conference on Human Factors in Computing Systems.*

Zadeh, L.A. (1965), Fuzzy Sets and Systems, J. Fox ed, *Systems Theory*, Polytechnic Press, Brooklyn, NY, pp. 29-37.

Zadeh, L.A. (1973), Outline of a New Approach to Analysis of Complex Systems, IEEE Transactions on Systems, Man, and Cybernetics, 1(1), pp. 28-44.

Zadeh, L. A. (1982), Fuzzy Systems Theory: Framework for Analysis of Buerocratic Systems, in R.E. Cavallo ed., *System Methods in Social Science Research*, Kluwer-Nijhoff, Boston, pp. 25-41.

Zander, A. (1979), The Psychology of the Group Process, *Annual Review of Psychology*, p.418.

Zhong, Y. (1996), *Principles of Information Science*, BPTU Press, Beijing, China.

Zuse, H. (1997), *A Framework of Software Measurement*, Walter de Gruyter, Berlin.

Appendix A

MATHEMATICAL SYMBOLS, NOTATIONS, AND ABBREVIATIONS

Mathematical symbols	Description
	Logic
$\wedge$	Conjunction
$\vee$	Disjunction
$\neg$	Negation
$\Rightarrow$	Implication
$\Leftrightarrow$	Equivalence
$\forall$	Universal qualifier, for all, or for every
$\exists$	Existential quantifier, there exist
$\mho$	Connectives, $\mho \in \{\wedge, \vee, \neg\}$
	Set
$\{\ldots\}$	A set
$(\ldots)$	A pair or tuple
$<\ldots>$	A sequence
$\varnothing$	The empty set
U	The universal set
$\in$	Is member of set
$\cup$	Union of sets
$\cap$	Intersection of sets
$\backslash$	Deference

#	Cardinal calculus, number of elements in a set
$\subset$	Is subset
$\subseteq$	Is subset or equal
$\times$	Cartesian product
$\oplus$	Symmetric difference
$\overline{S}$	Complement of set S
ÞS	A power set of set S
$\mid \ldots$	constraints or membership conditions of an element
$\mathbf{N}$	Set of natural numbers [1, ∞]
$\mathbf{N}_0$	Set of natural numbers [0, ∞]
$\mathbf{Z}$	Set of integers
$\mathbf{R}$	Set of real numbers
$\mathbf{B}$	Set of binary bytes
$\mathbf{H}$	Set of hexdecimals
$\mathbf{BL}$	Set of Boolean variables (**T**, **F**)
	Algebra
$\triangleq$	Definition
$\equiv$	Equivalence
$\vdash$	Yield
$\models$	To be relation
®	Cumulated relation
$\frown$	Concatenation
π	Addressing function
π^{-1}	Memory allocation
o	Composition of functions
$\sum$	Repeat sum
$\prod$	Repeat multiplication
R	Repeat a function (big-R calculus)
$\lfloor\ \rfloor$	Bottom of a decimal, the nearest minimum integer
$\lceil\ \rceil$	Ceiling of a decimal, the nearest largest integer
$\perp$	Bottom, undefined
~	Otherwise, else
$\propto$	Proportional to
iff	If and only if
	RTPA

$\mathfrak{P}$	The set of meta processes
$\mathfrak{R}$	The set of process relations (process operations)
$\mathfrak{T}$	The set of primitive types
$\mathbb{T}$	An arbitrary type ($\mathbb{T} \in \mathfrak{T}$)
$\mathbb{T}^*$	An arbitrary type for constant ($\mathbb{T}^* \in \mathfrak{T}^*$)
§	The system
℘	A program
P	A process
s	A statement
@	System event prefix
Ⓢ	System status prefix
⊙	System interrupt point suffix
Meta Processes	$\mathfrak{P}$
:=	Assignment
�	Evaluation
$\Rightarrow$	Addressing
$\Leftarrow$	Memory allocation
$\nLeftarrow$	Memory release
$\gtrdot$	Read
$\lessdot$	Write
$\vert\gtrdot$	Input
$\vert\lessdot$	Output
$\underline{\underline{@}}$	Timing
$\triangleq$	Duration
$\uparrow$	Increase
$\downarrow$	Decrease
!	Exception detection
$\otimes$	Skip
$\boxtimes$	Stop
§	System
Process Relations	$\mathfrak{R}$
$\rightarrow$	Sequence
$\curvearrowright$	Jump
$\vert$	Branch
$\vert \ldots \vert \ldots$	Switch
R^*	While-loop

R^{+}	Repeat-loop
R^{i}	For-loop
$\circlearrowleft$	Recursion
$\rightarrowtail$	Function call
$\parallel$	Parallel
$\oiint$	Concurrence
$\vert\vert\vert$	Interleave
»	Pipeline
↯	Interrupt
↳_t	Time-driven dispatch
↳_e	Event-driven dispatch
↳_i	Interrupt-driven dispatch
Primitive types	$\mathfrak{T}$
N	Natural number
Z	Integer
R	Real
S	String
BL	Boolean
T, F	Boolean constants
B	Byte
H	Hexadecimal
P	Pointer
TI = **hh:mm:ss:ms**	Time
D = **yy:MM:dd**	Date
DT = **yyyy:MM:dd:** **hh:mm:ss:ms**	Date/Time
RT	Run-time determinable type
ST	System architectural type
@*e***S**	Event
@*t***TM**	Timing
@*int*⦿	Interrupt
Ⓢs**BL**	Status
	System Algebra
$\sqsupseteq$ / $\sqsubseteq$	Super/sub relation
$\leftrightarrow$ / $\nleftrightarrow$	Related/independent

$=$	Equivalent
Π	Overlapped
$\sqcup$	Conjunction
$\boxminus$	Difference
$\Rightarrow$	Inheritance
$\overset{+}{\Rightarrow}$	Extension
$\overset{-}{\Rightarrow}$	Tailoring
$\overset{\sim}{\Rightarrow}$	Substitute
$\uplus$	Composition
$\pitchfork$	Decomposition
$\Lleftarrow$	Aggregation/generalization
$\vdash$	Specification
$\mapsto$	Instantiation
	Concept Algebra
$\succ$ / $\prec$	Super/sub relation
$\leftrightarrow$ / $\nleftrightarrow$	Related/independent
$=$	Equivalent
$\cong$	Consistent
$+$	Conjunction
$*$	Elicitation
$\sim$	Comparison
$\triangleq$	Definition
$\Rightarrow$	Inheritance
$\overset{+}{\Rightarrow}$	Extension
$\overset{-}{\Rightarrow}$	Tailoring
$\overset{\sim}{\Rightarrow}$	Substitute
$\uplus$	Composition
$\pitchfork$	Decomposition
$\Lleftarrow$	Aggregation/generalization
$\vdash$	Specification
$\mapsto$	Instantiation

Notations	**Description**
R	The big-R nation
§	A system
O( )	Order of complexity
Ξ	Instruction set of a given language
Ω	Behavioural space, constraints of a system
Θ	Environment of a system

Θ_t	Type environment
$\mathfrak{O}$	The empty system
$\mathfrak{U}$	The universal system
	Formal Linguistics
N	Noun
V	Verb
A	Adjective
Λ	Adverb
P	Preposition
τ	Determiner
δ	Degree word
κ	Qualifier
α	Auxiliary
γ	Conjunction word
¬	Negative
NP	Noun phrase
VP	Verb phrase
AP	Adjective phrase
ΛP	Adverb phrase
PP	Prepositional phrase
CP	Complement phrase
|=	To be
|⊂	To have
|>	To do
|>> … |>	Indirect to do
	EBNF
S ::= $S_1\ S_2 \ldots S_n$	Serial
S ::= $S_1\ [S_2] \ldots S_n$	Serial with option
S ::= $(S_1\ S_2 \ldots S_n)^*$	Repeat serial for 0 or more times
S ::= $(S_1\ S_2 \ldots S_n)^+$	Repeat serial for 1 or more times
S ::= $S_1 \mid S_2 \mid \ldots \mid S_n$	Alternative
S ::=	Alternative with option

$[S_1 \mid S_2 \mid \ldots \mid S_n]$	
$S ::= (S_1 \mid S_2 \mid \ldots \mid S_n)^*$	Repeat alternative for 0 or more times
$S ::= (S_1 \mid S_2 \mid \ldots \mid S_n)^+$	Repeat alternative for 1 or more times

Abbreviation	Description
ABM	Action Buffer Memory
AC	Autonomic Computing
ACM	Association of Computing Machinery
ADT	Abstract Data Types
AI	Artificial Intelligence
ANTLR	ANother Tool for Language Recognizer
ASQ	American Society of Quality
AST	Abstract Syntax Tree
ATM	Automated Teller Machine
BCS	Basic Control Structure
BIT	Built-In Test
BNF	Backus-Naur Form
BPA	Base Process Activity
CASE	Computer-Aided Software Engineering
CCITT	International Telegraph and Telephone Consulting Committee (now ITU)
CCS	The Calculus of Communicating Systems
CCSE	Computing Curricula – Software Engineering
CFG	Control Flow Graph
CH	Cohesion
CI	Cognitive Informatics
CIM	Cognitive Information Model
CLM	Component Logical Model
CMM	Capability Maturity Model
CMM	Cognitive Model of Memory
CNS	Central Nervous System
COCOMO	Constructive Cost Model
COTS	Commercial Off-The-Shelf (software components)
CP	Coupling
CPM	Critical Path Method

CPU	Central Processing Unit
CSCW	Computer-Supported Cooperative Work
CSP	Communication Sequential Processes
CU	Cognitive Unit
CWO	Cooperative Work Organization
DMA	Direct Memory Access
DOL	Division Of Labor
DSS	Decision Support System
DTSD	Distributed Time-Shared Development
EBNF	Extended Backus-Naur Form
EMM	Engineering Maturity Model
FCFS	First-Come-First-Served
FIFO	First-In-First-Out
FKS	Formal Knowledge System
FO	Function Object
FSM	Finite State Machine
FSM	Formal Socialization Model
GIM	Generic Intelligence Model
GUI	Graphic User Interface
HAMSD	Hierarchical Abstraction Model of System Descriptivity
HNC	Hierarchical Neural Cluster (model of memory)
HNH	Human Needs Hierarchy model
HTML	Hyper-Text Marking Language
IC	Imperative Computing
ICM	Intelligent Capability Metric
IE	Inference Engine
IEC	The International Electrotechnical Commission
IEEE	The Institute of Electrical and Electronics Engineers
IME	Information-matter-energy
IPO	Input-Process-Output model of systems
IQ	Intelligent Quotient
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
IT	Information Technology
ITU	International Telecommunication Union
LAN	Local Area Network
LDS	Lift Dispatching System

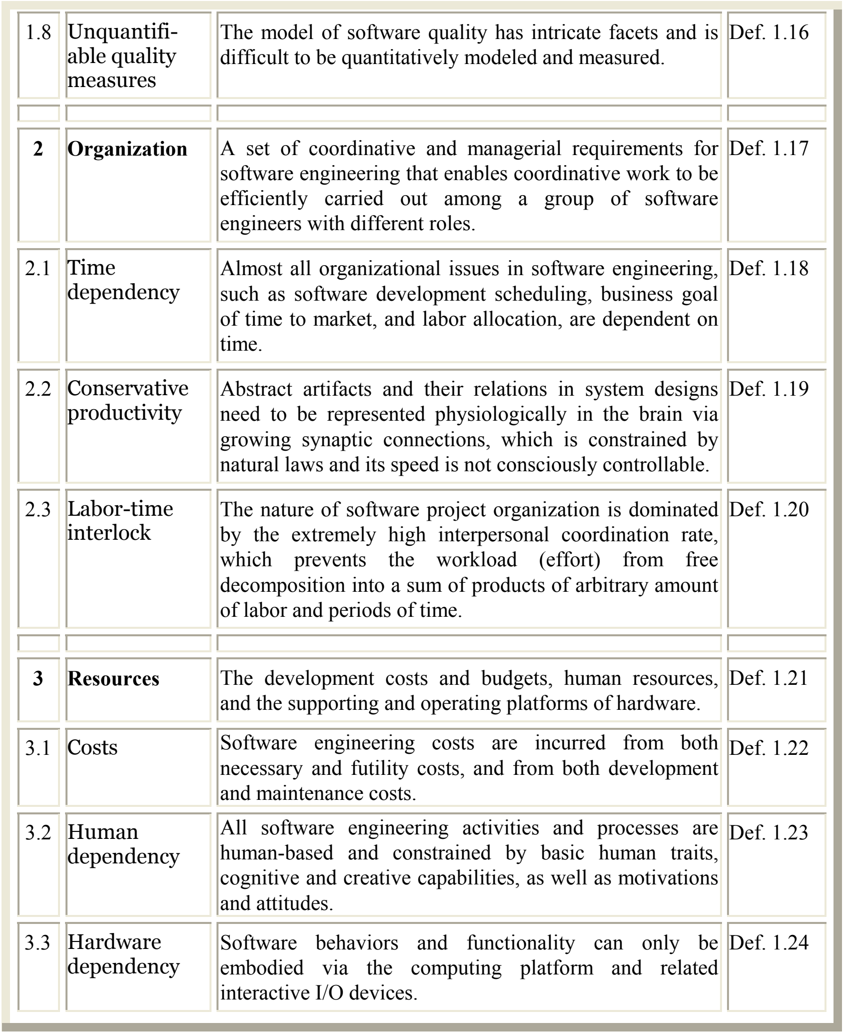
LIFO	Last-In-First-Out
LOC	Lines Of Code
LRMB	Layered Reference Model of the Brain
LTM	Long-Term Memory
MODEM	Modulator and demodulator
MPMC	Multi-Processor Multi-Clock
MPSC	Multi-Processor Single-Clock
NeI	Neural Informatics
NI	Natural Intelligence
NI-App	Natural Intelligent Applications
NI-OS	Natural Intelligent Operating System
NI-Sys	Natural Intelligent System
OAR	Object-Attribute-Relation
OO	Object-Orientation
OOP	Object-Oriented Programming
OPRM	Organization's Process Reference Model
OSI	Open Systems Interconnection
OT	Organization Tree
PBSE	Process-Based Software Engineering
PCB	Process Control Blocks
PE	Perception Engine
PERT	Program Evaluation and Review Technique
PTPM	Project's Tailored Process Model
RCB	Resource Control Block
RPC	Remote Procedure Call
RTOS	Real-Time Operating System
RTOS+	Real-Time Operating System plus
RTPA	Real-Time Process Algebra
SBM	Sensory Buffer Memory
SDL	Specification and Description Language (ITU)
SE	Software Engineering
SECM	Software Engineering Constraint Model
SEPRM	Software Engineering Process Reference Model
SESC	Software Engineering Standardization Committee (IEEE)
SLMC	Software Legacy Maintenance Cost
SMC	Sequential Message Chart

SMC	Software Maintenance Crisis
SOT	System Organization Tree
SPA	Software Process Assessment
SPI	Software Process Improvement
SPL	A Sample Programming Language
SPSC	Single-Processor Single-Clock
SQA	Software Quality Assurance
SS	Software Science
STM	Short-Term Memory
TCP/IP	Transmission Control Protocol/Internet Protocol
TCSE	IEEE Technical Council on Software Engineering
TQM	Total Quality Management
TSS	Telephone Switching System
UG	Universal Grammar
UML	Unified Modelling Language
USB	Universal Serial Bus
VM	Virtual Machine
VNA	von Neumann Architecture
VNM	Von Neumann Machine
WA	Wang Architecture
WAN	Wide Area Network
YACC	Yet Another Compiler-Compiler

Appendix B

CONSTRAINTS OF SOFTWARE ENGINEERING

No	Constraints	Description	Remark
1	**Cognition**	A set of innate cognitive attributes of software and the nature of the problems in software engineering that create the intricate relations of software objects and make software engineering inheritably difficult.	Def. 1.8
1.1	Intangibility	Software is abstract artifacts which is not constituted by physical objects or presence, and is difficult to be defined or expressed.	Def. 1.9
1.2	Complexity	Software is innately complex and its intricate internal connections and external couplings make it extremely difficult to be expressed or cognized.	Def. 1.10
1.3	Indeterminacy	The events, behaviors, or their sequence of occurring in a software system are not fully determinable on the basis of a given algorithm during design time; Instead, some of them may only be determined until run-time.	Def. 1.11
1.4	Diversity	The great variety of software in types, styles, architectures, behaviors, platforms, application domains, implementation techniques, usability, reliability, and quality.	Def. 1.12
1.5	Polymorphism	The approaches and styles of both software design and implementation are multifaceted and polyglottic.	Def. 1.13
1.6	Inexpressiveness	Software architectures and behaviors are inherently difficult to be expressed, modeled, represented, and quantified both formally and rigorously.	Def. 1.14
1.7	Inexplicit embodiment	Architectures and behaviors of software systems should be explicitly described by coherent symbolic notations in order to be processed and executed by computers.	Def. 1.15

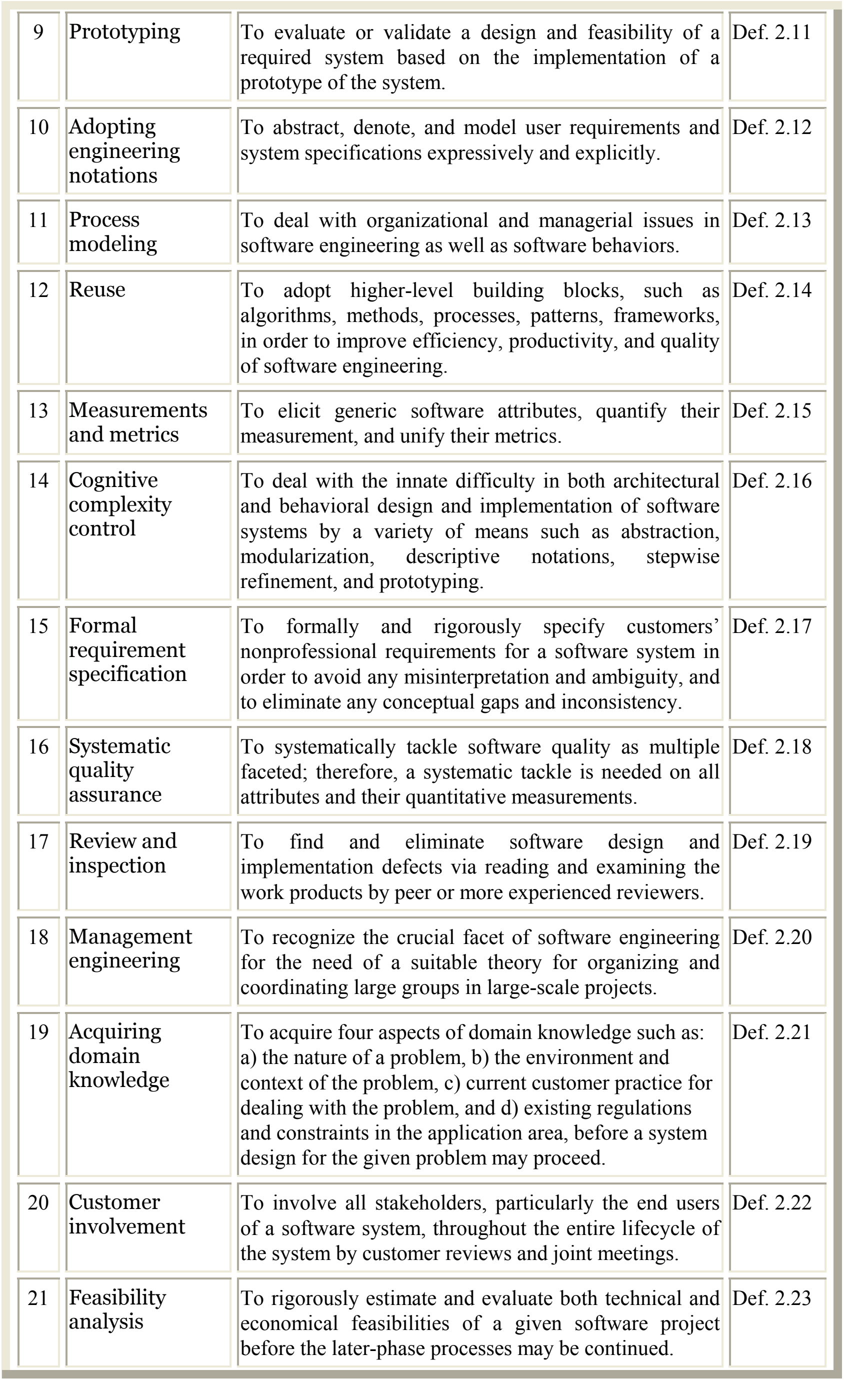


1.8	Unquantifiable quality measures	The model of software quality has intricate facets and is difficult to be quantitatively modeled and measured.	Def. 1.16
2	**Organization**	A set of coordinative and managerial requirements for software engineering that enables coordinative work to be efficiently carried out among a group of software engineers with different roles.	Def. 1.17
2.1	Time dependency	Almost all organizational issues in software engineering, such as software development scheduling, business goal of time to market, and labor allocation, are dependent on time.	Def. 1.18
2.2	Conservative productivity	Abstract artifacts and their relations in system designs need to be represented physiologically in the brain via growing synaptic connections, which is constrained by natural laws and its speed is not consciously controllable.	Def. 1.19
2.3	Labor-time interlock	The nature of software project organization is dominated by the extremely high interpersonal coordination rate, which prevents the workload (effort) from free decomposition into a sum of products of arbitrary amount of labor and periods of time.	Def. 1.20
3	**Resources**	The development costs and budgets, human resources, and the supporting and operating platforms of hardware.	Def. 1.21
3.1	Costs	Software engineering costs are incurred from both necessary and futility costs, and from both development and maintenance costs.	Def. 1.22
3.2	Human dependency	All software engineering activities and processes are human-based and constrained by basic human traits, cognitive and creative capabilities, as well as motivations and attitudes.	Def. 1.23
3.3	Hardware dependency	Software behaviors and functionality can only be embodied via the computing platform and related interactive I/O devices.	Def. 1.24

Appendix C

EMPIRICAL PRINCIPLES OF SOFTWARE ENGINEERING

No	Principle	Description	Remark
1	Abstraction	To elicit essential properties of a set of objects while omitting inessential details of them.	Def. 2.3
2	Decomposition and modularization	To partition and divide the functions of a software system into individual modules or components.	Def. 2.4
3	Information hiding	To mask and simplify unnecessary information of software at a given level from the lower level details.	Def. 2.5
4	Engineering approach	To adopt the proven generic engineering methodology and practice in software development and its organization.	Def. 2.6
5	Professionalism	To recognize the competence or skills expected for a professional software engineer gained in training and practice.	Def. 2.7
6	Tools and environments	To adopt software development tools and software engineering supporting environment in order to facilitate efficient organization of coordinative work or extend human physical and intelligent capability in software development.	Def. 2.8
7	Documentation	To represent system design and architectures, record work products, maintain traceability of serial decisions, log problems and maintenance solutions, and enable postmortem analysis.	Def. 2.9
8	Stepwise refinement	To deductively extend a conceptual model of the requirement for a given software system by a series of expatiated and incremental specifications at increased degrees of details.	Def. 2.10



9	Prototyping	To evaluate or validate a design and feasibility of a required system based on the implementation of a prototype of the system.	Def. 2.11
10	Adopting engineering notations	To abstract, denote, and model user requirements and system specifications expressively and explicitly.	Def. 2.12
11	Process modeling	To deal with organizational and managerial issues in software engineering as well as software behaviors.	Def. 2.13
12	Reuse	To adopt higher-level building blocks, such as algorithms, methods, processes, patterns, frameworks, in order to improve efficiency, productivity, and quality of software engineering.	Def. 2.14
13	Measurements and metrics	To elicit generic software attributes, quantify their measurement, and unify their metrics.	Def. 2.15
14	Cognitive complexity control	To deal with the innate difficulty in both architectural and behavioral design and implementation of software systems by a variety of means such as abstraction, modularization, descriptive notations, stepwise refinement, and prototyping.	Def. 2.16
15	Formal requirement specification	To formally and rigorously specify customers' nonprofessional requirements for a software system in order to avoid any misinterpretation and ambiguity, and to eliminate any conceptual gaps and inconsistency.	Def. 2.17
16	Systematic quality assurance	To systematically tackle software quality as multiple faceted; therefore, a systematic tackle is needed on all attributes and their quantitative measurements.	Def. 2.18
17	Review and inspection	To find and eliminate software design and implementation defects via reading and examining the work products by peer or more experienced reviewers.	Def. 2.19
18	Management engineering	To recognize the crucial facet of software engineering for the need of a suitable theory for organizing and coordinating large groups in large-scale projects.	Def. 2.20
19	Acquiring domain knowledge	To acquire four aspects of domain knowledge such as: a) the nature of a problem, b) the environment and context of the problem, c) current customer practice for dealing with the problem, and d) existing regulations and constraints in the application area, before a system design for the given problem may proceed.	Def. 2.21
20	Customer involvement	To involve all stakeholders, particularly the end users of a software system, throughout the entire lifecycle of the system by customer reviews and joint meetings.	Def. 2.22
21	Feasibility analysis	To rigorously estimate and evaluate both technical and economical feasibilities of a given software project before the later-phase processes may be continued.	Def. 2.23

