Andrea Janes
Giancarlo Succi

Lean Software Development in Action

Springer

Lean Software Development in Action

Andrea Janes • Giancarlo Succi

Lean Software Development in Action

Springer

Andrea Janes
Giancarlo Succi
Libera Università di Bolzano
Bolzano
Italy

ISBN 978-3-662-44178-7 ISBN 978-3-642-00503-9 (eBook)
DOI 10.1007/978-3-642-00503-9
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014953961

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

We wrote this book with the idea to show a practical implementation of Lean software development, gluing together well-proven tools to provide a way to develop Lean. The message this book wants to convey is the utilization of goal-oriented, automated measurement for the creation of a Lean organization and the facilitation of Lean software development.

Since its conception in the mid-1950s, Lean thinking has been very successful in manufacturing: it has helped organizations to focus on value-providing activities, to identify unnecessary ones, and therefore to increase the efficiency and effectiveness of the overall process. There are several proposals on how to translate Lean principles into software engineering practices, starting from the Agile Manifesto and the pioneering work of Kent Beck (see Chap. 4), of Mary and Tom Poppendieck (see Chap. 6), and of many others.

It is a fact that to be successful, a Lean orientation of software development has to go hand in hand with the business strategy of the company as a whole. To achieve such a goal, there are two interrelated aspects that require special attention: measurement and experience management.

Measurement means to describe real-world processes according to clearly defined rules to understand, control, and improve (see Chap. 9). Now, "Lean software company thinking" requires a comprehensive measurement program: measurement is a cornerstone of Lean thinking, as well evidenced by the founders of the Lean approach, such as Taiichi Ōno, James P. Womack, and Daniel T. Jones (see Chap. 1).

Managers of software companies have often perceived measurement as a waste of time, mostly because the measurement process required a lot of effort from developers and managers, and often the results of a measurement program could not be used to steer effectively the direction of the business. In the last decade, a new approach to software measurement has emerged, where, on one side, it has become evident that a successful software measurement program ought to be introduced in a company-wide measurement approach to promote process improvement, as discussed by the CMMI (see Chap. 3) and several other recent software process

improvement initiatives, and, on the other side, the measurement process has become much less invasive and therefore effort consuming (see Chap. 9).

Measurement is a necessary step in systematic quality approaches like Six Sigma or in performance management instruments like the Balanced Scorecard (see Chap. 3). A methodical approach to measurement as proposed in this book will allow to introduce quality and performance management instruments that rely on it.

The second aspect requiring special attention is experience, i.e., valuable, stored, specific knowledge that was acquired in a previous problem-solving situation (see Chap. 8).

Software development organizations have always been concerned with collecting and institutionalizing the experience of its developers and engineers, so that it would not be too dependent on its key people and the turnover would not affect it too much. The experience of these years have shown that Lean software organizations are even more dependent on the skills of individual people.

It appears therefore critical to promote the institutionalization of experience across the entire company. The studies that the authors have done on the field have evidenced that the collected software measures can be extremely instrumental to achieve such a goal.

Altogether, in this book the authors provide the necessary knowledge to establish "Lean software company thinking" taking advantage of the most recent approaches to software measurement.

The main target group of the book consists of people working in the field of software engineering that want to understand how to obtain an efficient and effective software development process. This group includes developers, managers, and students following an M.Sc. curriculum in software engineering.

This book particularly applies to small and medium enterprises—the dominating organizational form in the European software sector—that do not have the resources to put in place the mentioned processes of collecting and institutionalizing the experience of its developers and engineers. This creates a high dependency on its key people, and turnover can severely challenge the ability to continue to provide a given product or service to the market. Additionally, not institutionalizing the own experiences means wasting an important opportunity for small and medium enterprises to maintain or improve their position in the market.

A comprehensive, company-wide measurement approach is exactly what (even very small) companies need to align the performed activities to the needs of the stakeholders, to the business strategy, etc. With the automatic, non-invasive measurement proposed in this book—this is what we mean by "Lean Software Development in action"—more companies will be able to optimize their software development process towards Lean.

The discussion on how to transfer Lean concepts into software development is still ongoing. To support this process, we aim to inspire others to follow our line of research to apply measurement to understand the effects of introduced changes in the software development process even at an early stage. Measurement helps

to understand and to reason about the obtained desired and undesired changes. Moreover, it points out opportunities to improve.

We present our approach to Lean Software Development in three parts:

1. The first part illustrates what the term "Lean Production" means, why we think it is useful to transfer Lean concepts into software engineering, and which approaches exist to transfer the Lean concept into software engineering.
2. The second part illustrates the tools we use to achieve Lean Software Development: Non-invasive Measurement, the Goal Question Metric approach, and the Experience Factory.
3. The third part illustrates how we combined the different tools to enable Lean Thinking in software development.