22	Improve comprehensibility	To explicitly and expressively describe the intangible problem and its solution with improved understandability, readability, and cognitive capability.	Def. 2.24
23	Exception handling	To consider system design and specification not only customer required functions for a given system, but also all possible exceptions that may drive the system into illegal state(s) in the entire state space of the system.	Def. 2.25
24	Divide-and-Conquer	To suppose if a complex system may be divided into multiple components, the individual components of the system will be easier to be dealt with than the whole system.	Def. 2.26
25	Explicit embodiment	To deal with the implicitness and inexpressiveness in software engineering by introducing more powerful descriptive means at a higher level of abstraction and precision.	Def. 2.27
26	Establishing theoretical foundations	To elicit rigorous theories and generic laws once there are a wide variety of observed phenomena and alternative practices.	Def. 2.28
27	Architecture and behavior modeling	To understand software system models are a hybrid model where both architectures and behaviors should be coherently described.	Def. 2.29
28	Standardization	To integrate, regulate, unify, and optimize existing principles, best practices, and industrial norms into standards.	Def. 2.30
29	Systems engineering	To adopt system science theories and approaches to deal with complicated architectures and behaviors of software.	Def. 2.31
30	Engineering organization	To recognize that the organization issue is as important as that of pure technical and the cognitive issues in software engineering.	Def. 2.32
31	Cognitive engineering	To be aware that the cognitive complexity is the dominate problem in almost all processes of software design, implementation, and maintenance, which should be tackled by cognitive informatics theories.	Def. 2.33

Appendix D

MODELS OF ENTITIES AND STRUCTURES OF SOFTWARE ENGINEERING

No	Model	Description	Remark
1	IME	The *Information-Matter-Energy* (IME) model is a general worldview, which reveals that human beings are living in a dual world, which can be modeled by *information, matter*, and *energy*.	Def. 1.1
2	Software	The product properties (Sec. 1.2.1.2) The mathematical properties (Sec. 1.2.1.1) The Information properties (Def. 1.4, Def. 7.12) The intelligent behavioral properties (Def. 9.5) The cognitive complexity properties (Theorem 10.14) The system properties (Table 10.1)	Def. 1.3
3	Software engineering	A discipline that adopts engineering approaches, such as established methodologies, processes, measurement, tools, standards, organisation methods, management methods, quality assurance systems and the like, in the development of large-scale software seeking to result in high productivity, low cost, controllable quality, and measurable development schedule.	Def. 1.5
		An engineering discipline that studies the nature of software, approaches and methodologies of large-scale software development, and the theories and laws behind software behaviors and software engineering practices.	Def. 1.6
4	HAMSD	The abstract levels of cognitive information of both the objects and their behaviors can be divided into five levels such as those of *analogue objects, diagrams, natural languages, professional notations,* and *mathematics.*	Def. 1.7
5	SECM	The software engineering constraint model	Fig. 1.4

6	Cumulative relation ®	An ordered list of embedded relations where a relation R_{ij}, $j = i + 1$, $1 \le i < n\text{-}1$, $1 < j \le n$, is related to all previous relations R_{12} through $R_{i\text{-}1,j}$, i.e.: $®(s_1, s_2, ..., s_n) = (\ldots ((R_{12}) \text{ o } R_{23}) \text{ o } \ldots) \text{ o } R_{n\text{-}1,n}$ $= (\ldots ((s_1 R_{12} s_2) R_{23} s_3) \ldots s_{n\text{-}1}) R_{n\text{-}1,n} s_n,$ $s_i \in \Xi, R_{ij} \in \mathfrak{R}$ where Ξ is a set of predefined instructions in a given programming language, and $\mathfrak{R}$ a set of designated compositional rules in the same language.	Def. 4.21
7	The big-R notation	A generic mathematical operator that is used to denote: (a) a set of *repetitive* behaviors, or (b) a finite set of recurring architectural constructs in computing, in the following forms: (a) $\mathop{\mathrm{R}}\limits_{\exp \mathbf{BL}=\mathbf{T}}^{\mathbf{F}} P$ (b) $\mathop{\mathrm{R}}\limits_{i\mathbf{N}=1}^{n} P(i\mathbf{N})$ where **BL** and **N** are the type suffixes of Boolean and natural numbers, respectively; **T** and **F** are the Boolean constants true and false, respectively.	Def. 4.59
8	A process P	A composed component of *n* meta statements s_i and s_j, $1 \le i < n$, $j = i + 1$, according to certain composing relations r_{ij}, i.e.: $P = (...(((s_1)\ r_{12}\ s_2)\ r_{23}\ s_3)\ ...\ r_{n-1,n}\ s_n)$ where r_{ij} is a set of process relations as defined in RTPA.	Def. 4.64
9	Cumulative Relational Model (CRM) of processes	A *process P* is the basic unit of an applied computational behavior that is composed by a set of statements s_i, $1 \le i \le n\text{-}1$, with left-associated cumulative relations, , i.e.: $P = \mathop{\mathrm{R}}\limits_{i=1}^{n-1} (s_i\ r_{ij}\ \mathrm{s}_j), j = i+1$ $= (...(((s_1)\ r_{12}\ \mathrm{s}_2)\ r_{23}\ \mathrm{s}_3)\ ...\ r_{n-1,n}\ \mathrm{s}_n)$ where $s_i \in \mathfrak{P}$ and $r_{ij} \in \mathfrak{R}$.	Theorem 4.3
10	The structure of RTPA	An algebraic software engineering notation system encompassing six subsystems as follows: RTPA ≙ Meta processes || Process relations || System architecture models || Primary types || Abstract dada types || Specification refinement schemes	Def. 4.66
11	Primary types	The *RTPA type system* $\mathfrak{T}$ encompasses 17 primitive types	Theorem

	of RTPA	elicited from fundamental computing needs, i.e.: $\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST},$ $@_e\mathbf{S}, @_t\mathbf{TM}, @_{int}\odot, \circledS_s\mathbf{BL}\}$	4.4
12	Meta processes of RTPA	The *RTPA meta process system* $\mathfrak{P}$ encompasses 17 fundamental computational operations elicited from the most basic computing needs, i.e.: $\mathfrak{P} = \{:=,$ � $, \Rightarrow, \Leftarrow, \nLeftarrow, \gtrdot, \lessdot, \vert\gtrdot, \vert\lessdot, @, \triangleq, \uparrow, \downarrow,$ $!, \otimes, \boxtimes, \S\}$	Theorem 4.6
13	Processes relations of RTPA	The *RTPA process relation system* $\mathfrak{R}$ encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs, i.e.: $\mathfrak{R} = \{\rightarrow, \curvearrowright, \vert, \vert\ldots\vert\ldots, R^*, R^+, R^i, \circlearrowleft, \rightarrowtail,$ $\Vert, \oiint, \vert\vert\vert, \gg, \lightning, \hookrightarrow_t, \hookrightarrow_e, \hookrightarrow_i\}$	Theorem 4.7
14	The express power of RTPA	The total number of the possible computational operations $\mathcal{N}$ in RTPA is a set of combinations between two arbitrary meta processes $\mathbb{P}_1, \mathbb{P}_2 \in \mathfrak{P}$ composed by each of the process relations $\mathbb{R} \in \mathfrak{R}$ in RTPA, i.e.: $\mathcal{N} = \#\mathfrak{R} \bullet C^2_{\#\mathfrak{P}}$ $= 17 \bullet \frac{17!}{2!(17\text{-}2)!}$ $= 17 \bullet 136$ $= 2{,}312$	Theorem 4.8
15	A system model in RTPA	An RTPA system model, §(*SysID***ST**), encompasses the following three subsystems, i.e.: §(*SysID***ST**) ≜ SysID**ST**.Architecture $\Vert$ SysID**ST**.StaticBehaviors $\Vert$ SysID**ST**.DynamicBehaviors where **ST** is the system type suffix.	Def. 4.105
16	Component Logical Model (CLM)	An abstract model of a system architectural component that represents a hardware interface, an internal logical model, a data structure, and/or a common control structure of a system.	Def. 4.106
17	Finite State Machine (FSM)	An FSM is a 5-tuple, i.e.: $FSM \triangleq (\Sigma, S, s, T, \delta)$	Def. 5.1
18	A Turing	A TM is a 6-tuple, i.e.:	Def. 5.4

	Machine (TM)	$TM \triangleq (\Sigma, S, s, H, M, \delta)$	
19	von Neumann Architecture (VNA)	A VNA of computers is a 5-tuple, i.e.: $VNA \triangleq (ALU, CU, M, I/O, B)$	Def. 5.7
20	Wang Architecture (WA) of computers	A WA computer is a *cognitive machine* with a parallel structure encompassing an *Inference Engine* (IE) and a *Perception Engine* (PE), i.e.: $WA \triangleq$ (IE || PE) = (KMU // The *Knowledge* Manipulation Unit || BMU // The *Behavior* Manipulation Unit || EMU // The *Experience* Manipulation Unit || SMU // The *Skill* Manipulation Unit) || (BPU // The *Behavior* Perception Unit || EPU // The *Experience* Perception Unit)	Def. 5.10 Fig. 5.7
21	System memory model	MEM**ST** is a system architectural type **ST** with a finite linear space, i.e.: $\text{MEM}\mathbf{ST} \triangleq [addr_1\mathbf{H} \ldots addr_2\mathbf{H}]\mathbf{RT}$	Def. 5.21
22	System I/O port model	PORT**ST** is a system architectural type **ST** with a finite linear space, i.e.: $\text{PORT}\mathbf{ST} \triangleq [ptr_1\mathbf{H} \ldots ptr_2\mathbf{H}]\mathbf{RT}$	Def. 5.22
23	A type rule	An assertion of the validity of the conclusion of a judgment on a type $\Theta_t \vdash A$ based on the inference of a number of *n* premise judgments $\Theta_t \vdash A_t$, $0 \le p \le n$, denoted by the following convention: $\frac{Premise(s)}{Conclusion} = \frac{\Theta_t \vdash A_1, ..., \Theta_t \vdash A_n}{\Theta_t \vdash A}$ where the conclusion holds *iff* all of the premises are satisfied.	Def. 5.32
24	Abstract Data Type (ADT)	An ADT is a logical model of data objects, which defines both the logical architecture and valid operations of the data object, with the following schema: ADT_ID**ST** ≜ ADT_ID**S** :: (Architecture || Static behaviors || Dynamic behaviors)	Def. 5.37
25	Basic Control Structures	A set of essential flow control mechanisms that are used	Def. 5.39

	(BCS's)	for constructing logical architectures of software systems, i.e.: $\text{BCS's} \subseteq \mathfrak{R} = \{\rightarrow, \vert, \vert\ldots\vert\ldots, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \Vert\ (\oiint), \lightning\}$	
26	The generic mathematical model of programs	A software system or a program $\wp$ is a set of complex embedded cumulative relational processes P_k dispatching by system-level events e_k, i.e.: $\wp = \mathop{R}\limits_{k=1}^{m}(@e_k \mathbf{S} \hookrightarrow P_k)$ $= \mathop{R}\limits_{k=1}^{m} [@e_k \mathbf{S} \hookrightarrow \mathop{R}\limits_{i=1}^{n-1}(s_i(k)\ r_{ij}(k)\ \mathrm{s}_j(k))], j = i+1$	Theorem 5.7
27	A class	A dynamic construct in object-oriented programming to build hierarchical architectures of a system as given in Eqs. 5.72 through 5.74.	Def. 5.55
28	A generic software pattern	A generic pattern is formally described by the four-level hierarchical model, as shown in Fig. 5.26, known as the *interfaces, implementations, instantiations*, and *associations* among the interfaces, implementations, and instantiations.	Def. 5.63
29	The Generic Computing System (GCS)	GCS, §, is an abstract logical model of the executing platform of a target machine denoted by a set of parallel or concurrent computing resources and processes as modeled in Eq. 5.87.	Def. 5.65
30	RTOS state transition diagram	Refer to Fig. 5.42.	Fig. 5.42
31	Deductive Grammar of English (DGE)	Refer to Figs. 6.3 and 6.4.	Figs. 6.3/6.4
32	Type 0 grammar G$_0$	A grammar that has no restrictions on its productions.	Def. 6.22
33	Type 1 grammar G$_3$	A grammar that satisfies the following conditions: $\forall p \in G_1,\ p\colon \alpha \rightarrow \varnothing \vee (p\colon \alpha \rightarrow \beta \Rightarrow \vert\alpha\vert \le \vert\beta\vert)$	Def. 6.23
34	Type 2 grammar G$_3$	A grammar that satisfies the following conditions: $\forall p \in G_2,\ p\colon A \rightarrow \beta$	Def. 6.24
35	Type 3 grammar G$_3$	A grammar that satisfies the following conditions: $\forall p \in G_3,\ p\colon s_0 \rightarrow \varnothing \vee p\colon A \rightarrow a \vee p\colon A \rightarrow aB$ where s_0 is the start symbol, A and B are nonterminals, and a is a single terminal.	Def. 6.25

36	Context-sensitive grammar G_s	A grammar that is constrained by the following condition: $\forall p \in G_s,\ p\text{: } \alpha A\alpha' \to \alpha\beta\alpha'$ where $\alpha A\alpha'$ is the *context*, and A is a nonterminal symbol that can be replaced in the given context.	Def. 6.26
37	Context-free grammar G_f	A grammar that is constrained by the following condition: $\forall p \in G_f,\ p\text{: } A \to \beta$ where p is context-independent.	Def. 6.27
38	Regular grammar G_r	A grammar that is constrained by the following condition: $\forall p \in G_r,\ p\text{: } s_0 \to \varnothing \vee p\text{: } A \to a \vee p\text{: } A \to aB$	Def. 6.28
39	LL(k) grammar	A class of context-free grammars, where the first *L* defines that the parsing is from *l*eft to right, and the second *L* specifies that the next production is derived by *l*eft-most derivation, and *k*, $k \geq 1$, denotes that at most *k*-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.	Def. 6.30
40	LR(k) grammar	A class of context-free grammars, where the letter *L* defines that the parsing is from *l*eft to right, and the letter *R* specifies that the next production is derived by *r*ight-most derivation in reverse, and *k*, $k \geq 1$, denotes that at most *k*-symbol looking ahead into the unmatched part of the input string is required in order to uniquely determine the next production.	Def. 6.31
41	Relations among grammars	Summarize the following grammars and their relationships in a table basis of • *Chomsky grammars* type G_0, G_0, G_0, and G_3; • The *context-sensitive* Grammar G_s, the *context-free* grammar G_f, and the *regular* grammar G_r; • The LL(*k*) grammar and the *LR*(*k*) grammar	Corollaries 6.1 and 6.2
42	Extended Backus-Naur form (EBNF)	EBNF is a 5-tuple: $EBNF \triangleq (\Sigma, T, V, P, S')$	Def. 6.37
43	The semantic environment Θ	The run-time behavioral space Ω projected onto the Cartesian plane determined by *T* and *S*, i.e.: $\Theta = \frac{\partial^2\Omega}{\partial t\, \partial s},\ t \in T \wedge s \in S$ $= \frac{\partial^2\Omega}{\partial t\, \partial s}(OP, T, S)$ $= T \times S$ where, *T* is a finite set of discrete steps of program execution, *S* is a finite set of memory locations or their	Def. 6.60

		logical representations by identifiers of variables.	
44	Deductive semantics of a statement p, θ(p)	A double partial differential of the semantic function $f_\theta(p)$ on executing steps T and the sets of variables S on Θ, i.e.: $\theta(p) = \frac{\partial^2}{\partial t\,\partial s} f_\theta(p)$ $= \mathop{R}\limits_{i=0}^{\#T(p)} \mathop{R}\limits_{j=1}^{\#S(p)} v_p(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{\#\{s_1, s_2, ..., s_m\}} v_p(t_i, s_j)$ $= \left(\begin{array}{ccccc} & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\ \mathbf{t_0} & v_{01} & v_{02} & \cdots & v_{0m} \\ (\mathbf{t_0}, \mathbf{t_1}] & v_{11} & v_{12} & \cdots & v_{1m} \end{array}\right)$ where t denotes the discrete time immediately before and after the execution of p during $(t_0, t_1]$, and # is the *cardinal calculus* that counts the number of elements in a given set, i.e., $n = \#T(p)$ and $m=\#S(p)$.	Def. 6.63
45	Deductive semantics of a process P, θ(P)	A double partial differential of the semantic function $f_\theta(P)$ on the sets of variables S and executing steps T on Θ, i.e.: $\theta(P)= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P)$ $= \mathop{R}\limits_{k=1}^{n-1} \{[\frac{\partial^2}{\partial t\,\partial s} f_\theta(P_k)]\ r_{kl}\ [\frac{\partial^2}{\partial t\,\partial s} f_\theta(P_l)]\}, l = k+1$ $= \mathop{R}\limits_{k=1}^{n-1} \{[\mathop{R}\limits_{i=0}^{\#T(P_k)} \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)]\ r_{kl}\ [\mathop{R}\limits_{i=0}^{\#T(P_l)} \mathop{R}\limits_{j=1}^{\#S(P_l)} v_{P_l}(t_i, s_j)]\}$ $= \left(\begin{array}{ccccc} \mathbf{V_{P_1}} & & & & \mathbf{V_G} \\ & \mathbf{V_{P_2}} & & & \mathbf{V_G} \\ & & \ddots & & \vdots \\ & & & \mathbf{V_{P_{n-1}}} & \mathbf{V_G} \end{array}\right)$ where $\mathbf{V}_{\mathbf{P}_k}$, $1 \le k \le n\text{-}1$, is a set of values of local variables that belongs to processes P_k, and $\boldsymbol{V_G}$ is a finite set of values of global variables.	Def. 6.65
46	Deductive semantics of a program ℘, $\theta(\wp)$	A combination of the semantic functions of all processes $\theta(P_k)$, $1 \le k \le n$, on Θ, i.e.:	Def. 6.66

		$\theta(\wp) = \mathop{R}\limits_{k=1}^{\#K(\wp)} \frac{\partial^2}{\partial t\, \partial s} f_\theta(\wp)$ $= \mathop{R}\limits_{k=1}^{\#K(\wp)} \theta(P_k)$ $= \mathop{R}\limits_{k=1}^{\#K(\wp)} [\mathop{R}\limits_{i=0}^{\#T(P_k)} \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j)]$ where $\#K(\wp)$ is the number of processes or components in the program.	
47	Information (classic informatics)	A weighted probabilistic measure of the variability of messages (signals) that is expected from a message source via a transmission channel. The *total information variability* transmitted by a source or sender, *I,* is the weighted sum of the probability of all its *n* possible signs I_i, $1 \le i \le n$, known as the alphabet, in the message, i.e.: $I = \sum_{i=1}^{n} p_i \bullet I_i$ $= \sum_{i=1}^{n} p_i \bullet \log_2 \frac{1}{p_i}$ $= - \sum_{i=1}^{n} p_i \bullet \log_2 p_i \quad [\text{bit}]$	Def. 7.1 Def. 7.2
48	Information (contemporary informatics)	Any property or attribute of the natural world that can be generally abstracted, quantitatively represented, and mentally processed. The *measurement of information*, I_k, is defined by the cost of code to abstractly represent a given size of internal message *M* in the brain in a digital system based on *k*, i.e.: $I_k = f : M \to S_k$ $= \lceil \log_k M \rceil$ where I_k is the content of information in a *k*-based digital system, and S_k the measurement scale based on *k*. The unit of I_k is the number of *k*-based digits.	Def. 7.8 Theorem 7.1
49	Information (cognitive informatics)	Abstract artifacts and their relations that can be modeled, processed, stored, and processed by human brains. The *measurement of cognitive information*, I_k, is defined by the cost of code to abstractly represent a given size of internal message *X* in the brain in a digital system based on *k*, i.e.: $I_k = f : X \to S_k = \lceil \log_k X \rceil$	Def. 7.10 Def. 7.11

		where I_k is the content of information in a k-based digital system, and S_k the measurement scale based on k.	
50	The transformability between I-M-E	According to the IME model, the three essences of the world are predicated to be transformable between each other as described by the following generic functions f_1 to f_6: $I = f_1(M)$ $M = f_2(I) \overset{?}{=} f_1^{-1}(I)$ $I = f_3(E)$ $E = f_4(I) \overset{?}{=} f_3^{-1}(I)$ $E = f_5(M)$ $M = f_6(E) = f_5^{-1}(E)$ where a question mark on the equal sign denotes a hypothesis on the existence of such a reverse function.	Corollary 7.3
51	Engineering Maturity Model (EMM)	The applied engineering disciplines have four maturity levels known as the levels of *emergence* (L_1), *art* (L_2), *engineering* (L_3), and *post-engineering* (L_4), *i.e.*: $EMM : L_1 \subseteq L_2 \subseteq L_3 \subseteq L_4$	Theorem 8.3
52	Abstract work organization model	The *actual workload W* of a coordinative project is a function of the average interpersonal coordination rate r and the number of labor L in the project, i.e.: $W = L \bullet T$ $= L \bullet T_1(1+h)$ $= W_1(1+h)$ $= W_1(1 + r \bullet \frac{L(L-1)}{2}) \quad [\mathrm{PM}]$ where T_1 is the *indicational duration* needed to complete the work by only one person, and W_1 is the *ideal workload* without the interpersonal overhead h or that of a single person project.	Theorem 8.4
53	The shortest duration of coordinative work	There exists the *shortest duration* T_{min} under the *optimum labor allocation* L_0 for a given ideal workload W_1 with certain interpersonal coordination rate r, i.e.: $\begin{cases} T_{\min} = \{T \mid L = L_0\} \\ \quad = \frac{1}{2} W_1(rL_0 - r + \frac{2}{L_0}) \quad [M] \\ L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil, r \neq 0 \quad [P] \end{cases}$	Theorem 8.7
54	The *pigeon diagram*	The model of actual time against number of labors in software engineering projects.	Fig. 8.5
55	Optimal work	It must be carried out in the following order for a given cooperative project:	Corollary

	organization	a) To determine the optimum labor allocation L_0 (Eq. 8.16); b) To obtain the shortest duration of the cooperative work T_{min} under L_0 (Eq. 8.15).	8.1
56	Exchange-ability from labor to time	The *exchange rate from labor to time* $\gamma_{L\sim T}$ in a cooperative work organization is determined by the ratio between the increment of time ΔT and the increment of labor ΔL, i.e.: $\gamma_{L\sim T} = \frac{\Delta T}{\Delta L} = \frac{T_1 \text{ - } T_{min}}{L_0 \text{-} L_1} \quad [\text{M/P}]$	Theorem 8.8
57	Exchange-ability from time to labor	The *exchange rate from time to labor* $\gamma_{T\sim L}$ in a cooperative work organization is determined by the ratio between the increment of labor ΔL and the increment of time ΔT, i.e.: $\gamma_{T\sim L} = \frac{\Delta L}{\Delta T} = \frac{L_0 \text{ - } L_1}{T_1 \text{ - } T_{\text{min}}} \quad [\text{P/M}]$	Theorem 8.9
58	Cognitive Models of Memory (CMM)	The architecture of human memory is parallel configured by the Sensory Buffer Memory (SBM), Short-Term Memory (STM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM), i.e.: $\text{CMM} \triangleq \text{SBM} \parallel \text{STM} \parallel \text{LTM} \parallel \text{ABM}$	Theorem 9.3
59	SBM	The functional model of SBM is a set of *queues* corresponding to each of the sensors of the brain.	Model 9.1
60	STM	The functional model of STM is a set of *stacks*.	Model 9.2
61	LTM	The functional model of LTM is *hierarchical neural clusters* with partially connected *neurons* via *synopses*.	Model 9.3
62	ABM	The functional model of ABM is a set of *parallel queues*, each of them represents a sequence of actions or a process.	Model 9.4
63	NI-Sys	A real-time natural intelligent system with an inherited operating system (thinking engine) *NI-OS* and a set of acquired life applications *NI-App*, i.e.: $\textit{NI-Sys} \triangleq \textit{NI-OS} \parallel \textit{NI-App}$ where *NI-OS* represents the inherited life functions, *NI-*	Model 9.5

		App the developed life functions, and || a parallel relation.	
64	BRAIN	The *functional model of the brain* describes the functional configuration of the brain and how the *NI-Sys* interacts with the memory system, i.e.: $$\begin{aligned} BRAIN \triangleq\ & NI\_Sys \\ & \| CMM \\ =\ & (\ NI\_OS \\ & \quad \| NI\_App \\ &) \\ & \| (\ LTM \\ & \quad \| STM \\ & \quad \| SBM \\ & \quad \| ABM \\ &) \end{aligned}$$	Model 9.6
65	Functional model of LTM	A set of *Hierarchical Neural Clusters* (HNC) with partially connected *neurons* via *synapses*.	Model 9.7
66	OAR	The *Object-Attribute-Relation* (OAR) model of LTM can be described as a triple, i.e.: $$OAR \triangleq (O, A, R)$$	Model 9.8
67	EOAR	The *Extended OAR model* states that the external world is represented by *real entities*, and the internal world by *virtual entities* and *objects*. The internal world can be divided into two layers known as the *image layer* and the *abstract layer*.	Model 9.9
68	Human memory capacity model	Assuming there are *n* neurons in the brain, and on average there are *s* connections between a given neuron and a subset of the rest of them in the form of synapses, the magnitude of the brain's memory capacity C_m can be expressed by the following mathematical model: $$\begin{aligned} C_m \triangleq\ & \mathrm{C}_{\mathrm{n}}^{s} \\ =\ & \frac{10^{11}!}{10^3!(10^{11}-10^3)!} \quad \text{[bit]} \end{aligned}$$ where *n* is the total number of neurons, and *s* the number of average partial connections between neurons via synapses.	Model 9.10
69	Framework of cognitive informatics	The theoretical framework of cognitive informatics	Fig. 9.7
70	LRMB	The layered reference model of the brain	Table 9.5
71	CIM	The *Cognitive Information Model* (CIM) classifies cognitive information into four categories, according to their types of I/O information, known as *knowledge, behavior, experience,* and *skill*, i.e.:	Def. 9.16 Table 9.6

		a) Knowledge $K: I \to I$ b) Behavior $B: I \to A$ c) Experience $E: A \to I$ d) Skill $S: A \to A$	
72	Generic forms of cognitive information	There are four categories of internal information $\mathcal{I}$ in the brain known as *knowledge* ($\mathcal{I}_k$), *behaviors* ($\mathcal{I}_b$), *experience* ($\mathcal{I}_e$), and *skills* ($\mathcal{I}_s$), i.e.: $$\mathcal{I} = (\mathcal{I}_k, \mathcal{I}_b, \mathcal{I}_e, \mathcal{I}_s)$$	Theorem 9.4
73	Abstract information representation	The *abstract objects* in the brain such as data (D), information (I), knowledge (K), and behavior (B) can be formally modeled as follows: $D \triangleq r_d : M \to S_k = \log_k M,\ k_{\min} = 2$ $I \triangleq r_i : D \to C,\ r_i \in \mathfrak{R}$ $K \triangleq r_k : (\mathop{\mathsf{X}}\limits_{i=1}^{n} C_i) \to C_{n+1},\ r_k \in \mathfrak{R}$ $B \triangleq \wp$ $= \mathop{R}\limits_{k=1}^{m} (@e_k \hookrightarrow P_k)$ $= \mathop{R}\limits_{k=1}^{m} [@e_k \hookrightarrow \mathop{R}\limits_{i=1}^{n-1} (p_i(k)\, r_{ij}(k)\, p_j(k))],\ j = i+1, r_{ij} \in \mathfrak{R}$ where C is a *concept* as given in Definition 15.3, $\mathfrak{R}$ is the set of process relations as defined in Theorem 4.7, and the behavior B is equivalent to a program $\wp$.	Def. 9.18
74	The nature of intelligence	*Intelligence* $\mathfrak{I}$ can be classified into four forms called the *perceptive intelligence* $\mathfrak{I}_{\mathrm{p}}$, *cognitive intelligence* $\mathfrak{I}_{\mathrm{c}}$, *instructive intelligence* $\mathfrak{I}_{\mathrm{i}}$, and *reflective intelligence* $\mathfrak{I}_{\mathrm{r}}$ as modeled below: $\mathfrak{I} \triangleq \mathfrak{I}_{\mathrm{p}} : D \to I$ (Perceptive) $\Vert\, \mathfrak{I}_{\mathrm{c}} : I \to K$ (Cognitive) $\Vert\, \mathfrak{I}_{\mathrm{i}} : I \to B$ (Instructive) $\Vert\, \mathfrak{I}_{\mathrm{r}} : D \to B$ (Reflective)	Theorem 9.5
75	Generic Intelligence Model (GIM)	GIM describes the mechanisms of the natural intelligence, as shown in Fig. 9.10.	Def. 9.21
76	Generic forms of learning	There are sufficiently four categories of learning $\mathcal{L}$ known as those of *knowledge* ($\mathcal{L}_{\mathrm{k}}$), *behaviors* ($\mathcal{L}_{\mathrm{b}}$), *experience* ($\mathcal{L}_{\mathrm{e}}$), and *skills* ($\mathcal{L}_{\mathrm{s}}$), i.e.: $$\mathcal{L} = (\mathcal{L}_k, \mathcal{L}_b, \mathcal{L}_e, \mathcal{L}_s)$$	Theorem 9.6

77	Relationship between the brain	It can be analogized by: *Brain* : *mind* = *computer* : *program*	Def. 9.32
78	A closed system	A *closed system* $\widehat{S}$ is a 4-tuple, i.e., $\widehat{S} = (C, R, B, \Omega)$, where • C is a nonempty set of components of the system, $C = \{c_1, c_2, \ldots, c_n\}$. • R is a nonempty set of relations between pairs of the components in the system, $R = \{r_1, r_2, \ldots, r_m\}$, $R \subseteq C \times C$. • B is a set of behaviors (or functions), $B = \{b_1, b_2, \ldots, b_p\}$. • Ω is a set of constraints on the memberships of components, the conditions of relations, and the scopes of behaviors, $\Omega = \{\omega_1, \omega_2, \ldots, \omega_q\}$.	Def. 10.3 Fig. 10.4
79	An open system	An *open system* S is a 7-tuple, i.e.: $S = (C, R, B, \Omega, \Theta)$ $= (C, R^c, R^i, R^o, B, \Omega, \Theta)$ where the extensions of entities beyond the closed system are as follows: • Θ is the environment of S with a nonempty set of components C_Θ outside C. • $R^c \subseteq C \times C$ is a set of internal relations. • $R^i \subseteq C_\Theta \times C$ is a set of external input relations. • $R^o \subseteq C \times C_\Theta$ is a set of external output relations.	Def. 10.4 Fig. 10.5
80	Taxonomy of systems	Taxonomy of Systems	Table 10.1
81	The universal system $\mathfrak{U}$	The *universe* $\mathfrak{U}$ is an infinite system with unlimited sets of components U, as well as unlimited relations R_U, behaviors B_U, and constraints Ω_U, i.e.: $\mathfrak{U} = (U, R_{U,} B_U, \Omega_U)$ where U encompasses any component c ever identifiable in the physical world, i.e., $\forall\ c, c \in U$.	Def. 10.12
82	The empty system $\mathfrak{D}$	The *empty system* $\mathfrak{D}$ is the smallest finite system in which the sets of components $C_\varnothing$, relations $R_\varnothing$, behaviors $B_\varnothing$, and constraints $\Omega_\varnothing$ are empty, i.e.: $\mathfrak{D} = (C_\varnothing, R_{\varnothing,} B_\varnothing, \Omega_\varnothing)$ $= (\varnothing, \varnothing, \varnothing, \varnothing)$	Def. 10.13

83	Size of a system	The *size of a system* S_s is the number of components encompassed in the system, i.e.: $S_s = \#C = n_c$	Def. 10.31
84	Magnitude of systems	The *magnitude of system relations* M_s is the number of asymmetric binary relations among the n_c components of the system including the reflexive relations, i.e.: $M_s = \#R$ $= n_r$ $= \#(C \times C)$ $= n_c^{\ 2}$	Def. 10.32
85	Holism complexity of systems	The *holism complexity of systems* states that within the 7-level magnitudes of systems, known as the *empty, small, medium, large, giant, immense,* and *infinite* systems, almost all systems are too complicated to be cognitively understood or mentally handled as a whole, except small systems or those decomposed into small systems.	Theorem 10.1 Table 10.2
86	A complete n-nary tree T_c	A normalized tree in which each node of T_c can have at most n children, the level k of T_c can have at most n^k nodes, and at all levels expect the leave level, have the maximum number of possible nodes.	Def.10.35
87	Generic topology of normalized systems	The generic topology of systems tends to be normalized into a hierarchical structure in the form of a complete *n*-nary tree.	Theorem 10.2
88	System Organization Tree (SOT)	An *n*-nary complete tree in which all leave nodes represent a *component* and the remainder, all nodes above the leaves, represent a *subsystem*.	Def. 10.37 Fig. 10.7
89	Hierarchical structure of software systems	A hierarchical structure of software systems	Fig. 10.20
90	Hierarchical structure of SE products	The hierarchical structure of software engineering processes and work products	Fig. 10.21
91	Big-O notation	If a function $f(x)$ has an asymptotic function $f_a(x)$, the function $f(x)$ is said to be of *order of* $f_a(x)$, denoted by: $f(x) = O(f_a(x))$ where O is known as the *big O* notation.	Def. 10.85
92	Time complexity	For a given size of a problem n, the *time complexity* of an algorithm for solving the problem is a function of the *maximum* required number of *dominate operations* $C_t(n)$, i.e.:	Def. 10.86

		$C_t(n) = O(f_a(n))$ where $O(f_a(n))$ is the *order* of the maximum number of the dominate operations $c(n)$, and $f_a(n)$ is called the *asymptotic function* of $C_t(n)$.	
93	Space complexity	The *space complexity* of an algorithm for a given problem is the maximum required space for both working memory w and target code memory o, i.e.: $C_m(n) = O(f(w+o)) \approx O(f(w))$ where w refers to the memory for data objects under processing such as input/output and intermediate variables, and o refers to the memory for executable code.	Def. 10.87
94	Symbolic complexity	The *symbolic complexity* of a software system S, $C_s(S)$, is the linear length of its statements measured in the unit of lines of code (LOC), i.e.: $C_s(S) = \sum_{k=1}^{n_c} C_s(k) \quad [\text{LOC}]$ where C_s(k) represents the complexity of component k in S.	Def. 10.88
95	Cyclomatic complexity	The *cyclomatic complexity* of a software system S, $C_m(S)$, is determined by the number of regions contained in the CFG G, $r(G)$, provided that G is connected, i.e.: $C_m(S) = r(G) = e - n + 2$ where, e is the number of edges in G representing branches and cycles, n number of nodes in G where each node is equivalent to a block of sequential code.	Def. 10.89
96	Operational complexity	The *operational complexity* of a software system S, $C_{op}(S)$, is determined by the sum of the cognitive weights of its n linear blocks composed by individual BCS's, where each block may consist of q layers of embedded BCS's, and within each of the layers there are m linear BCS's, i.e.: $C_{op}(S) = \sum_{k=1}^{n_c} C_{op}(C_k) = \sum_{k=1}^{n_c} (\prod_{j=1}^{q_k} \sum_{i=1}^{m_{k,j}} w(k,j,i)) \quad [\text{F}]$	Def. 10.90
97	Architectural complexity	The *architectural complexity* of a software system S, $C_a(S)$, is determined by the number of data objects at the system and component levels, i.e.:	Def. 10.92

		$C_a(S) = \mathrm{OBJ}(S))$ $= \sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k)$ [O] where *OBJ* is a function that counts the number of data objects in a given CLM (number of global variables) or components (number of local variables).	
98	Cognitive (functional) complexity	The *cognitive complexity of software* states that the *cognitive complexity* of a software system *S*, $C_c(S)$, is a product of the operational complexity $C_{op}(S)$ and the architectural complexity $C_a(S)$, i.e.: $S_f(S) = C_{op}(S) \bullet C_a(S)$ $= \{\sum_{k=1}^{n_C} \sum_{i=1}^{m_k} w(k,i)\} \bullet$ $\{\sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k)\}$ [FO]	Theorem 10.14
99	Cohesion	The *cohesion of a software system S*, *CH(S),* is a ratio of the system's number of internal relations $\#R^c$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$, i.e.: $CH(S) = \frac{\#R^c}{\#R^c + \#R^i + \#R^o} \bullet 100\%$ where $0\% \leq CH(S) \leq 100\%$.	Def. 10.97
100	Coupling	The *coupling of a software system S*, *CP(S),* is a ratio of the system's number of external relations $\#R^i + \#R^o$ and its total number of internal and external relations $\#R^c + \#R^i + \#R^o$, i.e.: $CP(S) = \frac{\#R^i + \#R^o}{\#R^c + \#R^i + \#R^o} \bullet 100\%$ where $0\% \leq CP(S) \leq 100\%$ and a lower value indicates a better architectural design.	Def. 10.98
101	Framework of management systems	The structure of a management system.	Fig. 11.2
102	Natural group	A working group of people with peers in which work is carried out via temporal pairwise coordination when work has to be done by any pair of the peers.	Def. 11.8
103	Managed group	A working group of people with peers and a manager, in which work is carried out via one-to-many coordination by the manager.	Def. 11.9

104	Framework of decision theories	The structure of decision theories.	Fig. 11.8
105	A decision	A *decision* d is a selected alternative $a \in \mathcal{A}$ from a nonempty set of alternatives $\mathcal{A}$, $\mathcal{A} \subseteq U$, based on a given set of criteria C, i.e.: $d = f(\mathcal{A}, C)$ $= f\!: \mathcal{A} \times C \rightarrow \mathcal{A},\ \mathcal{A} \subseteq U, \mathcal{A} \neq \varnothing$ where × represents a Cartesian product.	Def. 11.23
106	Taxonomy of decision making	Taxonomy of strategies and criteria for decision making.	Table 11.3
107	Payoff table	A *payoff table* is a 2-D matrix as shown in Table 11.4 that quantifies the utility, value or level of satisfaction, u_{ij}, for each given pair of alternative a_i and situation s_j, where $1 \leq i \leq n$, and $1 \leq j \leq k$.	Def. 11.27
108	Certain decision	A *decision making under certainty* d_{max} or d_{min} is a selection of an certain alternative a_i among $\mathcal{A}$ that meets a given criterion C which is either the maximum of utility or profit $max(u_i)$, and the minimum of costs or effort $min(e_i)$, i.e.: $d_{max} = f\!: \mathcal{A} \times C \rightarrow \mathcal{A}$ $= \{a_i \mid max\ (u_i) \wedge a_i \in \mathcal{A}\}$ or $d_{min} = f\!: \mathcal{A} \times C \rightarrow \mathcal{A}$ $= \{a_i \mid min\ (e_i) \wedge a_i \in \mathcal{A}\}$	Def. 11.28
109	Optimistic decision	An *optimistic decision making under uncertainty* $d_{maximax}$ or $d_{minimin}$ yields a decision with the *maximum-maximum* strategy for utility or a *minimum-minimum* strategy for cost, respectively, i.e.: $d_{maximax} = f\!: \mathcal{A} \times C \rightarrow \mathcal{A}$ $= \{a_i \mid max\ (max\ (u_{ij} \mid 1 \leq i \leq n) \mid 1 \leq j \leq k)$ or $d_{minimin} = f\!: \mathcal{A} \times C \rightarrow \mathcal{A}$ $= \{a_i \mid min\ (min\ (u_{ij} \mid 1 \leq i \leq n) \mid 1 \leq j \leq k\ )\}$	Def. 11.30
110	Pessimistic decision	A *pessimistic decision making under uncertainty* $d_{maximin}$ or $d_{minimax}$ yields a decision with the *maximum-minimum* strategy for utility or a *minimum-maximum* strategy for cost, i.e.: $d_{maximin} = f\!: \mathcal{A} \times C \rightarrow \mathcal{A}$	Def. 11.31

		$= \{a_i \mid max\ (min\ (u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k)\}$ or $d_{minimax} = f\!: \mathcal{A} \times C \to \mathcal{A}$ $= \{a_i \mid min\ (max\ (u_{ij} \mid 1 \le i \le n) \mid 1 \le j \le k)\}$	
111	Uncertain minimum regret decision	A *minimum regret decision making under uncertainty* $d_{minimax}$ yields a decision with the *minimum-maximum regret* strategy for utility gain or cost save, i.e.: $d_{minimax} = f\!: \mathcal{A} \times C \to \mathcal{A}$ $= \{a_i \mid min\ (max\ (r_{ij} \mid 1 \le i \le n)\}$	Def. 11.33
112	Risky decision with max. expected utility	A *decision making under risk with maximum expected utility* d_{maxEU} yields a decision with the *maximum expected utilities* of all alternatives, i.e.: $d_{maxEU} = f\!: \mathcal{A} \times C \to \mathcal{A}$ $= \{a_i \mid max\ (EU_i \mid 1 \le i \le n)\}$	Def. 11.36
113	Risky decision with max. utility of max. probability	A *decision making under risk with maximum utility of maximum probability* $d_{maximax\text{-}p}$ yields a decision with the *maximum* utility of the *maximum* probability of outcome of all alternatives, i.e.: $d_{maximax\text{-}p} = f\!: \mathcal{A} \times C \to \mathcal{A}$ $= \{a_i \mid max\ (u_{ij} \mid (max\ (p_j \mid 1 \le j \le \text{k})), 1 \le i \le n\}$	Def. 11.37
114	A formal game	A *formal game* G is a 4-tuple, i.e., $G = (P, D, M, S)$, where • P is a finite set of *players* $P = \{p_1, p_2, \ldots, p_n\}$, and n is the number of players, $n \ge 2$. • D is a finite set of *decisions* for certain *moves*, $D = \{d_1, d_2, \ldots, d_k\}$, $k \ge 1$. All players in G have the same number of alternative decision. • M is a finite set of *matches* between player, $M = \{m_1, m_2, \ldots, m_q\}$, $q \ge 1$. • S is a finite set of cumulated *scores* for each player, $S = \{s_1, s_2, \ldots, s_n\}$.	Def. 11.39
115	Zero-sum game	A *zero-sum game* is a game where the total scores of all n players in the game is zero, i.e.: $\sum_{i=1}^{n} s_i = 0$	Def. 11.43
116	Nonzero-sum game	A *nonzero-sum game* is a game where the total scores of all players in the game is a positive nonzero value, i.e.:	Def. 11.44

		$\sum_{i=1}^{n} s_i > 0$	
117	A decision grid	The *formal model of a decision grid DG* is a 4-tuple, i.e., $DG = (T, D, E, S)$, where • T is a finite or infinite set of *trials* $T = \{t_1, t_2, \ldots, t_n\}$, and n is the time points of trials where n may be infinitive. • D is the *decision distance* of a series of decision trials, $D = t_i - t_0 = t_i$, $1 \leq i \leq n$. • E is the effort of a specific trial towards the success state in the grid, $0 \leq E \leq n$. • S is a finite or infinite set of *success states* of the grid, $S = \{s_1, s_2, \ldots, s_k\}$, $1 \leq k \leq n$.	Def. 11.46
118	Quality	*Quality* Q is a generic and collective attribute of a product, a service, or a system that is proportional to both its average utility U and the available duration T of the utility, i.e.: $Q = U \bullet T$ [Fh] where the unit of utility is *function* (F), and the unit of duration is *hour* (h), and these result in the unit of quality as *Function-hour* or shortly Fh.	Def. 11.54
119	Integrated quality	The *integrated quality* with dynamic utility, $Q(t)$, is an integral of the utility function $U(t)$ over the entire lifecycle of the utility [0, T], i.e.: $Q(t) = \int_0^T U(t)dt$ $= \int_0^T U(1 - e^{t-T})dt$ $= Q - U(1 - e^{-T})$ [Fh] where U is the initial quality of the product, service, or system.	Lemma 11.12
120	Benefit of quality	The *benefit of a product or a system B* is the quality gained per unit cost (C) in terms of resources, labor, and time, i.e.: $B = \frac{Q}{C}$ $= \frac{U \bullet T}{C}$ [Fh/\$]	Def. 11.57
121	Generic quality system	A generic quality control system.	Fig. 11.19

122	Equilibrium model of market systems	A negative feedback system, in which the increase or decrease of price in the market will result in a negated feedback, and so do the changes of quantities of demands and supplies on prices, both which intend to resist the tendency of deviating from the current equilibrium.	Def. 12.7
123	*E*(*D*+) mode	The reactions of the equilibrium mechanism to an event of demand increase, *E*(*D*+), can be described by the following chain of reactions: $E(D+) = D\uparrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\uparrow & \rightarrow \\ \rightarrow S\uparrow\ \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.1
124	*E*(*D*-) mode	The reactions of the equilibrium mechanism to an event of demand decrease, *E*(*D*-), are formally described as follows: $E(D\text{-}) = D\downarrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\downarrow & \rightarrow \\ \rightarrow S\downarrow\ \rightarrow P\uparrow & \rightarrow \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.2
125	*E*(*S*+) mode	The reactions of the equilibrium mechanism to an event of supply increase, *E*(*S*+), are formally described as follows: $E(S+) = S\uparrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\downarrow & \rightarrow \\ \rightarrow S\uparrow\ \rightarrow P\uparrow & \rightarrow \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.3
126	*E*(*S*-) mode	The reactions of the equilibrium mechanism to an event of supply decrease, *E*(*S*-), can be formally described as follows: $E(S\text{-}) = S\downarrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\uparrow & \rightarrow \\ \rightarrow S\downarrow\ \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.4
127	*E*(*D*+, *S*+) mode	The reactions of the equilibrium mechanism to an compound event of demand/supply increases, *E*(*D*+, *S*+), are described as follows: $E(D+,\ S+) = \left\langle \begin{array}{l} D\uparrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\uparrow & \rightarrow \\ \rightarrow S\uparrow\ \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \\ S\uparrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\downarrow & \rightarrow \\ \rightarrow D\uparrow\ \rightarrow P\uparrow & \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.5
128	*E*(*D*+, *S*-) mode	The reactions of the equilibrium mechanism to an compound event of demand increase/supply decrease, *E*(*D*+, *S*-), are described as follows: $E(D+,\ S\text{-}) = \left\langle \begin{array}{l} D\uparrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\uparrow & \rightarrow \\ \rightarrow S\uparrow\ \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \\ S\downarrow\ \rightarrow \left\langle \begin{array}{ll} \rightarrow P\uparrow & \rightarrow \\ \rightarrow D\downarrow\ \rightarrow P\downarrow & \rightarrow \end{array} \right\rangle \end{array} \right\rangle \Rightarrow P'_e$	Mode 12.6
129	*E*(*D*-, *S*+) mode	The reactions of the equilibrium mechanism to an compound event of demand decrease/supply increases, *E*(*D*-, *S*+), are described as follows:	Mode 12.7

		$E(D\text{-},\ S+) = \left\langle \begin{array}{l} D\downarrow\ \to \left\langle \begin{array}{ll} \to P\downarrow & \to \\ \to S\downarrow & \to P\uparrow \to \end{array}\right\rangle \\ S\uparrow\ \to \left\langle \begin{array}{ll} \to P\downarrow & \to \\ \to D\uparrow & \to P\uparrow \to \end{array}\right\rangle \end{array}\right\rangle \Rightarrow P'_e$	
130	*E*(*D*-, *S*-) mode	The reactions of the equilibrium mechanism to an compound event of demand/supply decreases, *E*(*D*-, *S*-), are described as follows: $E(D\text{-},\ S\text{-}) = \left\langle \begin{array}{l} D\downarrow\ \to \left\langle \begin{array}{ll} \to P\downarrow & \to \\ \to S\downarrow & \to P\uparrow \to \end{array}\right\rangle \\ S\downarrow\ \to \left\langle \begin{array}{ll} \to P\uparrow & \to \\ \to D\downarrow & \to P\downarrow \to \end{array}\right\rangle \end{array}\right\rangle \Rightarrow P'_e$	Mode 12.8
131	Equilibrium behaviors of market systems	Adaptive equilibrium behaviors of market systems.	Table 12.1
132	Dynamic value of money	The *dynamic value of money* or an asset, *V*(*t*), is its present worth *P* projected at a given point of time *t* for a given average or predicated interest rate *i* during [0, *t*], i.e.: $V(t) = f(P, i, t)$	Def 12.17
133	Linear depreciation of assents	Assume an asset provides an equal amount of utility or service in each year of its life-span *n*, the *linear depreciation* of the asset in each year *D* is: $D = \frac{P\text{ - }S}{n}$ where *P* is the *initial value* of the asset, and *S* the *salvage value* by the year end of *n*.	Def. 12.19
134	Benefit-cost ratio	*Benefit-cost ratio BC* of a project is a ratio between the total benefit *B* and the total cost *C*, i.e.: $\begin{aligned} \mathrm{BC} &= \frac{B}{C} \\ &= \frac{B}{C_0 + C'} \end{aligned}$	Def. 12.25
135	Elements of software costs	Elements of software costs.	Table 12.3
136	Elements of software revenues	Elements of software revenues.	Table 12.4
137	FEMSEC	The *Formal Economic Model of Software Engineering Cost* (FEMSEC) states that, on the basis of the workload-	Theorem 12.3

		driven project organization laws (Theorems 8.4 and 8.7), the expected project cost C can be determined optimally with the optimal labor allocation L_0 and the shortest duration T_{min} in the following 6 steps: 1) Estimate the project size $\overline{S_p}$ 2) Determine the ideal workload W_1 3) Allocate the optimal labor L_0 4) Determine the shortest duration T_{min} 5) Determine the expected workload W 6) Determine the expected project cost C	Fig. 12.12
138	COCOMO	The *cost factors* of software projects identified in COCOMO are software size, effort, duration, and multiple cost drivers. Their relationships are perceived as follows: $Cost = f(size, effort, duration, cost\ drivers)$	Def. 12.33
139	COCOMO II	The *effort* E of a software project in COCOMO II is estimated by the following empirical approximation, i.e.: $E = 2.94\ EAF \bullet (kSLOC)^E$ [PM] where EAF stands for *effort adjustment factor* derived from the 17 cost drivers, E is an *exponent* determined by the five scale drivers, and the unit of project effort is person-month (PM).	Def. 12.36
140	Overtaken time of legacy maintenance costs	The *overtaken time* t_o in which the maintenance cost exceeds the development cost in a software development organization can be determined using the following expression, i.e.: $t_0 = \{t \mid C_m = C_d\}$ $\approx \lfloor n \rfloor + \dfrac{C_m(n) - C_m(\lfloor n \rfloor)}{C_m(\lceil n \rceil) - C_m(\lfloor n \rfloor)}$ [year]	Lemma 12.5
141	Group	*A group* is a formal or informal social unit formed by two or more persons working towards a particular purpose.	Def. 13.3
142	Organization	An *organization* is a formal and stable social unit formed by one or more groups of people working towards a particular purpose.	Def. 13.4
143	Society	*A society* is the community of people that members of it are geographically connected and socially integrated with common customs, organizations, and values.	Def. 13.6
144	Social relations	A *social relation* R is a function between two or more persons, p, in a society, i.e.: $R(p) = g : p \rightarrow P$	Def. 13.8

		where P is all the individuals, $p \in P$, in the given society.	
145	Social roles	The *social roles RL* of a person p is a relation between the person p and a set of social functions F, $F \subseteq \mathfrak{F}$, i.e.: $RL(p) = f: p \to F$ where F is a subset of all defined social functions $\mathfrak{F}$.	Def. 13.10
146	Maslow hierarchy of human needs	The Maslow hierarchy of human needs is at five levels known as the needs of *physiological, safety, social, esteem,* and *self-actualization* from the bottom up.	Table 13.3
147	Human Needs Hierarchy (HNH)	The *Human Needs Hierarchy* (HNH) model is a hierarchical model that encompasses five levels of fundamental human needs known from the bottom-up as N_0 – physiological needs, N_1 – psychological needs, N_2 – cognitive needs, N_3 – social needs, and N_4 – self-expressive needs.	Def. 13.19
148	Formal model of emotion	The *strength of emotion* $\|E_m\|$ is a normalized measure of how strong a person's emotion on a scale of 0 through 4, i.e.: $0 \le \|E_m\| \le 4$	Def. 13.22
149	Formal model of motivation	The *strength of motivation M* is a normalized measure of how strong a person's motivation on a scale of 0 through 100, i.e.: $0 \le M \le 100$ where $M = 100$ is the strongest motivation and $M = 0$ is the weakest motivation.	Def. 13.25
150	Formal model of attitude	The *mode of an attitude A* is determined by both an *objective judgment* of its conformance to the social norm N and a *subjective judgment* of its empirical feasibility F, i.e.: $A = \begin{cases} 1, & N = \mathbf{T} \wedge F = \mathbf{T} \\ 0, & N = \mathbf{F} \vee F = \mathbf{F} \end{cases}$ where $A = 1$ indicates a positive attitude; otherwise, it indicates a negative attitude.	Def. 13.27
151	Formal model of Behavior	A *behavior B* driven by a motivation M_r and an attitude is a realized action initiated by a motivation M and supported by a positive attitude A and a positive decision D_a toward the action, i.e.: $B = \begin{cases} \mathbf{T}, & M_r \bullet D_a = \dfrac{2.5 \bullet \| E_m \| \bullet (E\text{-}S)}{C} \bullet A \bullet D_a > 1 \\ \mathbf{F}, & otherwise \end{cases}$	Lemma 13.5
152	Normalized	A *normalized organization tree* (OT_n) is a complete *n*-	Def. 13.43

	organization tree	nary tree in which all leave nodes represent *employees* and the remainder represent *managers*. When the leaves (employees) are not reached the maximum possible numbers in the OT, the right most leaves of it will be left open.	
153	Series work organization	A *series work organization* is a work allocation structure in which a given work is decomposed into a series of parts and each part is allocated to a person or a group.	Def. 13.46
154	Parallel work organization	A *parallel work organization* is a work allocation structure in which a given work is done repetitively or jointly by multiple persons or group.	Def. 13.47
155	Formal socialization model (FSM)	The *Formal Socialization Model* (FSM) is a relational model that describes the relationships between the basic human needs, economic structures, and social types, as shown in Fig. 13.7.	Def. 13.48
156	Behavioral space	A *human behavior B* is constituted by four basic elements known as the *object* (*O*), *action* (*A*), *space* (*S*), and *time* (*T*), i.e.: $B = (O, A, S, T)$	Def. 13.55
157	BMHE	The Behavioral Model of Human Errors (BMHEs).	Table 13.12
158	HET	The model of Human Error Tree (HET). The *random nature of human errors* in performing tasks in a group is the statistical phenomenon that the occurrences of the same errors by different individuals are most likely at different times.	Corollary 13.15 Fig. 13.9
159	Error reduction model of review	The *n-fold error reduction by reviewing* states that the *error rate of a work product* can be reduced upto *n* folds of the average error rate of individuals r_e in a group via *n*-nary *peer reviews* based on the random nature of error distributions and independent nature of error patterns of individuals, i.e.: $R_e = \prod_{k=1}^{n} r_e(k)$	Theorem 13.5
160	DTSD programming	*Distributed Time-Shared Development* (DTSD) is a software engineering methodology that geographically allocates software development work in broadly distributed time zones with a wide-area Intranet.	Def. 14.21
161	Software Maintenance Crisis (SMC)	A phenomenon that happens when the demand for software maintenance exceeds the capability that a software development organization can provide, or when the costs of legacy software maintenance predominately override the investment for new software development.	Def. 14.22

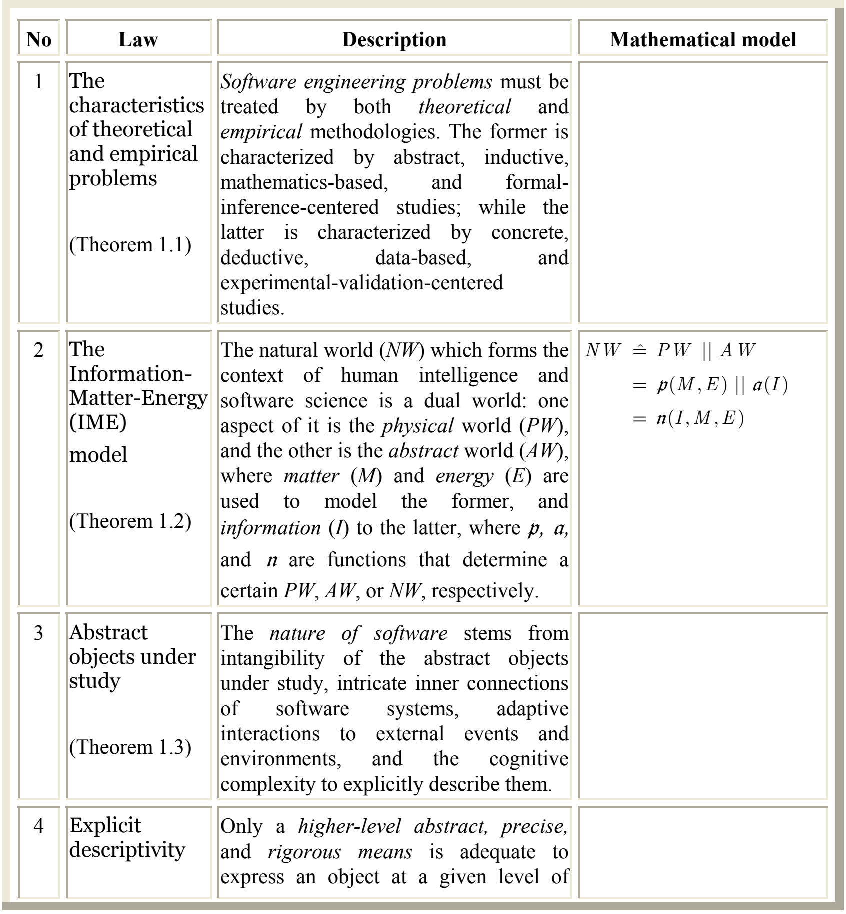
162	Framework of SE	The theoretical framework of software engineering.	Fig. 14.5
163	Impact of SE	The impact of software engineering theories on related disciplines.	Table 14.9
164	Formal knowledge	Taxonomy of formal knowledge.	Table 15.1
165	Framework of Formal Knowledge Systems (FKS)	The framework of Formal Knowledge System (FKS).	Fig. 15.1
166	Software Science	*Software science* is a branch of knowledge that studies the theoretical framework of software as instructive and behavioral information, which can be embodied and executed by generic computers in order to create expected system behaviors and machine intelligence.	Def. 15.1
167	Framework of Software Science	The Theoretical Framework of software science	Fig. 15.2
168	Denotational mathematics for SS	Denotational mathematical means for software science.	Table 15.2
169	Abstract concept	An *abstract concept* c is a 5-tuple, i.e., $c \triangleq (O, A, R^c, R^i, R^o)$, where • O is a nonempty set of object of the concept, $O = \{o_1, o_2, \ldots, o_m\}$ = ÞU, where ÞU denotes a power set of U. • A is a nonempty set of attributes, $A = \{a_1, a_2, \ldots, a_n\}$ = ÞM. • $R^c \subseteq O \times A$ is a set of internal relations. • $R^i \subseteq C' \times C$ is a set of input relations, where C′ is a set of external concepts. • $R^o \subseteq C \times C'$ is a set of output relations.	Def. 15.3
170	Concept algebra	*Concept algebra* is a new mathematical structure for the formal treatment of abstract concepts and their algebraic relations, operations, and associative rules for composing complex concepts and knowledge	Def. 15.4
171	Generic knowledge model	A *generic knowledge K* is an *n*-nary relation R_k among a set of *n* multiple concepts in *C*, i.e.: $$K = R_k : (\mathop{\text{X}}_{i=1}^{n} C_i) \rightarrow C$$ where $\bigcup_{i=1}^{n} C_i = C$, and	Def. 15.5

		$R_k \in \mathfrak{R} = \{\Rightarrow, \overset{+}{\Rightarrow}, \overline{\Rightarrow}, \tilde{\Rightarrow}, \uplus, \pitchfork, \Lleftarrow, \vdash, \mapsto\}$.	
172	Concept network	A *concept network CN* is a hierarchical network of concepts interlinked by the set of nine associations $\mathfrak{R}$ defined in concept algebra, i.e.: $CN = R_\kappa : \mathop{\mathsf{X}}\limits_{i=1}^{n} C_i \rightarrow \mathop{\mathsf{X}}\limits_{i=j}^{n} C_j$ where $R_k \in \mathfrak{R}$.	Def. 15.6
173	Taxonomy of denotational mathematics	Taxonomy of denotational mathematics for software science and engineering	Table 15.3
174	Imperative Computing (IC) behaviors	The *necessary and sufficient conditions of IC, C_{IC}*, are the possession of *event B_e, time B_t,* and *interrupt B_{int}* driven computational behaviors, i.e.: $C_{IC} = (B_e, B_t, B_{int})$	Theorem 15.2
175	Autonomic Computing (AC) behaviors	The *necessary and sufficient conditions of AC, C_{AC}*, are the possession of *goal B_g* and *inference B_{inf}* driven computational behaviors, in addition to the *event B_e, time B_t,* and *interrupt B_i* driven behaviors, i.e.: $C_{AC} = (B_g, B_{inf}, B_e, B_t, B_{int})$	Theorem 15.3
176	Imperative Computing System (ICS)	The *Imperative Computing System*, §IC, is an abstract logical model of conventional computing platforms denoted by a set of parallel or concurrent computing resources and behaviors as shown in Fig. 15.5.	Def. 15.9
177	Autonomic Computing System (ACS)	The *AC System*, §AC, is an abstract logical model of computing platform denoted by a set of parallel or concurrent computing resources and behaviors as shown in Fig. 15.6.	Def. 15.12
178	Cognitive model of ACS	The *cognitive informatics model of an AC system, ACS*, is equivalent to the high-level logical model of the brain as given in Model 9.6, i.e.: $\begin{array}{l} ACS \triangleq NI\_Sys \\ \quad \| \, CMM \\ = (\; NI\_OS \\ \quad \| \, NI\_App \\ \quad) \\ \quad \| \, (\; LTM \\ \quad\quad \| \, STM \\ \quad\quad \| \, SBM \\ \quad\quad \| \, ABM \\ \quad) \end{array}$	Def. 15.13

179	Hyper-programming	A *hyper-program* is a new type of nonlinear framework for software description and documentation that integrates software architectures, behaviors, code, and related design workproducts into a coherent and multidimensional framework by bidirectional hyperlinks.	Def. 15.14 Fig. 15.7

Appendix E

WANG'S LAWS OF SOFTWARE ENGINEERING



No	Law	Description	Mathematical model
1	The characteristics of theoretical and empirical problems (Theorem 1.1)	*Software engineering problems* must be treated by both *theoretical* and *empirical* methodologies. The former is characterized by abstract, inductive, mathematics-based, and formal-inference-centered studies; while the latter is characterized by concrete, deductive, data-based, and experimental-validation-centered studies.	
2	The Information-Matter-Energy (IME) model (Theorem 1.2)	The natural world (*NW*) which forms the context of human intelligence and software science is a dual world: one aspect of it is the *physical* world (*PW*), and the other is the *abstract* world (*AW*), where *matter* (*M*) and *energy* (*E*) are used to model the former, and *information* (*I*) to the latter, where p, a, and n are functions that determine a certain *PW*, *AW*, or *NW*, respectively.	$NW \hat{=} PW \parallel AW$ $= p(M, E) \parallel a(I)$ $= n(I, M, E)$
3	Abstract objects under study (Theorem 1.3)	The *nature of software* stems from intangibility of the abstract objects under study, intricate inner connections of software systems, adaptive interactions to external events and environments, and the cognitive complexity to explicitly describe them.	
4	Explicit descriptivity	Only a *higher-level abstract, precise,* and *rigorous means* is adequate to express an object at a given level of	

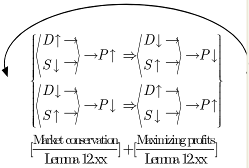
	(Theorem 1.4)	abstraction, where denotational mathematics is the top-level abstraction means.	
5	The basic constraints of SE (Theorem 1.5)	Software engineering faces the *cognitive, organizational,* and *resources constraints*.	
6	Conservative productivity (Theorem 1.7)	Software productivity is physiologically constrained by the growing speed of synaptic connections inside the brain, because before any creative artifact is generated externally, it must be created and represented physiologically inside the brain by the synaptic connections.	
7	Universal constraints (Theorem 3.1)	Both the natural world and the perceived abstract world are constrained by certain known restrictions and laws, or by those yet to be known due to both current limitations of natural resources and/or human cognitive capability.	
8	Law of causality (Theorem 3.3)	A condition must be both necessary and sufficient to qualify as a cause, where the *necessary* condition is a condition that must be present in order for the effect to occur, while the *sufficient* condition is a condition that will always produce the effect.	
9	Inclusive intelligent capability (Theorem 3.5)	*Artificial intelligence* (AI) is a subset of *natural intelligence* (NI).	$\text{AI} \subseteq \text{NI}$
10	Behavior space of software (Theorem 3.11)	The software behavior space Ω is innately three-dimensional, which can be described by a Cartesian product of computational operations *OP*, time *T*, and memory space *S*.	$\Omega = OP \times T \times S$
11	Utility of mathematics (Theorem 4.1)	*Denotational mathematics* is the means and rules to rigorously and explicitly express design notions and conceptual models on abstract architectures and complex interactive behaviors at the highest level of abstraction and in the largest scope of systems.	

12	Cumulative Relational Model (CRM) of processes (Theorem 4.3)	A *process* $\mathfrak{P}$ is the basic unit of an applied computational behavior that is composed by a set of statements s_i, $1 \le i \le n\text{-}1$, with left-associated cumulative relations, where $s_i \in \mathfrak{P}$ and $r_{ij} \in \mathfrak{R}$.	$\mathfrak{P} = \underset{i=1}{\overset{n-1}{R}} (s_i\ r_{ij}\ \mathrm{s}_j), j = i+1$ $= (...(((s_1)\ r_{12}\ \mathrm{s}_2)\ r_{23}\ \mathrm{s}_3)\ ...\ r_{n-1,n}\ \mathrm{s}_n)$
13	Express power of algebraic modeling (Theorem 4.8)	The *express power of RTPA* states that the total number of the possible computational operations $\mathcal{N}$ is a set of combinations between two arbitrary meta processes $\mathbb{P}_1, \mathbb{P}_2 \in \mathfrak{P}$ composed by each of the process relations $\mathbb{R} \in \mathfrak{R}$ in RTPA.	$\mathcal{N} = \#\mathfrak{R} \bullet C^2_{\#\mathfrak{P}}$ $= 17 \bullet \dfrac{17!}{2!(17\text{-}2)!}$ $= 17 \bullet 136$ $= 2{,}312$
14	Essential facets of software system modeling (Theorem 4.9)	Software systems can be formally specified by its *architectures, static behaviors,* and *dynamic behaviors* with multiple-level refinements.	
15	The root of computing and information science (Theorem 5.1)	The *most fundamental data object model* shared in both computing and information science is *binary digits* (bits).	
16	Domain constraints of data objects (Theorem 5.6)	Letting D_m, D_l, and D_u be the domains of *mathematical* (logical), *language defined,* and *user defined,* respectively, the following relationship between the domains of an identifier in programming is always held.	$D_u \subseteq D_l \subseteq D_m$
17	The generic mathematical model of programs (Theorem 5.7)	A software system or a program $\wp$ is a set of complex embedded cumulative relational processes P_k dispatched by system-level events e_k .	$\wp = \underset{k=1}{\overset{m}{R}} (@e_k\, \mathbf{S} \hookrightarrow P_k)$ $= \underset{k=1}{\overset{m}{R}} [@e_k\, \mathbf{S} \hookrightarrow$ $\underset{i=1}{\overset{n-1}{R}} (s_i(k)\ r_{ij}(k)\ \mathrm{s}_j(k))],$ $j = i+1$
18	Tradeoff between syntaxes and semantics (Theorem 6.1)	In the DGE system, the complexities of the syntactic rules (or grammar) C_{syn} and of the semantic rules C_{sem} are inversely proportional, i.e.:	$C_{syn} \propto \dfrac{1}{C_{sem}}$

19	Asynchronicity of program semantics (Theorem 6.2)	The semantics of a relatively timed program is invariant with the changes of executing speed, as long as any absolute time constraint is met.	
20	The least complete set of instructions in programming (Theorem 6.3)	A program is *composable* with sufficient descriptive power in a given language *iff* both the sufficient sets of meta instructions ($\mathfrak{P}$, Theorem 4.6) and compositional rules ($\mathfrak{R}$, Theorem 4.7) are rigorously defined.	
21	Informatics laws of software (Theorem 7.2)	Software architectures, behaviors, and processes are constrained by the 19 *informatics* laws of basic information properties.	
22	Conservation of basic engineering constraints (Theorem 8.2)	The three basic constraints of engineering goals known as time (*T*), costs (*C*), and utility (*U*) are conservative in a given engineering context, where both δ and *k* are a constant.	$f_t(T^{-1}) + f_c(C^{-1}) + f_u(U)$ $= k\dfrac{U}{T \bullet C}$ $= \delta$
23	Coordinative work load in engineering (Theorem 8.4)	The *actual workload W* of a coordinative project is a function of the average interpersonal coordination rate *r* and the number of labor *L* in the project, where T_1 is the *indicational duration* needed to complete the work by only one person, and W_1 is the *ideal workload* without the interpersonal overhead *h* or that of a single person project.	$W = L \bullet T$ $= L \bullet T_1(1 + h)$ $= W_1(1 + h)$ $= W_1(1 + r \bullet \dfrac{L(L-1)}{2})$ [PM]
24	Interchangeability of labor and time (ILT) (Theorem 8.6)	For a given workload *W*, labor *L* and duration *T* are transformable under the condition as given in the mathematical model.	$T = \dfrac{W}{L}$ $= \dfrac{W_1}{L}(1 + r \bullet \dfrac{L(L-1)}{2})$ $= \dfrac{W_1}{L}(\dfrac{1}{2}rL^2 - \dfrac{1}{2}rL + 1)$ $= \dfrac{1}{2}W_1(rL - r + \dfrac{2}{L})$

25	The shortest duration of coordinative work (Theorem 8.7)	There exists the *shortest duration T_{min}* under the *optimum labor allocation L_0* for a given ideal workload W_1 with a certain interpersonal coordinative rate *r*.	$\begin{cases} T_{\min} = \{T \mid L = L_0\} \\ \quad = \frac{1}{2} W_1 (rL_0 - r + \frac{2}{L_0}) \ [M] \\ L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil, \ r \neq 0 \quad [P] \end{cases}$
26	Quantitative advantage of human brain (Theorem 9.1)	The magnitude of the memory capacity of the brain is tremendously larger than that of the closest species.	
27	Qualitative advantage of human brain (Theorem 9.2)	The possession of the abstract layer of memory and the abstract reasoning capacity makes the human brain profoundly powerful on the basis of the quantitative advantage.	
28	Generic forms of information (Theorem 9.4)	There are four categories of internal information $\mathcal{I}$ in the brain known as *knowledge* ($\mathcal{I}_k$), *behaviors* ($\mathcal{I}_b$), *experience* ($\mathcal{I}_e$), and *skills* ($\mathcal{I}_s$).	$\mathcal{I} = (\mathcal{I}_k, \mathcal{I}_b, \mathcal{I}_e, \mathcal{I}_s)$
29	The nature of intelligence (Theorem 9.5)	*Intelligence* $\mathfrak{I}$ is a capability that transfers between data, information, knowledge, and behaviors known as the *perceptive intelligence* $\mathfrak{I}_p$, *cognitive intelligence* $\mathfrak{I}_c$, *instructive intelligence* $\mathfrak{I}_i$, and *reflective intelligence* $\mathfrak{I}_r$.	$\mathfrak{I} \triangleq \mathfrak{I}_p : D \rightarrow I$ (Perceptive) $\| \mathfrak{I}_c : I \rightarrow K$ (Cognitive) $\| \mathfrak{I}_i : I \rightarrow B$ (Instructive) $\| \mathfrak{I}_r : D \rightarrow B$ (Reflective)
30	Dynamic properties of neural clusters (Theorem 9.9)	The LTM is dynamic. New neurons (to represent *objects or attributes*) are assigning, and new synaptic connections (to represent *relations*) are creating and reconfiguring all the time in the brain.	
31	Establishment cycle of LTM (Theorem 9.11)	The cycle of LTM establishment requires at least 24 hours, where the 24-hour cycle includes any kind of combinations of awake, asleep, and siesta.	*LTM establishment cycle* ≥ 24 [hrs]
32	Holism complexity of systems (Theorem 10.1)	Within the 7-level magnitudes of systems, known as the *empty, small, medium, large, giant, immense,* and *infinite* systems, almost all systems are too complicated to be cognitively understood or mentally handled as a whole, except small systems or those that can be decomposed into small systems.	

33	Generic topology of normalized systems (Theorem 10.2)	Systems tend to be normalized into a hierarchical structure in the form of a complete *n*-nary tree.	
34	System gain of functionality (Theorem 10.4)	System conjunction or composition between two systems S_1 and S_2 creates *new relations* ΔR_{12} and/or *new behaviors* (functions) ΔB_{12} that are solely a property of the newly established super system *S*, which can be determined by the sizes of the two intersected component sets $\#C_1$ and $\#C_2$.	$\Delta R_{12} = \#R - (\#R_1 + \#R_2)$ $= (\#(C_1 + C_2))^2 –$ $((\#C_1)^2 + (\#C_2)^2)$ $= 2\ (\#C_1 \bullet \#C_2)$
35	System mutation (Theorem 10.5)	The gradual increment of quantity of system, i.e., ΔC or ΔR, in a system beyond the point of the critical mass Q_{cm} triggers the abrupt generation of functionality (quality) F_{cm} of the system.	
36	System gain of work (Theorem 10.6)	Work done by a system is always greater than any of its components, but must not greater than the sum of those of its components	$\begin{cases} W(S) \le \sum_{i=1}^{n} W(C_i), & \eta \le 100\% \\ W(S) > \max(W(C_i)), & C_i \in E_S \end{cases}$
37	Conservative work of equilibrium systems (Theorem 10.9)	The sum of all types of work is always zero in an equilibrium system, where $W(C_i)$ is the abstract work of a system component C_i.	$\sum_{i=1}^{n} W(C_i) = 0$
38	Conditions of system self-organization (Theorem 10.10)	The *necessary* and *sufficient* condition of self-organization is the existence of at least one minimum on the state curve of a system *f*(*x*), which satisfies the following requirements, where *f'*(*x*) and *f "*(*x*) are the first and second order derivatives of *f*(*x*) on (a, b).	$\begin{cases} f'\ (x_{min} \mid x_{min} \in\ (a,b)) = 0 \\ f''(x_{min} \mid x_{min} \in\ (a,b)) \ne 0 \end{cases}$ $\begin{cases} f'\ (x_{min} \mid x_{min} \in\ (a,b)) = 0 \\ f''(x \mid x < x_{min} \in\ (a,b)) < 0 \\ f''(x \mid x > x_{min} \in\ (a,b)) > 0 \end{cases}$
39	System synchronization (Theorem 10.11)	A system reaches its maximum utility $\vec{S}_{\max}$ when all components' efforts $\vec{S_1}$ and $\vec{S_2}$ are synchronized.	$\begin{cases} \vec{S} = \vec{S_1} + \vec{S_2} \\ \vec{S}_{\max} = \ \mid \vec{S_1} \mid + \mid \vec{S_2} \mid \end{cases}$
40	System dissimilation	Any system tends to undergo a continuous degradation that leads to the eventual loss of its designed utility and	

	(Theorem 10.12)	against its initial purposes to form the system.	
41	Cognitive complexity of software (Theorem 10.14)	The *cognitive complexity* of a software system *S*, *C_c*(*S*), is a product of the operational complexity *C_{op}*(*S*) and the architectural complexity *C_a*(*S*).	$S_f(S) = C_{op}(S) \bullet C_a(S)$ $= \{\sum_{k=1}^{n_C}\sum_{i=1}^{m_k} w(k,i)\} \bullet$ $\{\sum_{k=1}^{n_{CLM}} \text{OBJ}(CLM_k)$ $+ \sum_{k=1}^{n_C} \text{OBJ}(C_k)\}$ [FO]
42	Gain of management (Theorem 11.1)	Management is required to reduce the complexity of working group organization, to improve the efficiency of groups (*e*(*n*)), and to simplify the forms of interpersonal coordination.	$e(n) = \frac{\Delta m(n)}{C_2(n)} \bullet 100\%$ $= (1 - \frac{c_m(n)}{c_2(n)}) \bullet 100\%$ $= (1 - \frac{\text{n+1}}{n \bullet (n\text{ - }1)}) \bullet 100\%$
43	Gain of division of labor (Theorem 11.2)	The relative gain *g_r*(*k*) via division of labor in work organization is proportional to the repetitive times *k* at specialized subtask-level, where *c* is a positive constant, 1 < *c* < *e*.	$g_r(k) = \frac{E(k)\text{ - }E_d(k)}{E(k)} \bullet 100\%$ $= (1\text{-}\frac{E_d(k)}{E(k)}) \bullet 100\%$ $= (1\text{-}\frac{\sum_{i=1}^{k}\frac{1}{(\frac{e}{c})^{k\text{-}1}}}{k}) \bullet 100\%$
44	Adaptive economic equilibrium (Theorem 12.1)	A market with autonomic interactions between demands *D* and supplies *S* is a self-regulated and self-adaptive system, where any change in demand, supply, or both will be autonomously adjusted via the leverage of price *P* to an equilibrium.	 $\left\{\begin{array}{l} \left\langle\begin{array}{l} D\uparrow\rightarrow \\ S\downarrow\rightarrow \end{array}\right\rangle \rightarrow P\uparrow \Rightarrow \left\langle\begin{array}{l} D\downarrow\rightarrow \\ S\uparrow\rightarrow \end{array}\right\rangle \rightarrow P\downarrow \\ \left\langle\begin{array}{l} D\downarrow\rightarrow \\ S\uparrow\rightarrow \end{array}\right\rangle \rightarrow P\downarrow \Rightarrow \left\langle\begin{array}{l} D\uparrow\rightarrow \\ S\downarrow\rightarrow \end{array}\right\rangle \rightarrow P\uparrow \end{array}\right\}$ $[\frac{\text{Market conservation}}{\text{Lemma 12.xx}}] + [\frac{\text{Maximizing profits}}{\text{Lemma 12.xx}}]$
45	Formal Economic Model of Software Engineering Cost (FEMSEC) (Theorem 12.3)	On the basis of the workload-driven project organization laws, the expected project cost *C* can be rigorously determined with the optimal labor allocation *L₀* and the shortest duration *T_{min}* by the following 6 steps: 1) Estimate the project size $\overline{S_p}$ 2) Determine the ideal workload *W₁* 3) Allocate the optimal labor *L₀* 4) Determine the shortest duration *T_{min}* 5) Determine the expected workload *W* 6) Determine the expected project cost *C*	$\overline{S_p} = \frac{1}{6}(S_{max} + 4S_{exp} + S_{min})$[kLOC] $W_1 = \frac{\overline{S_p}}{\rho} \bullet 12$ [PM] $L_0 = \left\lceil \frac{1.414}{\sqrt{r}} \right\rceil$ $[P]$ $T_{\min} = \frac{1}{2}W_1(rL_0 - r + \frac{2}{L_0})$ $W = \frac{1}{2}W_1$ $(rL^2{}_0 - rL_0 + 2)$ [PM] $C = L_0 \bullet T_{min} \bullet C_L$ [\$]

46	Basic essences for evolution (Theorem 13.1)	The *basic evolutional needs* of mankind are to preserve both the species' biological traits via *gene pools*, and the cumulated knowledge via various *information systems*.	
47	Organizational coordination efficiency (Theorem 13.3)	The natural constraints for social organization that forces the architecture of large groups to be evolved and adapted to tree-form hierarchical structures in an organization is the need to maintain acceptable coordinating efficiency at each level of the organization tree.	
48	Time-oriented optimization for large-scale project organization (Theorem 13.4)	*Time-oriented optimization for large-scale project organization* states that in order to further reduce the shortest duration T_{min} of an entire large-scale project constrained by Theorem 8.7, the optimal form of organization is to evenly partition the whole project into *n* lightly-coupled parallel subprojects that may be conducted by independent groups with a shorter duration T^i_{min}, $1 \le i \le n$, so that an average *n*-fold time deduction can be gained.	$\overline{T^i}_{\min} = \frac{1}{n}\sum_{i=1}^{n} T^i_{\min} = \frac{1}{n}T_{\min} + \varpi$
49	The *n*-fold error reduction structure (Theorem 13.5)	The *error rate of a work product* can be reduced up to *n* folds from the average error rate of individuals r_e in a coordinative group via *n*-nary *peer reviews* based on the random nature of error distributions and independent nature of error patterns of individuals.	$R_e = \prod_{k=1}^{n} r_e(k)$
50	Power of multi-disciplinary knowledge (Theorem 14.2)	The *ratio of knowledge space* Ω_Σ between the knowledge of an expert with coherently *m* disciplinary knowledge K_Σ and that of a group of *m* experts with separated individual disciplinary knowledge K_m is shown in the mathematical model, where *n* is the number of *average knowledge objects* or concepts in the disciplines.	$\Omega_\Sigma(m,n) = \frac{K_\Sigma}{K_m} = \frac{\mathrm{C}^2_{m\bullet n}}{\sum_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{C}^2_n} = \frac{\frac{(mn)!}{2!(mn\text{-}2)!}}{\frac{m(n)!}{2!(n\text{-}2)!}} \approx \frac{(mn)^2}{mn^2} = m$

Appendix F

WANG'S FORMAL PRINCIPLES OF SOFTWARE ENGINEERING

No	Principle	Description	Mathematical model
1	Polymorphous solutions (Theorem 1.6)	The solution space *SS* of software engineering for a given problem is a product of the number of possible design options N_d and the number of possible implementation options N_i.	$SS = N_d \bullet N_i$
2	Formalization of principles (Theorem 2.1)	The empirical principles for software engineering are heuristic and data-based; while the formal principles for software engineering are rigorous and mathematics-based, which are elicited and refined from the empirical principles.	
3	Validation of abstract propositions (Theorem 3.2)	The abstract and information-based propositions and work products, such as a design or a specification of a system, are bounded by logical verifications, mathematical proofs, systematical reviews, behavioral simulations and tests, and/or in field trials.	
4	Compatible intelligent capability (Theorem 3.4)	*Natural intelligence* (NI) and *artificial intelligence* (AI) are compatible by sharing the same mechanisms of intelligent capability.	AI $\propto$ NI
5	Deductive inference	Given an arbitrary nonempty set $\mathfrak{X}$, let $p(x)$ be a proposition for $\forall x \in \mathfrak{X}$, a specific conclusion on $\exists a \in \mathfrak{X}, p(a)$ can	$\forall x \in \mathfrak{X}, p(x) \vdash \exists a \in \mathfrak{X}, p(a)$ or

	(Theorem 3.6)	be drawn as in the mathematical models.	$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash$ $(\exists a \in \mathbf{X}, p(a) \Rightarrow q(a))$
6	Inductive inference (Theorem 3.7)	If $\exists a, k, succ(k) \in \mathbf{X}$, $p(a)$ and $p(k) \Rightarrow p(succ(k))$ are three valid propositions, then a generic conclusion on $\forall x \in \mathbf{X}$, $p(x)$ can be drawn as in the mathematical models.	$((\exists a \in \mathbf{X}, p(a)) \wedge$ $(\exists k,\ succ(k) \in \mathbf{X},\ (p(k) \Rightarrow p(succ(k)))) \vdash \forall x \in \mathbf{X}, p(x)$ or $((\exists a \in \mathbf{X},\ p(a) \Rightarrow q(a)) \wedge (\exists k,\ succ(k) \in \mathbf{X},\ ((p(k) \Rightarrow q(k)) \Rightarrow (p(succ(k)) \Rightarrow q(succ(k))))) \vdash$ $\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)$
7	Abductive inference (Theorem 3.8)	Based on a general implication $\forall x \in \mathbf{X}$, $p(x) \Rightarrow q(x)$, a specific conclusion on $\exists a \in \mathbf{X}$, $p(a)$ can be drawn as in the mathematical models.	$(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x)) \vdash$ $(\exists a \in \mathbf{X}, q(a) \Rightarrow p(a))$ or $(\forall x \in \mathbf{X}, p(x) \Rightarrow q(x) \wedge r(x) \Rightarrow q(x)) \vdash (\exists a \in \mathbf{X}, q(a) \Rightarrow (p(a) \vee r(a)))$
8	Analogical inference (Theorem 3.9)	Based on a specific predicate $\exists a \in \mathbf{X}$, $p(a)$, a similar specific conclusion can be drawn *iff* $\exists x \in \mathbf{X}$, $p(x)$ as in the mathematical models.	$\exists x \in \mathbf{X}, p(x) \wedge \exists a \in \mathbf{X}, p(a) \vdash \exists b \in \mathbf{X} \wedge b \neq a, p(b)$ or $(\exists x \in \mathbf{X}, p(x) \wedge \exists a \in \mathbf{X}, p(a) \Rightarrow q(a)) \vdash (\exists b \in \mathbf{X} \wedge \mathrm{b} \neq a, p(b) \Rightarrow q(b))$
9	Necessary and sufficient conditions of software usage (Theorem 3.10)	Those that warrant the requirements for software solutions are the system behaviors of *repeatability*, *programmability*, and *run-time determinability*.	
10	Principle of abstraction (Theorem 4.2)	Given an arbitrary set X and any property p, there is a set A such that the elements of A are exactly those members of X which have the property p.	$A = \{a \mid a \in X \wedge p(a)\}$
11	Primary types of computational objects (Theorem 4.4)	The *RTPA type system* $\mathfrak{T}$ encompasses 17 primitive types elicited from fundamental computing needs.	$\mathfrak{T} = \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @_e\mathbf{S}, @_t\mathbf{TM}, @_{int}⦿, ⓢ_s\mathbf{BL}\}$
12	Type equivalence	Two types $\mathbb{T}_1$ and $\mathbb{T}_2$ are *equivalent, iff* the domain of type $\mathbb{T}_1$ is either identical	$\mathbb{T}_1(x) = \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y)$ or

	(Theorem 4.5)	to or a subset of that of $\mathbb{T}_2$.	$\mathbb{T}_1(x) \subseteq \mathbb{T}_2(y) \Rightarrow \mathbb{T}_1(x) \simeq \mathbb{T}_2(y)$
13	Meta software processes (Theorem 4.6)	The *RTPA meta process system* $\mathfrak{P}$ encompasses 17 fundamental computational operations elicited from the most basic computing needs.	$\mathfrak{P} = \{:=, \text{�}, \Rightarrow, \Leftarrow, \nLeftarrow, \gtrdot, \lessdot, \vert\gtrdot, \vert\lessdot, @, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\}$
14	Software composing rules (Theorem 4.7)	The *RTPA process relation system* $\mathfrak{R}$ encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs.	$\mathfrak{R} = \{\rightarrow, \curvearrowright, \vert, \vert\ldots\vert\ldots, R^{*}, R^{+}, R^{i}, \circlearrowleft, \rightarrowtail, \Vert, \oiint, \vert\vert\vert, \gg, \lightning, \hookrightarrow_{t}, \hookrightarrow_{e}, \hookrightarrow_{i}\}$
15	The primitive computational behaviors (Theorem 5.2)	The *most fundamental computational operations* are logical, arithmetic, and memory access operations on *bits*.	
16	Nature of requirements and specifications (Theorem 5.3)	Requirement elicitation focuses on desired functions of a system δ, while system specification focuses on the entire behavioral space of the system Ω, including both δ and the undesired but potential system transitions represented by $\overline{\delta}$ in the behavioral space.	$S_{\Omega} = \#\delta + \#\overline{\delta}$ $= \#S \bullet \#\Sigma$
17	The weaknesses of automata (Theorem 5.4)	Automata and FSMs as a system composition and modeling method built on event-driven mechanisms are inadequate to model the complete basic computational requirements, particularly the lack of the descriptive power for: a) System architectures and data objects modeling; b) Nonevent-driven transitional process modeling; c) Detailed behavioral descriptions; d) Mathematical operations and processing of complicated languages.	
18	Fundamental computational capabilities (Theorem 5.5)	The essential capabilities for computation are as follows: • A memory for storing bit information; • A simple addressing capability for accessing information in the memory; • Read/write operations for retrieving or	

		updating the memory; • A conditional and quantitative evaluation capability for interpreting the inputted information; • A stored-information-driven mechanism for determining the next step.	
19	Primitive form of information (Theorem 7.1)	The most fundamental form of information that can be represented and processed is binary digit where $k = b = 2$.	$I_b = f : M \to S_b$ $= \lceil \log_b M \rceil$ $= \lceil \log_2 M \rceil \; [bit]$
20	Relationship between a hypothesis and a theory (Theorem 8.1)	The necessary and sufficient conditions for a hypothesis $H_g(C, O, G, P, F)$ to be proven as a theory $\mathcal{T}$ are *iff* it fulfills the following criteria.	$H_g \vdash \mathcal{T}$, *iff* $C \wedge O \wedge G \wedge P \wedge F = \mathbf{T}$
21	Engineering Maturity Model (EMM) (Theorem 8.3)	The applied engineering disciplines have four maturity levels known as the levels of *emergence* (L_1), *art* (L_2), *engineering* (L_3), and *post-engineering* (L_4).	$EMM : L_1 \subseteq L_2 \subseteq L_3 \subseteq L_4$
22	Incompressible workload (Theorem 8.5)	A given workload W_1 in software engineering can not be compressed by any kind of labor allocation, and in the best case when there is only one person involved, the minimum workload $W = W_1 = W_{min}$ may be reached.	$W \geq W_1 = W_{min}$
23	Exchangeability from labor to time (Theorem 8.8)	The *exchange rate from labor to time* $\gamma_{L\sim T}$ in a coordinative work organization is determined by the ratio between the increment of time ΔT and the increment of labor ΔL.	$\gamma_{L\sim T} = \dfrac{\Delta T}{\Delta L}$ $= \dfrac{T_1 \text{ - } T_{min}}{L_0\text{-}L_1} \; [\mathrm{M/P}]$
24	Exchangeability from time to labor (Theorem 8.9)	The *exchange rate from time to labor* $\gamma_{T\sim L}$ in a coordinative work organization is determined by the ratio between the increment of labor ΔL and the increment of time ΔT.	$\gamma_{T\sim L} = \dfrac{\Delta L}{\Delta T}$ $= \dfrac{L_0 \text{ - } L_1}{T_1 \text{ - } T_{\min}} \; [\mathrm{P/M}]$
25	Constraint on group size in coordinative work	There exists an upper limit of group size S_{max} in coordinative work organization in software engineering. Therefore, large projects must be partitioned into multiple parallel groups that each of the groups obeys the same natural	$S_{max} = \max(L_0(r)) = 20$ [P]

	(Theorem 8.10)	constraint.	
26	The risk of nonoptimal work organization (Theorem 8.11)	The risks $\mathcal{R}$ due to irrational decisions of work organization are proportional to the coordination rate *r* in a project. That is, the higher the *r*, the higher the risk under nonoptimal labor allocation.	$\mathcal{R} \propto r$
27	Cognitive Model of Memory (CMM) (Theorem 9.3)	The architecture of human memory is parallel configured by the Sensory Buffer Memory (SBM), Short-Term Memory (STM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM).	$\text{CMM} \triangleq \text{SBM}$ $\parallel \text{STM}$ $\parallel \text{LTM}$ $\parallel \text{ABM}$
28	Generic forms of learnings (Theorem 9.6)	There are sufficiently four categories of learning $\mathcal{L}$ known as those of *knowledge* ($\mathcal{L}_k$), *behaviors* ($\mathcal{L}_b$), *experience* ($\mathcal{L}_e$), and *skills* ($\mathcal{L}_s$).	$\mathcal{L} = (\mathcal{L}_k, \mathcal{L}_b, \mathcal{L}_e, \mathcal{L}_s)$
29	Representa-tion of learning results (Theorem 9.7)	The internal memory in the form of the *OAR* structure can be updated by a conjunction between the existing *OAR* and the newly created sub-*OAR*.	$OAR'\mathbf{ST} \triangleq OAR\mathbf{ST} \cup$ $sOAR\mathbf{ST}$ $= OAR\mathbf{ST} \cup (O_s, A_s, R_s)$
30	Principal intelligent advantages (Theorem 9.8)	On the basis of two principal advantages known as the *qualitative* properties (Theorem 9.1) and *quantitative* properties (Theorem 9.2), humans gain the power as the most intelligent species in the world.	
31	Cognitive mechanism of sleeping (Theorem 9.10)	Sleeping is a subconscious process for LTM establishment.	*Cognitive purpose of sleep* *= LTM establishment*
32	Mechanism of LTM establishment (Theorem 9.12)	The entire memory of information represented as an OAR model in the brain is updated by incorporating the sub-OARs formed in STM based on the following selective criteria: a) A new sub-OAR in STM was more frequently used in the	

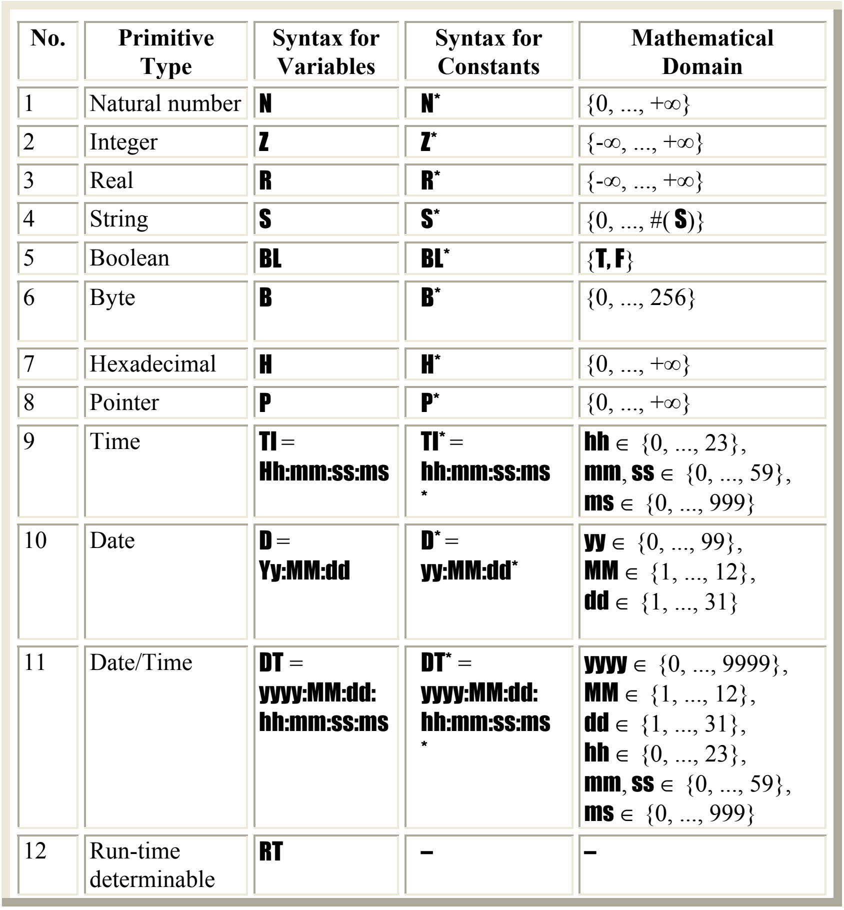
		previous 24 hours; b) A new sub-OAR in STM was related to the existing OAR in LTM at a higher level of the neural cluster hierarchy; c) A new sub-OAR in STM was given special attention so that it obtained a higher retention weight.	
33	Equivalence between open and closed systems (Theorem 10.3)	An open system S and a closed system $\widehat{S}$ in the same context is transformable when their environments Θ_S and $\Theta_{\widehat{S}}$ ($\Theta_{\widehat{S}} = C \not\subset \widehat{S}$) are taken into consideration, respectively.	$\begin{cases} \widehat{S} = \mathrm{S} \sqcup \Theta_{\mathrm{S}} \\ \mathrm{S} = \widehat{S} \sqcup \Theta_{\widehat{\mathrm{S}}} \end{cases}$
34	The bottleneck principle of systems (Theorem 10.7)	The output work of a serial system $W(S_s)$ is determined by the least powerful component of the system.	$W(S_s) = \min(\mathrm{W}(C_i) \mid C_i \in C_s \wedge 1 \leq i \leq n))$
35	The linear sum principle of systems (Theorem 10.8)	The output work of a parallel system $W(S_p)$ is a sum of the work done by all its components less the overhead of the system ϖ.	$W(S_p) = \sum_{i=1}^{\mathrm{n}} W(C_i)$ - ϖ, $C_i \in E_p,\ \varpi > 0$
36	Orientation of software engineering complexity theories (Theorem 10.13)	The complexity theories of computation and software engineering are different. The former is focused on the problems of *high throughput complexity* that are computing *time efficiency* centered; while the latter puts emphases on the problems of *functional complexity* that are human *cognition time* and *workload* oriented.	
37	Normalized software system architectures (Theorem 10.15)	Components of different subsystems should not be coupled directly, rather than be invoked through their top layer components shared in the same subsystem.	
38	Properties of games (Theorem 11.3)	A formal game G is *deterministic* and *conservative*. That is, once the game $G = (P, D, M, S)$ is set, the properties of G are determined and predictable, but not changeable by any player in the game.	

39	Conditions of win-win decisions (Theorem 11.4)	A win-win decision can be achieved when the following condition of a nonzero-sum game is satisfied, where σ is the sum of the game that is a positive nonzero constant, s_i is the expected score of player *i*, and n_s is the number of sets of matches in the game.	$\sigma \geq \frac{1}{n_s}\sum_{i=1}^{n} s_i$
40	Property of decision grids (Theorem 11.5)	The decision distance D_t in a decision grid is a constant that is determined by the number of decision trials t_i spent in the time series, where d_r and d_w represent numbers of right and wrong decisions, respectively.	$D_t = t_i = d_r + d_w$
41	Random series of unlimited trials (Theorem 11.6)	Random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under *unlimited trials*.	
42	Random series of limited trials (Theorem 11.7)	Random decisions, or equal probability right and wrong trials, will not lead to a success in any series of decisions under *limited trials*.	
43	Conditions of quality control systems (Theorem 11.8)	The *necessary conditions* for implementing a quality control system for a given product, service, or system are that all attributes of its quality can be: a) Abstractly identified b) Quantitatively defined, and c) Independently measurable.	
44	Predictability of new equilibrium (Theorem 12.2)	A *newly established equilibrium* on price P'_e is determined by the effect P' and feedback effect P'' of the driving forces deviating from the current equilibrium, and the increment of price caused by the shifting of equilibriums is as shown in the mathematical models, where ΔP may be positive or negative that represents a upward or downward shifting of the current equilibrium, respectively.	$P'_e = P'' + \frac{P'-P''}{2} = \frac{P'+P''}{2},\ P' > P'_e$ $\Delta P = P'_e - P_e = \frac{P'+P''}{2} - P_e,\ P' > P'_e$
45	Ultimate objective of software engineering	Automatic code generation is the only silver bullet to overcome the natural obstacles of the conservative software development productivity, to reduce software development costs, and to	

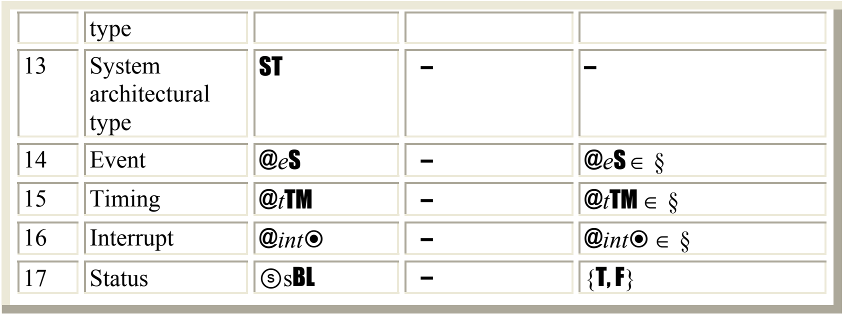
	(Theorem 12.4)	improve software quality as a result of reduced human involvement and uncertainty.	
46	Exponential Software Legacy Maintenance Costs (SLMC) (Theorem 12.5)	The ratio of maintenance cost C_m in a software development organization, r_m%, tends to exponentially increase over time t, and it is proportional to the total number of legacy systems N_L that the organization produced.	
47	Strength of motivations (Theorem 13.2)	A *motivation* M is proportional to both the strength of emotion $\|E_m\|$ and the difference between the expectancy of desire E and the current status S, of a person, and is inversely proportional to the cost to accomplish the expected motivation C, where $0 \leq \|E_m\| \leq 4$, $0 \leq (E,S) \leq 10$, and $1 \leq C \leq 10$.	$M = \frac{2.5 \bullet \mid E_m \mid \bullet (E\text{-}S)}{C}$
48	Mechanism of Software Maintenance Crisis (SMC) (Theorem 14.1)	A software development organization may face a situation known as the *software maintenance crisis*, in which the ratio of the maintenance costs r_m% is approaching 100% of the total costs that the organization spent.	
49	Rigorous levels of empirical and theoretical knowledge (Theorem 15.1)	An *empirical truth* is a truth based on or verifiable by observations, experiments, or experiences. In contrary, a *theoretical proposition* is an assertion based on formal theories or logical inferences.	
50	Necessary and sufficient conditions of IC (Theorem 15.3)	The conditions of IC, C_{IC}, are the possession of *event* B_e, *time* B_t, and *interrupt* B_{int} driven computational behaviors.	$C_{IC} = (B_e, B_t, B_{int})$
51	Necessary and sufficient conditions of AC (Theorem 15.3)	The conditions of AC, C_{AC}, are the possession of *goal* B_g and *inference* B_{inf} driven computational behaviors, in addition to the *event* B_e, *time* B_t, and *interrupt* B_i driven behaviors.	$C_{AC} = (B_g, B_{inf}, B_e, B_t, B_{int})$

Appendix G

THE TYPE SYSTEM OF SOFTWARE ENGINEERING



No.	Primitive Type	Syntax for Variables	Syntax for Constants	Mathematical Domain
1	Natural number	N	N*	{0, ..., +∞}
2	Integer	Z	Z*	{-∞, ..., +∞}
3	Real	R	R*	{-∞, ..., +∞}
4	String	S	S*	{0, ..., #(S)}
5	Boolean	BL	BL*	{T, F}
6	Byte	B	B*	{0, ..., 256}
7	Hexadecimal	H	H*	{0, ..., +∞}
8	Pointer	P	P*	{0, ..., +∞}
9	Time	TI = Hh:mm:ss:ms	TI* = hh:mm:ss:ms *	hh ∈ {0, ..., 23}, mm, ss ∈ {0, ..., 59}, ms ∈ {0, ..., 999}
10	Date	D = Yy:MM:dd	D* = yy:MM:dd*	yy ∈ {0, ..., 99}, MM ∈ {1, ..., 12}, dd ∈ {1, ..., 31}
11	Date/Time	DT = yyyy:MM:dd: hh:mm:ss:ms	DT* = yyyy:MM:dd: hh:mm:ss:ms *	yyyy ∈ {0, ..., 9999}, MM ∈ {1, ..., 12}, dd ∈ {1, ..., 31}, hh ∈ {0, ..., 23}, mm, ss ∈ {0, ..., 59}, ms ∈ {0, ..., 999}
12	Run-time determinable	RT	–	–



	type			
13	System architectural type	**ST**	–	–
14	Event	@_e_**S**	–	@_e_**S** ∈ §
15	Timing	@_t_**TM**	–	@_t_**TM** ∈ §
16	Interrupt	@_int_⦿	–	@_int_⦿ ∈ §
17	Status	ⓢs**BL**	–	{**T, F**}

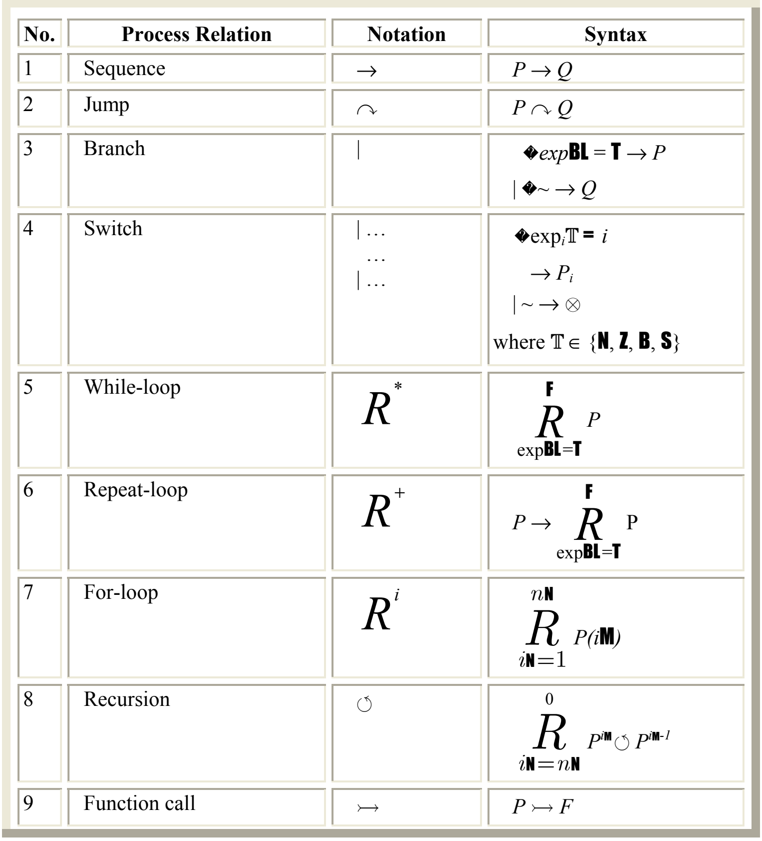
Appendix H

META PROCESSES OF SOFTWARE ENGINEERING

No.	Meta Process	Notation	Syntax
1	Assignment	$:=$	$y\mathbb{T} := x\mathbb{T},\ \mathbb{T} \in \mathcal{T}$
2	Evaluation	�	�$exp\mathbf{BL} \in \{\mathbf{T}, \mathbf{F}\}$, �$exp\mathbf{N} \in \mathbb{P}\mathbf{N}$
3	Addressing	$\Rightarrow$	$id\mathbb{T} \Rightarrow \mathrm{MEM}[ptr\mathbf{P}]\ \mathbb{T}$
4	Memory allocation	$\Leftarrow$	$id\mathbb{T} \Leftarrow \mathrm{MEM}[ptr\mathbf{P}]\ \mathbb{T}$
5	Memory release	$\nLeftarrow$	$id\mathbb{T} \nLeftarrow \mathrm{MEM}[\perp]\mathbb{T}$
6	Read	$\gtrdot$	$\mathrm{MEM}[ptr\mathbf{P}]\mathbb{T} \gtrdot x\mathbb{T}$
7	Write	$\lessdot$	$x\mathbb{T} \lessdot \mathrm{MEM}[ptr\mathbf{P}]\mathbb{T}$
8	Input	$\vert\gtrdot$	$\mathrm{PORT}[ptr\mathbf{P}]\mathbb{T} \vert\gtrdot x\mathbb{T}$
9	Output	$\vert\lessdot$	$x\mathbb{T} \vert\lessdot \mathrm{PORT}[ptr\mathbf{P}]\mathbb{T}$
10	Timing	$\underline{\underline{@}}$	$@t\mathbf{TM}\ \underline{\underline{@}}\ \S t\mathbf{TM}$ **TM = yy:MM:dd** | **hh:mm:ss:ms** | **yy:MM:dd:hh:mm:ss:ms**
11	Duration	$\triangleq$	$@t_n\mathbf{TM}\ \underline{\underline{\Delta}}\ \S t_n\mathbf{TM} + \Delta n\mathbf{TM}$
12	Increase	$\uparrow$	$\uparrow(n\mathbb{T})$
13	Decrease	$\downarrow$	$\downarrow(n\mathbb{T})$
14	Exception detection	!	$!\ (@e\mathbf{S})$
15	Skip	$\otimes$	$\otimes$
16	Stop	$\boxtimes$	$\boxtimes$
17	System	$\S$	$\S(SysID\mathbf{S})$

Appendix I

ALGEBRAIC PROCESS RELATIONS OF SOFTWARE ENGINEERING



No.	Process Relation	Notation	Syntax
1	Sequence	$\to$	$P \to Q$
2	Jump	$\curvearrowright$	$P \curvearrowright Q$
3	Branch	$\mid$	�$exp\mathbf{BL} = \mathbf{T} \to P$ $\mid$ �$\sim \to Q$
4	Switch	$\mid \ldots$ $\ldots$ $\mid \ldots$	�$\mathrm{exp}_i\mathbb{T} = i$ $\to P_i$ $\mid \sim \to \otimes$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$
5	While-loop	R^*	$\underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} P$
6	Repeat-loop	R^+	$P \to \underset{\mathrm{exp}\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} \mathrm{P}$
7	For-loop	R^i	$\underset{i\mathbf{N}=1}{\overset{n\mathbf{N}}{R}} P(i\mathbf{M})$
8	Recursion	$\circlearrowleft$	$\underset{i\mathbf{N}=n\mathbf{N}}{\overset{0}{R}} P^{i\mathbf{M}} \circlearrowleft P^{i\mathbf{M}-1}$
9	Function call	$\rightarrowtail$	$P \rightarrowtail F$

10	Parallel	$\vert\vert$	$P \vert\vert Q$
11	Concurrence	$\oiint$	P $\oiint Q$
12	Interleave	$\vert\vert\vert$	$P \vert\vert\vert Q$
13	Pipeline	»	P » Q
14	Interrupt	↯	P ↯Q
15	Time-driven dispatch	↳$_t$	@t_i**TM** ↳P_i
16	Event-driven dispatch	↳$_e$	@e_i**S** ↳P_i
17	Interrupt-driven dispatch	↳$_i$	@int_i⦿ ↳P_i

Appendix J

DEDUCTIVE SEMANTICS OF SOFTWARE ENGINEERING

Meta Processes (RTPA)		
Notation	**Syntax**	**Semantics**
Assignment :=	$y\mathbb{T} := x\mathbb{T}$, $\mathbb{T} \in \mathfrak{T}$	$\theta(\text{y}\mathbf{RT} := \text{x}\mathbf{RT}) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{y}\mathbf{RT} := \text{x}\mathbf{RT})$ $= \underset{i=0}{\overset{\#T(\text{y}\mathbf{RT} := \text{x}\mathbf{RT})}{R}} \quad \underset{j=1}{\overset{\#S(\text{y}\mathbf{RT} := \text{x}\mathbf{RT})}{R}} \quad v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}} \underset{j=1}{\overset{2}{R}} v(t_i, s_j)$ $= \left(\begin{array}{ccc} & \mathbf{xRT} & \mathbf{yRT} \\ \mathbf{t_0} & x\mathbf{RT} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT} & x\mathbf{RT} \end{array}\right)$
Evaluation �	�$exp\mathbf{BL} \in$ $\{\mathbf{T}, \mathbf{F}\}$, �$exp\mathbf{N} \in \mathbb{P}\mathbf{N}$	$\theta(\text{�}(exp\mathbf{BL}) \rightarrow \mathbf{BL})$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}(exp\mathbf{BL}) \rightarrow \mathbf{BL})$ $= \underset{i=0}{\overset{\#T(\text{�}exp\mathbf{BL} \rightarrow \mathbf{BL})}{R}} \quad \underset{j=1}{\overset{\#S(\text{�}exp\mathbf{BL} \rightarrow \mathbf{BL})}{R}} \quad v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}} \underset{j=1}{\overset{2}{R}} v(t_i, s_j)$ $= \left(\begin{array}{ccc} & exp\mathbf{B} & \text{�}(exp\mathbf{BL})\mathbf{BL} \\ (\mathbf{t_0}, \mathbf{t_1}] & \delta(exp\mathbf{BL})\mathbf{BL} & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & \mathbf{T} & \mathbf{T} \\ (\mathbf{t_1}, \mathbf{t_{2'}}] & \mathbf{F} & \mathbf{F} \end{array}\right)$

		$\theta(\text{�}exp\mathbb{T} \to \mathbb{T})$ $\triangleq \dfrac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}exp\mathbb{T} \to \mathbb{T})$ $= \mathop{R}\limits_{i=0}^{\#T(\text{�}exp\mathbb{T}\to\mathbb{T})} \mathop{R}\limits_{j=1}^{\#S(\text{�}exp\mathbb{T}\to\mathbb{T})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{2} v(t_i, s_j)$ $= \begin{pmatrix} & exp\mathbb{T} & \text{�}(exp\mathbb{T})\mathbb{T} \\ (\mathbf{t_0}, \mathbf{t_1}] & \delta(exp\mathbb{T})\mathbb{T} & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & n\mathbb{T} & n\mathbb{T} \end{pmatrix}$
Addressing ⇒	$id\mathbb{T} \Rightarrow$ $\text{MEM}[ptr\mathbf{P}]\ \mathbb{T}$	$\theta(id\mathbf{S} \Rightarrow ptr\mathbf{P})$ $\triangleq \dfrac{\partial^2}{\partial t\ \partial s} f_\theta(\text{id}\mathbf{S} \Rightarrow \text{ptr}\mathbf{P})$ $= \mathop{R}\limits_{i=0}^{\#T(\text{id}\mathbf{S}\ \Rightarrow\ \text{ptr}\mathbf{P})} \mathop{R}\limits_{j=1}^{\#S(\text{id}\mathbf{S}\ \Rightarrow\ \text{ptr}\mathbf{P})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{2} v(t_i, s_j)$ $= \begin{pmatrix} & \mathbf{idS} & \mathbf{ptrP} \\ \mathbf{t_0} & id\mathbf{S} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & id\mathbf{S} & \pi(id\mathbf{S})\mathbf{H} \end{pmatrix}$
Memory allocation ⇐	$id\mathbb{T} \Leftarrow$ $\text{MEM}[ptr\mathbf{P}]\ \mathbb{T}$	$\theta(\text{id}\mathbf{S} \Leftarrow \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $\triangleq \dfrac{\partial^2}{\partial t\ \partial s} f_\theta(\text{id}\mathbf{S} \Leftarrow \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $= \mathop{R}\limits_{i=0}^{\#T(\text{id}\mathbf{S}\Leftarrow\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})} \mathop{R}\limits_{j=1}^{\#S(\text{id}\mathbf{S}\Leftarrow\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{3} v(t_i, s_j)$ $= \begin{pmatrix} & \mathbf{idS} & \mathbf{ptrP} & \mathbf{MEMRT} \\ \mathbf{t_0} & id\mathbf{S} & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & id\mathbf{S} & \pi(id\mathbf{S})\mathbf{H} & \text{MEM}[ptr\mathbf{P}]\mathbf{RT} \end{pmatrix}$

Memory release $\nLeftarrow$	$id\mathbb{T} \nLeftarrow$ MEM[$\perp$]$\mathbb{T}$	$\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})$ $= \mathop{R}\limits_{i=0}^{\#T(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})} \mathop{R}\limits_{j=1}^{\#S(id\mathbf{S} \nLeftarrow \text{MEM}[\perp]\mathbf{RT})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{3} v(t_i, s_j)$ $= \left(\begin{array}{cccc} & \textbf{id}\mathbf{RT} & \textbf{ptr}\mathbf{P} & \textbf{MEM}\mathbf{RT} \\ \mathbf{t_0} & (id\mathbf{S}) & \pi(id\mathbf{S})\mathbf{H} & \text{MEM}(ptr\mathbf{P})\mathbf{RT} \\ (\mathbf{t_0}, \mathbf{t_1}] & \perp & \perp & \perp \end{array} \right)$
Read $\gtrdot$	MEM[$ptr\mathbf{P}$]$\mathbb{T}$ $\gtrdot x\mathbb{T}$	$\theta(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \lessdot \text{x}\mathbf{RT})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \lessdot \text{x}\mathbf{RT})$ $= \mathop{R}\limits_{i=0}^{\#T(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \lessdot \text{x}\mathbf{RT})} \mathop{R}\limits_{j=1}^{\#S(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \lessdot \text{x}\mathbf{RT})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{3} v(t_i, s_j)$ $= \left(\begin{array}{cccc} & \textbf{x}\mathbf{RT} & \textbf{ptr}\mathbf{P} & \textbf{MEM[ptr}\mathbf{P}\textbf{]}\mathbf{RT} \\ \mathbf{t_0} & x\mathbf{RT} & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT} & ptr\mathbf{P} & x\mathbf{RT} \end{array} \right)$
Write $\lessdot$	$x\mathbb{T} \lessdot$ MEM[$ptr\mathbf{P}$]$\mathbb{T}$	$\theta(\text{x}\mathbf{RT} \lessdot \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{x}\mathbf{RT} \lessdot \text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $= \mathop{R}\limits_{i=0}^{\#T(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \text{x}\mathbf{RT})} \mathop{R}\limits_{j=1}^{\#S(\text{MEM}[\text{ptr}\mathbf{P}]\mathbf{RT} \gtrdot \text{x}\mathbf{RT})} v(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{1} \mathop{R}\limits_{j=1}^{3} v(t_i, s_j)$ $= \left(\begin{array}{cccc} & \textbf{x}\mathbf{RT} & \textbf{ptr}\mathbf{P} & \textbf{MEM[ptr}\mathbf{P}\textbf{]}\mathbf{RT} \\ \mathbf{t_0} & x\mathbf{RT} & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT} & ptr\mathbf{P} & x\mathbf{RT} \end{array} \right)$

Input $\mid\gtrdot$	PORT[$ptr\mathbf{P}$]$\mathbb{T}$ $\mid\gtrdot x\mathbb{T}$	$\theta(\text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT}\mid\gtrdot \text{x}\mathbf{RT})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT}\mid\gtrdot \text{x}\mathbf{RT})$ $= \overset{\#T(\text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT}\mid\gtrdot \text{x}\mathbf{RT})}{\underset{i=0}{R}} \quad \overset{\#S(\text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT}\mid\gtrdot \text{x}\mathbf{RT})}{\underset{j=1}{R}} v(t_i,s_j)$ $= \overset{1}{\underset{i=0}{R}}\,\overset{3}{\underset{j=1}{R}} v(t_i,s_j)$ $= \left(\begin{array}{cccc} & \mathbf{ptrP} & \mathbf{PORT[ptrP]RT} & \mathbf{xRT} \\ \mathbf{t_0} & ptr\mathbf{P} & \perp & \perp \\ (\mathbf{t_0},\mathbf{t_1}] & ptr\mathbf{P} & \text{PORT}[ptr\mathbf{P}]\mathbf{RT} & \text{PORT}[ptr\mathbf{P}]\mathbf{RT} \end{array}\right)$
Output $\mid\lessdot$	$x\mathbb{T}\mid\lessdot$ PORT[$ptr\mathbf{P}$]$\mathbb{T}$	$\theta(\text{x}\mathbf{RT}\mid\lessdot \text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\text{x}\mathbf{RT}\mid\lessdot \text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT})$ $= \overset{\#T(\text{x}\mathbf{RT}\mid\lessdot \text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT})}{\underset{i=0}{R}} \quad \overset{\#S(\text{x}\mathbf{RT}\mid\lessdot \text{PORT}[\text{ptr}\mathbf{P}]\mathbf{RT})}{\underset{j=1}{R}} v(t_i,s_j)$ $= \overset{1}{\underset{i=0}{R}}\,\overset{3}{\underset{j=1}{R}} v(t_i,s_j)$ $= \left(\begin{array}{cccc} & \mathbf{xRT} & \mathbf{ptrP} & \mathbf{PORT[ptrP]RT} \\ \mathbf{t_0} & x\mathbf{RT} & \perp & \perp \\ (\mathbf{t_0},\mathbf{t_1}] & x\mathbf{RT} & ptr\mathbf{P} & x\mathbf{RT} \end{array}\right)$
Timing $\underline{\underline{@}}$	$@t\mathbf{TM}\ \underline{\underline{@}}\ \S t\mathbf{TM}$ **TM =** **yy:MM:dd** $\mid$ **hh:mm:ss:ms** $\mid$ **yy:MM:dd:hh:mm:ss:ms**	$\theta(@\,\text{t}\mathbf{TM}\underline{\underline{@}}\S\text{t}\mathbf{TM})$ $\triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(@\,\text{t}\mathbf{TM}\underline{\underline{@}}\S\text{t}\mathbf{TM})$ $= \overset{\#T(@\,\text{t}\mathbf{TM}\underline{\underline{@}}\S\text{t}\mathbf{TM})}{\underset{i=0}{R}} \quad \overset{\#S(@\,\text{t}\mathbf{TM}\underline{\underline{@}}\S\text{t}\mathbf{TM})}{\underset{j=1}{R}} v(t_i,s_j)$ $= \overset{1}{\underset{i=0}{R}}\,\overset{2}{\underset{j=1}{R}} v(t_i,s_j)$ $= \left(\begin{array}{ccc} & \mathbf{\S tTM} & \mathbf{@tTM} \\ \mathbf{t_0} & \S\text{t}\mathbf{TM} & \perp \\ (\mathbf{t_0},\mathbf{t_1}] & \S\text{t}\mathbf{TM} & \S\text{t}\mathbf{TM} \end{array}\right)$

Duration $\triangleq$	$@\mathrm{t_n}\mathbf{TM} \triangleq \S\mathrm{t_n}\mathbf{TM}$ $+ \Delta\mathrm{n}\mathbf{TM}$	$\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM}+\Delta\mathrm{d}\mathbf{Z})$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM}+\Delta\mathrm{d}\mathbf{Z})$ $= \underset{i=0}{\overset{\#T(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM})}{R}} \quad \underset{j=1}{\overset{\#S(@\,\mathrm{t}\mathbf{TM}\underline{\underline{\Delta}}\S\mathrm{t}\mathbf{TM})}{R}} \quad v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}} \underset{j=1}{\overset{3}{R}} v(t_i, s_j)$ $= \begin{pmatrix} & \S\mathrm{t}\mathbf{TM} & \Delta\mathrm{d}\mathbf{N} & @\mathrm{t}\mathbf{TM} \\ \mathbf{t_0} & \S\mathrm{t}\mathbf{TM} & \Delta\mathrm{d}\mathbf{N} & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & \S\mathrm{t}\mathbf{TM} & \Delta\mathrm{d}\mathbf{N} & \S\mathrm{t}\mathbf{TM} + \Delta d\mathbf{N} \end{pmatrix}$
Increase $\uparrow$	$\uparrow(n\mathbb{T})$	$\theta(\uparrow(\mathrm{x}\mathbf{RT}))$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\uparrow(\mathrm{x}\mathbf{RT}))$ $= \underset{i=0}{\overset{\#T(\uparrow(\mathrm{x}\mathbf{RT}))}{R}} \quad \underset{j=1}{\overset{\#S(\uparrow(\mathrm{x}\mathbf{RT}))}{R}} \quad v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}} \underset{j=1}{\overset{1}{R}} v(t_i, s_j)$ $= \begin{pmatrix} & \mathrm{x}\mathbf{RT} \\ \mathbf{t_0} & x\mathbf{RT} \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT} + 1 \end{pmatrix}$
Decrease $\downarrow$	$\downarrow(n\mathbb{T})$	$\theta(\downarrow(\mathrm{x}\mathbf{RT}))$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\downarrow(\mathrm{x}\mathbf{RT}))$ $= \underset{i=0}{\overset{\#T(\downarrow(\mathrm{x}\mathbf{RT}))}{R}} \quad \underset{j=1}{\overset{\#S(\downarrow(\mathrm{x}\mathbf{RT}))}{R}} \quad v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}} \underset{j=1}{\overset{1}{R}} v(t_i, s_j)$ $= \begin{pmatrix} & \mathrm{x}\mathbf{RT} \\ \mathbf{t_0} & x\mathbf{RT} \\ (\mathbf{t_0}, \mathbf{t_1}] & x\mathbf{RT}\ \text{-}1 \end{pmatrix}$

Exception detection !	$!\ (@e\mathbf{S})$	$\theta(!(@e\mathbf{S})$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(!(@e\mathbf{S})$ $= \underset{i=0}{\overset{\#T(!(@e\mathbf{S})}{R}}\ \underset{j=1}{\overset{\#S(!(@e\mathbf{S})}{R}} v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{3}{R}} v(t_i, s_j)$ $= \left(\begin{array}{cccc} & \mathbf{@eS} & \mathbf{ptrP} & \mathbf{PORT(ptrP)S} \\ \mathbf{t_0} & @e\mathbf{S} & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & @e\mathbf{S} & ptr\mathbf{P} & @e\mathbf{S} \end{array}\right)$
Skip $\otimes$	$\otimes$	$\theta(\otimes) \triangleq \theta(P^k \curvearrowright P^{k-1})$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P^k \curvearrowright P^{k-1})$ $= \underset{i=0}{\overset{\#T(P^k \curvearrowright P^{k-1})}{R}}\ \underset{j=1}{\overset{\#S(P^k \curvearrowright P^{k-1})}{R}} v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{2}{R}} v(t_i, s_j)$ $= \left(\begin{array}{ccc} & \mathbf{S_{P^{k-1}}} & \mathbf{S_{P^k}} \\ \mathbf{t_0} & S_{P^{k-1}} & S_{P^k} \\ (\mathbf{t_0}, \mathbf{t_1}] & S_{P^{k-1}} \setminus S_{P^k} & \perp \end{array}\right)$
Stop $\boxtimes$	$\boxtimes$	$\theta(\boxtimes) \triangleq \theta(P \curvearrowright \S)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \curvearrowright \S)$ $= \underset{i=0}{\overset{\#T(P \curvearrowright \S)}{R}}\ \underset{j=1}{\overset{\#S(P \curvearrowright \S)}{R}} v(t_i, s_j)$ $= \underset{i=0}{\overset{1}{R}}\ \underset{j=1}{\overset{2}{R}} v(t_i, s_j)$ $= \left(\begin{array}{ccc} & \mathbf{S_\S} & \mathbf{S_P} \\ \mathbf{t_0} & S_\S & S_P \\ (\mathbf{t_0}, \mathbf{t_1}] & S_\S \setminus S_P & \perp \end{array}\right)$

System §	$\S(SysID\mathbf{S})$	$\theta(\S) \triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(\S)$ $= \frac{\partial^2}{\partial t\,\partial s} f_\theta\{ \mathop{R}\limits_{i\mathbf{N}=0}^{n_e\mathbf{N}-1}(@e_i\mathbf{S} \hookrightarrow P_i)$ $\Vert \mathop{R}\limits_{j\mathbf{N}=0}^{n_t\mathbf{N}-1}(@t_j\mathbf{TM} \hookrightarrow P_j)$ $\Vert \mathop{R}\limits_{k\mathbf{N}=0}^{n_{\text{int}}\mathbf{N}-1}(@int_k\mathbf{S} \hookrightarrow P_k)$ $\}$ $= \mathop{R}\limits_{SysShuntDown\mathbf{BL}=\mathbf{F}}^{\mathbf{T}} \{ \mathop{R}\limits_{i\mathbf{N}=0}^{n_e\mathbf{N}-1}(@e_i\mathbf{S} \hookrightarrow P_i)$ $\Vert \mathop{R}\limits_{j\mathbf{N}=0}^{n_t\mathbf{N}-1}(@t_j\mathbf{TM} \hookrightarrow P_j)$ $\Vert \mathop{R}\limits_{k\mathbf{N}=0}^{n_{\text{int}}\mathbf{N}-1}(@int_k\mathbf{S} \hookrightarrow P_k)$ $\}$

Process Relations (RTPA)

Notation	Syntax	Semantics
Sequence →	$P \to Q$	$\theta(P \to Q) \triangleq \frac{\partial^2}{\partial t\,\partial s} f_\theta(P \to Q)$ $= \frac{\partial^2}{\partial t\,\partial s} f_\theta(P) \to \frac{\partial^2}{\partial t\,\partial s} f_\theta(Q)$ $= \mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j) \to$ $\mathop{R}\limits_{i=0}^{\#T(Q)} \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{\#T(\widehat{PQ}} \mathop{R}\limits_{j=1}^{\#S(P \cup Q)} v(t_i, s_j)$ $= \begin{pmatrix} & \mathbf{s_P} & \mathbf{s_Q} & \mathbf{s_{PQ}} \\ \mathbf{t_0} & \perp & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & - & V_{1PQ} \\ (\mathbf{t_1}, \mathbf{t_2}] & - & V_{2Q} & V_{2PQ} \end{pmatrix}$ $= \begin{pmatrix} \mathbf{V_P} & & \mathbf{V_{PQ}} \\ & \mathbf{V_Q} & \mathbf{V_{PQ}} \end{pmatrix}$

Jump $\curvearrowright$	$P \curvearrowright Q$	$\theta(P \curvearrowright Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \curvearrowright Q)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \ \curvearrowright\ \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \mathop{R}\limits_{i=0}^{\#T(P)} \ \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j) \ \curvearrowright\ \mathop{R}\limits_{i=0}^{\#T(Q)} \ \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j)$ $= \mathop{R}\limits_{i=0}^{\#T(\widehat{PQ})} \ \mathop{R}\limits_{j=1}^{\#S(P \cup Q)} v(t_i, s_j)$ $= \left(\begin{array}{lcccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{addr}\mathsf{\mathbf{H}} \\ \mathbf{[t_0, t_1]} & V_{1P} & \perp & V_{1PQ} & \perp \\ \mathbf{(t_1, t_2]} & - & - & - & \pi(Q\mathsf{\mathbf{S}})\mathsf{\mathbf{H}} \\ \mathbf{(t_2, t_3]} & - & V_{3Q} & V_{3PQ} & \end{array} \right)$
Branch $\mid$	$\text{�}exp\mathsf{\mathbf{BL}} = \mathsf{\mathbf{T}}$ $\to P$ $\mid \text{�}\sim \to Q$	$\theta(\text{�}exp\mathsf{\mathbf{RT}} \to P \mid \text{�}\sim \to Q)$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}exp\mathsf{\mathbf{RT}} \to P \mid \text{�}\sim \to Q)$ $= \quad \text{�}exp\mathsf{\mathbf{BL}} \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P)$ $\quad \mid \ \text{�}\sim \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \quad \text{�}exp\mathsf{\mathbf{BL}} \to \mathop{R}\limits_{i=0}^{\#T(P)} \ \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j)$ $\quad \mid \ \text{�}\sim \to \mathop{R}\limits_{i=0}^{\#T(Q)} \ \mathop{R}\limits_{j=1}^{\#S(Q)} v_Q(t_i, s_j)$ $= \left(\begin{array}{lcccc} & \mathbf{exp}\mathsf{\mathbf{BL}} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\ \mathbf{(t_0, t_1]} & \delta(exp\mathsf{\mathbf{BL}}) & \perp & \perp & \perp \\ \mathbf{(t_1, t_2]} & \mathsf{\mathbf{T}} & V_{2P} & - & V_{2PQ} \\ \mathbf{(t_1, t_{2'}]} & \mathsf{\mathbf{F}} & - & V_{3Q} & V_{3PQ} \end{array} \right)$

Switch $\mid \ldots \mid \ldots$	$\text{�}\mathrm{exp}_i\mathbb{T} = i$ $\to P_i$ $\mid \sim \to \otimes$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$	$\theta(\text{�}exp_{\mathrm{i}}\mathbf{RT} \to \mathrm{P}_{\mathrm{i}} \mid \text{�}\sim \to \otimes)$ $\triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\text{�}exp_{\mathrm{i}}\mathbf{RT} \to \mathrm{P}_{\mathrm{i}} \mid \text{�}\sim \to \otimes)$ $= \ \text{�}exp\mathbf{RT} = 0 \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_0)$ $\mid \ldots$ $\mid exp\mathbf{RT} = n-1 \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_{n-1})$ $\mid exp\mathbf{RT} = n \to \frac{\partial^2}{\partial t\ \partial s} f_\theta(\otimes)$ $= \ \text{�}exp\mathbf{RT} = 0 \to \mathop{R}\limits_{i=0}^{\#T(P_0)} \mathop{R}\limits_{j=1}^{\#S(P_0)} v_{P_0}(t_i, s_j)$ $\mid \ldots$ $\mid exp\mathbf{RT} = n-1 \to \mathop{R}\limits_{i=0}^{\#T(P_{n-1})} \mathop{R}\limits_{j=1}^{\#S(P_{n-1})} v_{P_{n-1}}(t_i, s_j)$ $\mid exp\mathbf{RT} = n \to \otimes$ $= \left(\begin{array}{cccccc} & \mathbf{exp}\mathbf{RT} & \mathbf{S}_{\mathbf{P_0}} & \cdots & \mathbf{S}_{\mathbf{P_{n\text{-}1}}} & \mathbf{S}_{\mathbf{G}} \\ [\mathbf{t_0}, \mathbf{t_1}] & \delta(\mathrm{exp}\mathbf{RT}) & \perp & \cdots & \perp & \perp \\ (\mathbf{t_1}, \mathbf{t_{2_0}}] & 0 & V_{2_0 P} & \cdots & - & V_G \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (\mathbf{t_1}, \mathbf{t_{2_{n\text{-}1}}}] & n-1 & - & \cdots & V_{2_{n-1} P} & V_G \\ (\mathbf{t_1}, \mathbf{t_{2_n}}] & n & - & \cdots & - & V_G \end{array} \right)$
While-loop R^*	$\mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}} P$	$\theta(\mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}}{}^*(P)) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}}{}^*(P))$ $= \mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}}{}^*(\frac{\partial^2}{\partial t\ \partial s} f_\theta(P))$ $= \mathop{R}\limits_{\mathrm{exp}\mathbf{BL}=\mathbf{T}}^{\mathbf{F}}{}^*(\mathop{R}\limits_{i=0}^{\#T(P)} \mathop{R}\limits_{j=1}^{\#S(P)} v_P(t_i, s_j))$ $= \left(\begin{array}{ccc} & \mathbf{exp}\mathbf{BL} & \mathbf{S_P} \\ [\mathbf{t_0}, \mathbf{t_1}] & \delta(\mathrm{exp}\mathbf{BL}) & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & \mathbf{T} & V_P \\ (\mathbf{t_1}, \mathbf{t_{2'}}] & \mathbf{F} & \otimes \\ \vdots & \vdots & \vdots \\ (\mathbf{t_3}, \mathbf{t_4}] & \delta(\mathrm{exp}\mathbf{BL}) & - \\ (\mathbf{t_4}, \mathbf{t_5}] & \mathbf{T} & V_P \\ (\mathbf{t_4}, \mathbf{t_{5'}}] & \mathbf{F} & \otimes \end{array} \right)$

Repeat-loop R^{+}	$P \rightarrow$ $\underset{\exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} \mathrm{P}$	$\theta(\underset{\exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}}{}^{+}(P)) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\underset{\exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}}{}^{+}(P))$ $= \underset{\exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}}{}^{+}(\frac{\partial^2}{\partial t\ \partial s} f_\theta(P))$ $= P \rightarrow \underset{\exp\mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}}{}^{*}(\underset{i=0}{\overset{\#T(P)}{R}} \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j))$ $= \left(\begin{array}{ccc} & \exp\mathbf{BL} & \mathbf{S_P} \\ [\mathbf{t_0}, \mathbf{t_1}] & \perp & V_P \\ (\mathbf{t_1}, \mathbf{t_2}] & \delta(\exp\mathbf{BL}) & - \\ (\mathbf{t_2}, \mathbf{t_3}] & \mathbf{T} & V_P \\ (\mathbf{t_2}, \mathbf{t_{3'}}] & \mathbf{F} & \otimes \\ \vdots & \vdots & \vdots \\ (\mathbf{t_4}, \mathbf{t_5}] & \delta(\exp\mathbf{BL}) & - \\ (\mathbf{t_5}, \mathbf{t_6}] & \mathbf{T} & V_P \\ (\mathbf{t_5}, \mathbf{t_{6'}}] & \mathbf{F} & \otimes \end{array}\right)$
For-loop R^{i}	$\underset{i\mathbf{N}=1}{\overset{n\mathbf{N}}{R}} P(i\mathbf{M})$	$\theta(\underset{\mathrm{i}\mathbf{N}=1}{\overset{\mathrm{n}}{R}}\mathrm{P(i)}) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(\underset{\mathrm{i}\mathbf{N}=1}{\overset{\mathrm{n}}{R}}\mathrm{P(i)})$ $= \underset{\mathrm{k}\mathbf{N}=1}{\overset{\mathrm{n}}{R}}(\frac{\partial^2}{\partial t\ \partial s} f_\theta(P_i))$ $= \underset{\mathrm{k}\mathbf{N}=1}{\overset{\mathrm{n}}{R}}(\underset{i=0}{\overset{\#T(P_k)}{R}} \underset{j=1}{\overset{\#S(P_k)}{R}} v_{P_k}(t_i, s_j))$ $= \left(\begin{array}{ccc} & \mathbf{k}\mathbf{N} & \mathbf{S_P} \\ [\mathbf{t_0}, \mathbf{t_1}] & 1 & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & 1 & V_P \\ \vdots & \vdots & \vdots \\ (\mathbf{t_{n-2}}, \mathbf{t_{n-1}}] & n & - \\ (\mathbf{t_{n-1}}, \mathbf{t_n}] & n & V_P \end{array}\right)$
Function call $\rightarrowtail$	$P \rightarrowtail F$	$\theta(P \rightarrowtail Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \rightarrowtail Q)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \rightarrowtail \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \underset{i=0}{\overset{\#T(P)}{R}} \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j) \rightarrowtail \underset{i=0}{\overset{\#T(Q)}{R}} \underset{j=1}{\overset{\#S(Q)}{R}} v_Q(t_i, s_j)$ $= \underset{i=0}{\overset{\#T([t_0, t_1] \frown (t_1, t_2] \frown (t_2, t_3])}{R}} \underset{j=1}{\overset{\#S(P \cup Q)}{R}} v(t_i, s_j)$ $= \left(\begin{array}{cccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\ \mathbf{t_0} & \perp & \perp & \perp \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & - & V_{1PQ} \\ (\mathbf{t_1}, \mathbf{t_2}] & - & v_{2Q} & V_{2PQ} \\ (\mathbf{t_2}, \mathbf{t_3}] & V_{3P} & - & V_{3PQ} \end{array}\right)$

Recursion $\circlearrowleft$	$\overset{0}{\underset{i\mathbf{N}=n\mathbf{N}}{R}}(P^{i\mathbf{M}}\circlearrowleft P^{i\mathbf{M}-1})$	$\theta(P \circlearrowleft P) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \circlearrowleft P)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \circlearrowleft \frac{\partial^2}{\partial t\ \partial s} f_\theta(P)$ $= \overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v(t_i, s_j) \circlearrowleft \overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v(t_i, s_j)$ $= \overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v(t_i, s_j)$ $= \begin{pmatrix} & \mathbf{S_P} \\ [\mathbf{t_0},\mathbf{t_1}] & V_{P^n} \\ (\mathbf{t_1},\mathbf{t_2}] & V_{P^{n-1}} \\ \vdots & \vdots \\ (\mathbf{t_3},\mathbf{t_4}] & V_{P^0} \\ \vdots & \vdots \\ (\mathbf{t_5},\mathbf{t_6}] & V_{P'^{n-1}} \\ (\mathbf{t_6},\mathbf{t_7}] & V_{P'^n} \end{pmatrix}$
Parallel $\parallel$	$P \parallel Q$	$\theta(P \parallel Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \parallel Q)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \parallel \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v_P(t_i, s_j) \parallel \overset{\#T(Q)}{\underset{i=0}{R}}\ \overset{\#S(Q)}{\underset{j=1}{R}} v_Q(t_i, s_j)$ $= \overset{\max(\#T(P),\#T(Q))}{\underset{i=0}{R}}\ \overset{\#S(P\cup Q)}{\underset{j=1}{R}} v(t, s)$ $= \begin{pmatrix} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\ \mathbf{t_0} & V_{0P} & V_{0Q} & V_{0PQ} \\ (\mathbf{t_0},\mathbf{t_1}] & V_{1P} & V_{1Q} & V_{1PQ} \\ (\mathbf{t_1},\mathbf{t_2}] & - & V_{2Q} & V_{2PQ} \end{pmatrix}$
Concurrence $\oiint$	P $\oiint Q$	$\theta(P \oiint Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \oiint Q)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \oiint \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \overset{\#T(P)}{\underset{i=0}{R}}\ \overset{\#S(P)}{\underset{j=1}{R}} v_P(t_i, s_j) \oiint \overset{\#T(Q)}{\underset{i=0}{R}}\ \overset{\#S(Q)}{\underset{j=1}{R}} v_Q(t_i, s_j)$ $= \overset{\max(\#T(P),\#T(Q))}{\underset{i=0}{R}}\ \overset{\#S(P\cup Q)}{\underset{j=1}{R}} v(t_i, s_j)$ $= \begin{pmatrix} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{comRT} \\ \mathbf{t_0} & V_{0P} & V_{0P} & V_{0P} & V_{0com} \\ (\mathbf{t_0},\mathbf{t_1}] & V_{1P} & - & V_{1PQ} & V_{1com} \\ (\mathbf{t_1},\mathbf{t_2}] & V_{2P} & V_{2Q} & V_{2PQ} & V_{2com} \\ (\mathbf{t_2},\mathbf{t_3}] & V_{3P} & - & V_{3PQ} & V_{3com} \end{pmatrix}$

<table>
<tr>
<td>Interleave
|||</td>
<td>$P \mathbin{|||} Q$</td>
<td>$$\theta(P \mathbin{|||} Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \mathbin{|||} Q)$$
$$= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \mathbin{|||} \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$$
$$= \underset{i=0}{\overset{\#T(P)}{R}}\ \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j) \mathbin{|||} \underset{i=0}{\overset{\#T(Q)}{R}}\ \underset{j=1}{\overset{\#S(Q)}{R}} v_Q(t_i, s_j)$$
$$= \underset{i=0}{\overset{\#T([\mathrm{t}_0,\mathrm{t}_1]\frown(\mathrm{t}_1,\mathrm{t}_2]\frown(\mathrm{t}_2,\mathrm{t}_3]\frown(\mathrm{t}_3,\mathrm{t}_4]\frown(\mathrm{t}_4,\mathrm{t}_5])}{R}}\ \underset{j=1}{\overset{\#S(P\cup Q)}{R}} v(t_i, s_j)$$
$$= \left(\begin{array}{cccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} \\ \mathbf{t_0} & V_{0P} & V_{0Q} & V_{0PQ} \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P'} & - & V_{1PQ} \\ (\mathbf{t_1}, \mathbf{t_2}] & - & V_{2Q'} & V_{2PQ} \\ (\mathbf{t_2}, \mathbf{t_3}] & V_{3P''} & - & V_{3PQ} \\ (\mathbf{t_3}, \mathbf{t_4}] & - & V_{4Q''} & V_{4PQ} \\ (\mathbf{t_4}, \mathbf{t_5}] & V_{5P'''} & - & V_{5PQ} \end{array}\right)$$</td>
</tr>
<tr>
<td>Pipeline
»</td>
<td>$P \gg Q$</td>
<td>$$\theta(P \gg Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P \gg Q)$$
$$= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) \gg \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$$
$$= \underset{i=0}{\overset{\#T(P)}{R}}\ \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j) \gg \underset{i=0}{\overset{\#T(Q)}{R}}\ \underset{j=1}{\overset{\#S(Q)}{R}} v_Q(t_i, s_j)$$
$$= \underset{i=0}{\overset{\#T(\overset{\frown}{PQ})}{R}}\ \underset{j=1}{\overset{\#S(P\cup Q)}{R}} v(t_i, s_j)$$
$$= \left(\begin{array}{cccc} & \mathbf{S_P} & \mathbf{S_{Po}} = \mathbf{S_{Qi}} & \mathbf{S_Q} \\ \mathbf{t_0} & V_{0P} & V_{0PQ} & V_{0Q} \\ (\mathbf{t_0}, \mathbf{t_1}] & V_{1P} & V_{1PQ} & - \\ (\mathbf{t_1}, \mathbf{t_2}] & - & V_{2PQ} & V_{2Q} \end{array}\right)$$</td>
</tr>
</table>

Interrupt ↯	$P ↯ Q$	$\theta(P ↯ Q) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(P ↯ Q)$ $= \frac{\partial^2}{\partial t\ \partial s} f_\theta(P) ↯ \frac{\partial^2}{\partial t\ \partial s} f_\theta(Q)$ $= \underset{i=0}{\overset{\#T(P)}{R}}\ \underset{j=1}{\overset{\#S(P)}{R}} v_P(t_i, s_j) ↯ \underset{i=0}{\overset{\#T(Q)}{R}}\ \underset{j=1}{\overset{\#S(Q)}{R}} v_Q(t_i, s_j)$ $= \underset{i=0}{\overset{\#T(P'^\frown Q^\frown P'')}{R}}\ \underset{j=1}{\overset{\#S(P\cup Q)}{R}} v(t_i, s_j)$ $= \left(\begin{array}{lcccc} & \mathbf{S_P} & \mathbf{S_Q} & \mathbf{S_{PQ}} & \mathbf{int}\odot \\ [\mathbf{t_0}, \mathbf{t_1}] & V_{1P'} & \perp & V_{1PQ} & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & - & - & V_{2PQ} & V_{2\mathbf{int}\odot} \\ (\mathbf{t_2}, \mathbf{t_3}] & - & V_{3Q} & V_{3PQ} & - \\ (\mathbf{t_3}, \mathbf{t_4}] & - & - & V_{4PQ} & V_{4\mathbf{int}'\odot} \\ (\mathbf{t_4}, \mathbf{t_5}] & V_{5P''} & - & V_{5PQ} & - \end{array}\right)$
Time-driven dispatch $↳_t$	$@t_i\mathbf{TM} ↳ P_i$	$\theta(@t_k\mathbf{TM} ↳_k \mathrm{P}_k) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@t_k\mathbf{TM} ↳_k \mathrm{P}_k)$ $== \underset{k=1}{\overset{n}{R}}(@t_k\mathbf{TM} \rightarrow \frac{\partial^2}{\partial t\ \partial s} f_\theta(P_k))$ $= \underset{k=1}{\overset{n}{R}}(@t_k\mathbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_k)}{R}}\ \underset{j=1}{\overset{\#S(P_k)}{R}} v_{P_k}(t_i, s_j))$ $= @t_1\mathbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_1)}{R}}\ \underset{j=1}{\overset{\#S(P_1)}{R}} v_{P_1}(t_i, s_j)$ $\vert \ldots$ $\vert @t_n\mathbf{TM} \rightarrow \underset{i=0}{\overset{\#T(P_n)}{R}}\ \underset{j=1}{\overset{\#S(P_n)}{R}} v_{P_n}(t_i, s_j)$ $= \left(\begin{array}{lcccc} & \boldsymbol{@t_k}\mathbf{TM} & \mathbf{S_{P_1}} & \ldots & \mathbf{S_{P_n}} \\ [\mathbf{t_0}, \mathbf{t_1}] & \delta(@t_k\mathbf{TM}) & \perp & \cdots & \perp \\ (\mathbf{t_1}, \mathbf{t_2}] & @t_1 & V_{P_1} & \cdots & - \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\mathbf{t_1}, \mathbf{t_n}] & @t_n & - & \cdots & V_{P_n} \end{array}\right)$

Event-driven dispatch $\hookrightarrow_e$	$@e_i\mathbf{S} \hookrightarrow P_i$	$\theta(@\mathrm{e}_k\mathbf{S} \hookrightarrow_e \mathrm{P}_k) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@\mathrm{e}_k\mathbf{S} \hookrightarrow_e \mathrm{P}_k)$ $= \mathop{R}\limits_{k=1}^{n}(@e_k\mathbf{S} \to \frac{\partial^2}{\partial t(P_k)\partial s(P_k)} f_\theta(P_k))$ $= \mathop{R}\limits_{k=1}^{n}(@e_k\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_k)}\ \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i,s_j))$ $= @e_1\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_1)}\ \mathop{R}\limits_{j=1}^{\#S(P_1)} v_{P_1}(t_i,s_j)$ $\mid \ldots$ $\mid @e_n\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_n)}\ \mathop{R}\limits_{j=1}^{\#S(P_n)} v_{P_n}(t_i,s_j)$ $= \left(\begin{array}{ccccc} & \boldsymbol{@e_k}\mathbf{S} & \mathbf{S_{P_1}} & \ldots & \mathbf{S_{P_n}} \\ \mathbf{[t_0,t_1]} & \delta(@e_k\mathbf{S}) & \perp & \cdots & \perp \\ \mathbf{(t_1,t_2]} & @e_1 & V_{P_1} & \cdots & - \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{(t_1,t_n]} & @e_n & - & \cdots & V_{P_n} \end{array}\right)$
Interrupt-driven dispatch $\hookrightarrow_i$	$@int_i\odot \hookrightarrow P_i$	$\theta(@int_k\mathbf{S} \hookrightarrow_{\mathrm{i}} \mathrm{P}_k) \triangleq \frac{\partial^2}{\partial t\ \partial s} f_\theta(@int_k\mathbf{S} \hookrightarrow_{\mathrm{i}} \mathrm{P}_k)$ $= \mathop{R}\limits_{k=1}^{n}(@int_k\mathbf{S} \to \frac{\partial^2}{\partial t(P_k)\partial s(P_k)} f_\theta(P_k))$ $= \mathop{R}\limits_{k=1}^{n}(@int_k\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_k)}\ \mathop{R}\limits_{j=1}^{\#S(P_k)} v_{P_k}(t_i,s_j))$ $= @int_1\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_1)}\ \mathop{R}\limits_{j=1}^{\#S(P_1)} v_{P_1}(t_i,s_j)$ $\mid \ldots$ $\mid @int_n\mathbf{S} \to \mathop{R}\limits_{i=0}^{\#T(P_n)}\ \mathop{R}\limits_{j=1}^{\#S(P_n)} v_{P_n}(t_i,s_j)$ $= \left(\begin{array}{ccccc} & \boldsymbol{@int_k}\mathbf{S} & \mathbf{S_{P_1}} & \ldots & \mathbf{S_{P_n}} \\ \mathbf{[t_0,t_1]} & \delta(@int_k\mathbf{S}) & \perp & \cdots & \perp \\ \mathbf{(t_1,t_2]} & @e_1 & V_{P_1} & \cdots & - \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{(t_1,t_n]} & @e_n & - & \cdots & V_{P_n} \end{array}\right)$

Appendix K

FORMAL MODEL OF THE ATM SYSTEM IN RTPA

K.1 DESCRIPTION OF SYSTEM ARCHITECTURE

$$
\begin{aligned}
\textbf{§(ATM)} \triangleq\ & \text{ATM.Architecture} \\
& \| \ \text{ATM.StaticBehaviors} \\
& \| \ \text{ATM.DynamicBehaviors}
\end{aligned}
$$

1.1 ATM System Architecture

$$
\begin{aligned}
\textbf{ATM.Architecture} \triangleq\ & \text{<ATMProcessor : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <SystemClock : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <CardReader : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <Keypad : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <Monitor : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <AccountDatabase : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <CashBank : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <CashDisburser : } \textbf{ST} \text{ | [1]>} \\
& \| \text{ <Events : } \textbf{ST} \text{>} \\
& \| \text{ <Status : } \textbf{ST} \text{>}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{ATM.Architecture.EventsST} \triangleq\ & \text{@SysInitial}\textbf{S} \\
& \text{| @SysClock1msInt}\textbf{S}
\end{aligned}
$$

```
| @‘PNN = 0’
| @‘PNN = 1’
| @’PNN = 2’
| @’PNN = 3’
| @’PNN = 4’
| @’PNN = 5’
| @’PNN = 6’
| @’PNN = 7’
| @’PNN = 8’
| @’ⓈCardEjectedBL := T’
| @’ⓈCardReaderFaultBL = T’
| @’ⓈCashAvailableBL = F’
| @’ⓈCashAvailableBL = T’
| @’ⓈCashBankFaultyBL = T’
| @’ⓈCashDisbursedBL = T’
| @’ⓈMonitorFaultBL = T’
| @’ⓈOperationTimeOutBL = T’
| @’ⓈServiceCompletedBL = T’
| @’ⓈServiceCancelledBL = T’
| @’ⓈSsytemFailureBL = T’
| @’ⓈSystemFailureBL := F’
| @’ⓈSysShutDownBL = T’
| @’ⓈOperationTimeOutBL = T’
| @’ⓈValidAmountBL := T’
| @’ⓈValidAmountBL = F’
| @’ⓈValidBalanceBL = F’
| @’ⓈValidBalanceBL = T’
| @’ⓈValidCardBL = F’
| @’ⓈValidCardBL = T’
| @’ⓈValidPINBL := T’
| @’ⓈValidPINBL = F’
```

ATM.Architecture.StatusST ≜ ⓈCardEjected**BL**

| ⓈCardReaderFault**BL**

| ⓈCashAvailable**BL**

| ⓈCashBankFaulty**BL**

| ⓈCashDisbursed**BL**

| ⓈMonitorFault**BL**

| ⓈOperationTimeOut**BL**

| ⓈServiceCompleted**BL**

| ⓈServiceCancelled**BL**

| ⓈSsytemFailure**BL**

| ⓈSstShutDown**BL**

| ⓈValidAmount**BL**

| ⓈValidBalance**BL**

| ⓈValidCard**BL**

| ⓈValidPIN**BL**

1.2 System CLM Schemas/Objects

CardReaderST ≜ CardReader**S** ::

(<Data : **N** | 0 ≤ Data**N** ≤ 1000000>,

<Status : **BL** | **T** = Normal ∧ **F** = Faulty>,

<CardStatus : **BL** | **T** = Inserted ∧ **F** = NoCard>,

<CardEjectDriver : **BL** | **T** = On ∧ **F** = Off >,

<Port : **B** | Port**B** = FFF1**H** >

)

KeypadST ≜ Keypad**S** ::

(<Digits : **N** | 0 ≤ Digits**N** ≤ 9>,

<EnterKey : **BL** | **T** = Pressed ∧ **F** = Unpressed>,

<CancelKey : **BL** | **T** = Pressed ∧ **F** = Unpressed>,

<Port : **B** | Port**B** = FFF2**H** >

)

```
MonitorST ≜ MonitorS ::
                    ( <Instruction : S  |  0 ≤ #(InstructionS)  ≤ 255>,
                      <Status : BL  | T = Normal ∧ F = Faulty>,
                      <Port : P  | PortP = FFF3H >
                    )

SysDatabaseST ≜ SysDatabaseS (<AccountNum : N |
                              0 ≤ AccountNumN  ≤ 1000000>)::
        ( <Status : BL  | T = Active ∧ F = Inactive>,
          <PIN : N  | 000000 ≤ PINN  ≤ 999999>,
          <Balance : N  | 0 ≤ BalanceN  ≤ 10000>,
          <MaxAllowableWithdraw : N  | MaxAllowableWithdrawN = 500>
        )

CashBankST ≜ CashBankS ::
                    ( <CashLevel : N | 0 ≤ CashLevelN  ≤ MaxLevelN>,
                      <Status : BL | T = Active ∧ F = Inactive>
                    )

CashDisburserST ≜ CashDisburserS ::
        ( <Status : BL  | T = Normal ∧ F = Faulty>,
          <CashDisburseAmount : N | 5 ≤ CashDisburseAmountN  ≤ 500>,
          <CashDisburseDriver : BL  | T = On ∧ F = Off>,
          <Port : B | PortB = FFF4H >
        )

SysClockST ≜ SysClockS ::
        ( <§t : N |  0 ≤ §tN  ≤ 1M>,
          <CurrentTime : hh:mm:ss:ms  |  00:00:00:000  ≤
                          CurrentTimehh:mm:ss:ms ≤ 23:59:59:999>,
          <Timer : ss  | 0 ≤ Timerss ≤ 3600>,
          <MainClockPort : B | MainClockPortB = FFF0H>,
          <ClockInterval : N | TimeIntervalN = 1ms>,
          <InterruptCounter : N | 0 ≤ InterruptCounterN ≤ 999>
        )
```

K.2 ATM STATIC BEHAVIORS

2.1 ATM Static Behaviors

```
ATM.StaticBehaviors ≙ SysInitial (<I:: ( )>; <O:: ( )>)
    | SysClock (<I:: ( )>; <O:: ( )>
    | Welcome (<I:: ( )>; <O:: AccountNumN, PNN, ⓢValidCardBL>)
    | CheckPIN (<I:: AccountNumN>; <O:: PNN, ⓢValidPINBL,
                 ⓢServiceCancelledBL>)
    | CheckCashAmount (<I:: ( ) >;  <O:: AmountToWithdrawN, PNN,
                        ⓢValidAmountBL, ⓢServiceCancelledBL >)
    | VerifyAccount (<I:: AccountNumN, AmountToWithdrawN>;
                     <O:: PNN, AmountToWithdrawN, ValidBalanceBL,
                     ⓢServiceCancelledBL>)
    | VerifyCashAvailability (<I:: AmountToWithdrawN>;
                              <O:: PNN, ⓢCashAvailableBL,
                              ⓢServiceCancelledBL>)
    | DisburseCash (<I:: AccountNumN, AmountToWithdrawN>;
                    <O:: PNN, ⓢCashDisbursedBL,
                    ⓢServiceCompletedBL, ⓢServiceCancelledBL>)
    | EjectCard (<I:: ( )>; <O::  PNN, ⓢCardEjectedBL>)
    | SystemFailure (<I:: ( )>; <O:: ⓢSystemFailureBL,
 ⓢSysShutDownBL>)
```

2.2 Refined ATM Static Behaviors

2.2.1 System Initialization

```
SysInitial (<I:: ( )>; <O:: ( )>) ≙
{
  Initial ATM_CLMsST
  → SysClock.§tN := 0
```

```
   → SysClock.CurrentTimehh:mm:ss:ms := CurrentTimehh:mm:ss:xx
   → SysClock.InterruptCounterN := 0
   → SysClock.TimerN := 0
   → PNN := 1
   → ⓈSystemFailureBL := F
}
```

2.2.2 System Clock

```
SysClock (<I:: ( )>; <O:: ( )>) ≙
{
   ↑(SysClock.InterruptCounterN)                    // 1ms clock interrupt
   → �SysClock.InterruptCounterN = 999              // Set to 1 second
         (   → SysClock.InterruptCounterN = 0
             → ↑(SysClock.§tN)
             → ↑(SysClock.CurrentTimehh:mm:ss)
             → ↓ (SysClock.Timerss)
             → �SysClock.CurrentTimehh:mm:ss:ms = 23:59:59:xxx
                   (  → SysClock.CurrentTimehh:mm:ss:ms := 00:00:00:xxx
                      → SysClock.§tN := 0
                   )
         )
}
```

2.2.3 Transactional States

```
// State 1
Welcome (<I:: ( )>; <O:: AccountNumN, PNN, ⓈValidCardBL>) ≙
{
      $\overset{T}{\underset{CardInsertedBL=F}{R}}$   ( PORT(MonitorST.PortP).StatusBL |> MonitorStatusBL
                       → PORT(CardReaderST.PortP).StatusBL |>
                                CardReaderStatusBL
                       → ( �MonitorStatusBL = T ∧ CardReaderStatusBL = T
                            → 'Welcome!' |<
                              PORT(MonitorST.PortP).InstructionS
                            → 'Please insert your card.' |<
                              PORT(MonitorST.PortP).InstructionS
```

```
                                  → PORT(CardReaderST.PortP).StatusBL  |>
                                     CardInsertedBL
                              | � ~
                                  → (    � MonitorStatusBL = F
                                              → ! (@‘MonitorFaultBL = T’)
                                        |  � CardReaderStatusBL = F
                                              → ! (@’CardReaderFaultBL = T’)
                                     )
                                  → PNN := 8                   // To system failure
                                  → ⊗
                              )
   → Port(CardReaderST.PortP).DataN  |>  AccountNumN
   →  (�SysDatabaseST(AccountNumN).StatusBL := T
         →ⓢValidCardBL = T
         → SysClockST.Timerss := 10                       // To wait for PIN, 10s
         → PINEnterTimesN  := 3
         → PNN := 2                                       // To check PIN again
      | � ~
         →ⓢValidCardBL = F
         → PNN := 7                                       // To eject card
      )
}


// State 2
CheckPIN(<I: AccountNumN>; <O::  PNN,
            ⓢValidPINBL,ⓢServiceCancelledBL>) ≙
{
    $\underset{\text{DataEntered}\mathbf{BL}=\mathbf{F}}{\overset{\mathbf{T}}{R}}$     ( PORT(MonitorST.PortP).StatusBL  |>  MonitorStatusBL
                              → Port (CardReaderST.PortP).StatusBL  |>
                                     CardReaderStatusBL
                              → ( �SysClockIDS.Timerss ≠ 0 ∧ MonitorStatusBL = T
                                  ∧ CardReaderStatusBL = T
                                    →   ‘Enter your PIN.’ |<
                                         PORT(MonitorST.PortP).InstructionS
```

```
                                      → PORT(KeypadST.PortP).EnterKeyBL |>
                                        DataEteredBL
                                   | � ~
                                      → (  �SysClockIDS.Timerss = 0

                                              → ⓢTimeoutBL = T

                                              → ! (‘ⓢOperationTimeOutBL = T’)
                                              → PNN := 7            // To eject card
                                         |  �MonitorStatusBL = F

                                              → ! (@ ‘MonitorFaultBL = T’)
                                              → PNN := 8            // To system failure
                                         |  �CardReaderStatusBL = F

                                              → ! (@ ’CardReaderFaultBL = T’)
                                              → PNN := 8            // To system failure
                                         )
                                      → ⊗
                                 )
                        )
   → PORT(KeypadST.PortP).DataN |>  PINN
  → (  �PINN = SysDatabaseST(AccountNumN).PINN)

        → ⓢValidPINBL := T
        →  SysClockST.Timerss := 10             // Wait for amount enter
        →  PNN := 3                             // To check cash amount
     | � ~

        → ⓢValidPINBL = F
        → ↓ (PINEnterTimesN)
        → (   �PINEnterTimesN > 0

                  →   ‘Wrong PIN. Do you want to try again?’ |<
                      PORT(MonitorST.PortP).InstructionS

                  →   $\overset{T}{\underset{TimeOutBL=F}{R}}$   ( � ΔtN := §tN + 3            // Delay 3s

                                           → TimeOutBL = T
                                        )
                  → PORT(KeypadST.PortP).EnterKeyBL |>
                             EnterKeyPressedBL
                   → PORT(KeypadST.PortP).CancelKeyBL |>
```

```
                        CancelKeyPressedBL
                 → (   �EnterKeyPressedBL = T
                         →  PNN := 2                    // To retry PIN
                    |  �CancelKeyPressedBL = T
                         → ⓈServiceCancelledBL  = T
                         →  PNN := 7                    // To eject card
                    )
            | � ~
                 → 'Invalid PIN.' |<   PORT(MonitorST.PortP).InstructionS
                 → ⓈValidPINBL  = F
                 →  PNN := 7                            // To eject card
            )
        )
}


// State 3
CheckCashAmount (<I:: ()>; <O:: AmountToWithdrawN, PNN,
                    ⓈValidAmountBL,ⓈServiceCancelledBL>) ≙
{
         $\overset{T}{\underset{DataEnteredBL=F}{R}}$   (  PORT(MonitorST.PortP).StatusBL |>  MonitorStatusBL
                  → PORT(CardReaderST.PortP).StatusBL |>
                         CardReaderStatusBL
                  → (  � SysClockIDS.Timerss ≠ 0 ∧ MonitorStatusBL = T
                         ∧ CardReaderStatusBL = T
                         →   'Enter the amount you wish to withdraw
                              ($5 … $500).' |<
                              PORT(MonitorST.PortP).InstructionS
                         →   PORT(KeypadST.PortP).EnterKeyBL |>
                              DataEteredBL
                      | � ~
                         → (  �SysClockIDS.Timerss = 0
                                → ⓈOperationTimeOutBL = T
                                → ! (@ 'OperationTimeOutBL = T')
                                → PNN := 7              // To eject card
```

```
                                          | �MonitorStatusBL = F
                                            → ! (@ ‘MonitorFaultBL = T’)
                                            → PNN := 8             // To system failure
                                          | �CardReaderStatusBL = F
                                            → ! (@ ‘CardReaderFaultBL = T’)
                                            → PNN := 8             // To system failure
                                          )
                                    → ⊗
                                )
                            )
  → PORT(KeypadST.PortP).DataN |> AmountToWithdrawN
  → ( � 5 ≤ AmountToWithdrawN ≤ MaxAllowableWithdrawN
          → ⓈValidAmountBL := T
          → PNN := 4                    // To check balance of account
    | � ~
          → ⓈValidAmountBL = F
          → ‘Amount required is out of range.’ |<
             PORT(MonitorST.PortP).InstructionS
          → $\underset{\text{ⓈTimeOutBL=F}}{\overset{\text{T}}{R}}$ ( �ΔtN := §tN + 3        // Delay 3s
                                   → TimeOutBL = T
                                 )
          → PORT(KeypadST.PortP).EnterKeyBL |> EnterKeyPressedBL
          → PORT(KeypadST.PortP).CancelKeyBL |>
                    CancelKeyPressedBL
          → ( �EnterKeyPressedBL = T
                    → SysClockST.TimerSS := 10        // Reset timer
                    → PNN := 3                        // To retry PIN
            | �CancelKeyPressedBL = T
                    → ⓈServiceCancelledBL = T
                    → PNN := 7                        // To eject card
            )
    )
}
```

```
// State 4
VerifyAccount ( <I:: AccountNumN, AmountToWithdrawN>;
                <O:: PNN, AmountToWithdrawN, ValidBalanceBL,
               ⓈServiceCancelledBL>) ≙
{
  PORT(MonitorST.PortP).StatusBL  |>  MonitorStatusBL
  → Port (CardReaderST.PortP^).StatusBL |>  CardReaderStatusBL
  → (  �MonitorStatusBL = F
          → ! (@‘MonitorFaultBL = T
          → PNN := 8                              // To system failure
          → ⊗
     | � CardReaderStatusBL = F
          → ! (@’CardReaderFaultBL = T’)
          → PNN := 8                              // To system failure
          → ⊗
    )
  → (  �AmountToWithdrawN ≤ SysDatabaseST(AccountNumN).BalanceN)
          → ⓈValidBalanceBL := T
          →  PNN := 5                         // To check cash availability
     | � ~
          → ⓈValidBalanceBL = F
          →  ‘Account balance is insufficient to withdraw. Retry a new
              amount?’ |<  PORT(MonitorST.PortP).InstructionS
          →  $\mathop{R}\limits_{\text{ⓈTimeOutBL=F}}^{\text{T}}$  ( � ΔtN := §tN + 3           // Delay 3s
                                  → TimeOutBL = T
                               )
          → PORT(KeypadST.PortP).EnterKeyBL |>   EnterKeyPressedBL
          → PORT(KeypadST.PortP).CancelKeyBL |>
                  CancelKeyPressedBL
          →  (  �EnterKeyPressedBL = T
                 →  SysClockST.TimerSS := 10        // Set timer
                 →  PNN := 3                        // To reenter amount
             | �  CancelKeyPressedBL = T
                 → ⓈServiceCancelledBL  = T
```

```
                    → PNN := 7                              // To eject card
                )
        )
}


// State 5
VerifyCashAvailability ( <I:: AmountToWithdrawN>; <O:: PNN,
                         ⓢCashAvailableBL, ⓢServiceCancelledBL>) ≙
{
  PORT(MonitorST.PortP).StatusBL |> MonitorStatusBL
  → PORT(CardReaderST.PortP).StatusBL |> CardReaderStatusBL
  → ( �MonitorStatusBL = F
          → ! (@‘MonitorFaultBL = T
          → PNN := 8                                   // To system failure
          → ⊗
      | �CardReaderStatusBL = F
          → ! (@’CardReaderFaultBL = T’)
          → PNN := 8                                   // To system failure
          → ⊗
      )
  → (�CashBankST.StatusBL) = T
          → ( �CashBankST.CashLevelN ≥ AmountToWithdrawN
                  → ⓢCashAvailableBL = T
                  → PNN := 6                           // To disburse cash
              | � ~
                  → ⓢCashAvailableBL = F
                  → ‘No sufficient cash available in this machine. Retry a
                     new amount?’ |<
                     PORT(MonitorST.PortP).InstructionS
                  → $\overset{\mathbf{T}}{\underset{ⓢ\text{TimeOut}\mathbf{BL}=\mathbf{F}}{R}}$ ( �ΔtN := §tN + 3           // Delay 3s
                                  → TimeOutBL = T
                               )
                  → PORT(KeypadST.PortP).EnterKeyBL |>
                         EnterKeyPressedBL
                  → PORT(KeypadST.PortP).CancelKeyBL |>
                       CancelKeyPressedBL
                  → ( �EnterKeyPressedBL = T
                            → SysClockST.Timerss := 10      // Set timer
                            → PNN := 3                  // To reenter amount
                     | � CancelKeyPressedBL = T
                            → ⓈServiceCancelledBL  = T
                            → PNN := 7                      // To eject card
                     )
          )
      | � ~
           → ( !( @‘CashBankFaultyBL = T’)
           → PNN := 8                               // To system failure
           → ⊗
  )
}


// State 6
DisburseCash ( <I:: AccountNumN, AmountToWithdrawN>;
               <O:: ⓈCashDisburseBL, PNN, ⓈServiceCompletedBL,
               ⓈServiceCancelledBL> ) ≙
{
  PORT(MonitorST.PortP).StatusBL  |>  MonitorStatusBL
  → PORT(CardReaderST.PortP).StatusBL  |>  CardReaderStatusBL
  → (  �MonitorStatusBL = F
           → ! (@‘MonitorFaultBL = T’)
           → PNN := 8                               // To system failure
           → ⊗
      | � CardReaderStatusBL = F
           → ! (@‘CardReaderFaultBL = T’)
           → PNN := 8                               // To system failure
           → ⊗
     )
  → PORT(CashDisburserST.PortP).StatusBL |>   CashDisburserStatusBL
  → ( �CashDisburserStatusBL = T
           → AmountToWithdrawN  |<
```

```
                PORT(CashDisburserST.PortP).CashDisburseAmountN
            → ⓢCashDisburseBL |<
                PORT(CashDisburserST.PortP^).CashDisburseDriverBL
            → ⓢCashDisburseBL = T
            → CashBankST.CashLevelN - AmountToWithdrawN
            → SysDatabaseST(AccountNumN).BalanceN -
                                    AmountToWithdrawN
            → ⓢServiceCancelledBL  = T
            → PNN := 7                                  // To eject card
        | � ~
            → ⓢCashDisburseBL = F
            → ( !( @‘CashDisburserFaultyBL = T’)
            → ‘System failure. Please use another machine.’ |<
               Port (MonitorST.PortP).InstructionS
            →  PNN := 8                                 // To eject card
            → ⊗
        )
}

// State 7
EjectCard  (<I::( )>; <O::  PNN, ⓢCardEjectedBL>) ≙
{
  (� ⓢServiceCompletedBL = T
        → ‘Please collect your card.’ |<  Port (MonitorST.PortP).InstructionS
        → EjectCardBL := T
        → EjectCardBL |>  PORT(CardReaderST.PortP).CardEjectDriverBL
        → ⓢCardEjectedBL := T
        →  PNN := 1
   | � ⓢServiceCancelledBL = T
        → ‘Please collect your card.’ |<  Port (MonitorST.PortP).InstructionS
        → EjectCardBL := T
        → EjectCardBL |>  PORT(CardReaderST.PortP).CardEjectDriverBL
        → ⓢCardEjectedBL := T
        → PNN := 1
   | � ⓢOperationTimeOutBL = T
```

```
        → ‘Service time out. Please collect your card.’ |<
           PORT(MonitorST.PortP).InstructionS
        → EjectCardBL := T
        → EjectCardBL |>  PORT(CardReaderST.PortP).CardEjectDriverBL
        → ⓈCardEjectedBL := T
        → PNN := 1
   | ? ⓈValidCardBL = F
        → ‘Invalid Card.’ |<  PORT(MonitorST.PortP).InstructionS
        → EjectCardBL := T
        → EjectCardBL |>  PORT(CardReaderST.PortP).CardEjectDriverBL
        → ⓈCardEjectedBL := T
        → PNN := 1
   | � ⓈValidPINBL = F
        → ‘Invalid PIN.’ |<  PORT(MonitorST.PortP).InstructionS
        → EjectCardBL := T
        → EjectCardBL |> PORT(CardReaderST.PortP).CardEjectDriverBL
        → ⓈCardEjectedBL := T
        → PNN := 1
}


// State 8
SystemFailure (<I::( )>; <O:: ⓈSystemFailureBL, ⓈSysShutDownBL>) ≙
{
   ⓈSystemFailureBL = T
   → ( !( @‘ⓈSystemFailureBL = T’)
   → ‘System failure. Please use another machine.’ |<
      PORT(MonitorST.PortP).InstructionS
   → EjectCardBL := T
   → EjectCardBL |> PORT(CardReaderST.PortP).CardEjectDriverBL
   → ⓈCardEjectedBL := T
   → ⓈSysShutDownBL := T
   → ⊗
}
```

K.3 ATM DYNAMIC BEHAVIORS

3.1 ATM Dynamic Behaviors

ATM.DynamicBehaviors ≙

```
{
    §
  || // Base level
     (   SysInitial
        | Welcome
        | CheckPIN
        | CheckAmount
        | VerifyAccountBalance
        | VerifyCashAvailability
        | DisburseCash
        | EjectCard
        | SystemFailure
     )
  ||  // Interrupt level
    (   SysClock
      || SysDiagnosis
    )
}
```

3.2 ATM Process Deployment

ATM.ProcessDeployment ≙

```
{ // Basic level processes
  @SysInitial$\textbf{S}$
     ↳ ( SysInitial
          ↳  $\mathop{R}\limits_{ⓈSysShutDown\textbf{BL}=\textbf{F}}^{\textbf{T}}$  ATMProcessDispatching
          → ⊠
        )
  || // Interrupt level processes
  ⊙ @SysClock1msIntS
     ↗ ( SysClock
           ↳ SysDiagnosis
         )
     ↘ ⊙
}
```

3.3 ATM Process Dispatch

ATMProcessDispatch ≜

```
{
   @‘PNN = 0’ → ∅
 | @‘PNN = 1’ ↳ Welcome  (<I:: ( )>; <O:: AccountNumN, PNN,
                               ⓈValidCardBL>)
 | @’PNN = 2’ ↳ CheckPIN (<I:: AccountNumN>; <O::  PNN,
                               ⓈValidPINBL, ⓈServiceCamcelledBL>)
 | @’PNN = 3’ ↳ CheckCashAmount (<I:: ( )>; <O:: AmountToWithdrawN,
                 PNN, ⓈValidAmountBL, ⓈServiceCancelledBL>)
 | @’PNN = 4’ ↳ VerifyAccount  (<I:: AccountNumN,
                 AmountToWithdrawN>; <O:: PNN,
                 AmountToWithdrawN, ⓈValidBalanceBL,
                 ⓈServiceCancelledBL>)
 | @’PNN = 5’ ↳ VerifyCashAvailability  (<I:: AmountToWithdrawN>;
                 <O:: PNN, ⓈCashAvailableBL, ⓈServiceCancelledBL>)
 | @’PNN = 6’ ↳ DisburseCash  (<I:: AccountNumN,
                 AmountToWithdrawN>;  <O:: ⓈCashDisbursedBL, PNN,
                 ⓈServiceCompletedBL, ⓈServiceCancelledBL>)
 | @’PNN = 7’ ↳ EjectCard  (<I:: ( )>; <O::  PNN, ⓈCardEjectedBL>)
 | @’PNN = 8’ ↳ SsytemFailure (<I:: ( )>; <O:: ⓈSystemFailureBL,
                                   ⓈSysShutDownBL>)
}
```

Appendix L

LIST OF FIGURES

Chapter 3 **Philosophical Foundations of SE**

Chapter 4 **Mathematical Foundations of SE**

Chapter 5 **Computing Foundations of SE**

Chapter 6 **Linguistic Foundations of SE**

Chapter 7 **Information Science Foundations of SE**

Chapter 8 **Engineering Foundations of SE**

Chapter 9 **Cognitive Informatics Foundations of SE**

Chapter 13 **Sociology Foundations of SE**

Chapter 14 **Retrospect on Software Engineering**

Chapter 15 **Prospect on Software Science**

Appendix M

LIST OF TABLES

Chapter 7 **Information Science Foundations of SE**

Chapter 8 **Engineering Foundations of SE**

Chapter 9 **Cognitive Informatics Foundations of SE**

Chapter 10 **System Science Foundations of SE**

Chapter 11 **Management Science Foundations of SE**