Background Knowledge

Whenever we thought that some background knowledge is interesting or useful in a particular part of the text, we added it in a box like this.

A last but important note: throughout this book, wherever the masculine form is used, it applies to the feminine form as well.

Andrea Janes
Giancarlo Succi

Bozen, Italy
May 2014

Acronyms

ABC	Activity Based Costing
AJAX	Asynchronous JavaScript and XML
ANSI	American National Standards Institute
API	Application Programming Interface
PMI	Project Management Institute
AT&T	American Telephone and Telegraph Company
BC	Before Christ
CC	Cyclomatic Complexity
CD	Compact Disk
CIO	Chief Information Officer
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
DAD	Disciplined Agile Delivery
DMAIC	Define, Measure, Analyze, Improve, Control
DSDM	Dynamic Systems Development Method
ETL	Extract Transform Load
FSS	Flagship Software Services, a company in our fictional story
GQM	Goal Question Metrics
GUI	Graphical User Interface
HTML	HyperText Markup Language
IBM	International Business Machines
ICSE	International Conference on Software Engineering
ISO	International Organization for Standardization
IT	Information Technology
IQ	Intelligence Quotient
JSP	Java Server Pages
LOC	Lines Of Code
NASA	National Aeronautics and Space Administration
NPV	Net Present Value
PC	Process Costing
PDCA	Plan Do Check Act

PDSA	Plan Do Study Act
PSP	Personal Software Process
QIP	Quality in Practice
RCA	Resource Consumption Accounting
REST	REpresentational State Transfer
RFID	Radio-frequency identification
RAD	Rapid Application Development
ROI	Return on Investment
SOA	Service Oriented Architecture
SWOT	Strengths, Weaknesses, Opportunities, and Threats
UML	Unified Modeling Language
VP	Vice President
WMC	Weighted Methods per Class
WWI	World War I
WWII	World War II
XP	Extreme Programming
XXX	The boss in our fictional story

Contents

Part I
Motivation for Lean Software Development

The first part illustrates what the term “Lean Production” means in manufacturing, why we think its useful to transfer Lean concepts into software engineering, and which existing approaches exist to transfer the Lean concept into software engineering.

Chapter 1
Introduction

God grant me the serenity to accept the things I cannot change;
The courage to change the things I can;
And the wisdom to know the difference.

Francis of Assisi

It was a rainy say. Uli was walking on the street and meditating.

He just had a morning session and a lunch with his senior software architects. They worked intensively to revise the architecture of the system according to the new specs; they restructured the old model into a new one, which appeared solid enough; lastly they put together a reasonable Gantt to meet the incoming deadline—they still had eight weeks but they did not want to end up as the last 4 times working 24/7 and still having delays and missing functionalities. Neither they wanted the pain to work under such stress, nor they wanted that their company, Flagship Software Services (FSS), got a bad reputation.

So far the project had been quite hard. ADE Inc., the customer, had changed its mind a few times; nonetheless, it was pressing to see soon tangible results. Moreover, the boss of ADE, XXX, expressed several times his technical opinion on the project suggesting (ehm, imposing) his technical view he claimed he was a software engineer and he knew the subject. Indeed he was a software engineer, but the last time he had a technical task was in the late eighties and he had no knowledge whatsoever of what UML, SOA, Web-Services, REST interfaces, etc. are. So Uli had a hard time tutoring XXX in these new technologies.

Also Uli had to explain in details to XXX why the results were late and the importance to spend enough time at the beginning of the project on defining the requirements in a sound and complete way, designing a robust architecture, prototyping the critical part of the system. XXX wanted the results now but Uli said he needed to be patient and to grant them enough time for the work to be done. A bit more work now would have resulted in a far better system later. Each time the discussion reached this point, XXX started shouting that he was not paying for

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_1

requirement documents, architectures, prototypes and all these "useless stuff" (well, XXX used a more colored expression), but for only one thing, a "running system." And each time XXX said such things, Uli had a spike in his stomach, started repeating his mantra (better not to detail it here), smiled, and with his best effort to be calm, claimed that he was a professional not a wizard, he could not produce buggy or inconsistent solutions, and that at the end of the project and for the 20 years to come XXX would have thanked him for the robust, adaptable, simple, efficient, and effective system he produced.

Uli suffered for some turnover in his team: the people were stressed and unmotivated, so someone left. Since Uli clearly divided the competencies in the team to have what he believed was a more effective organization, sometimes turnover was hard, especially when the only owner of a technology left. However, he managed to keep the core of the team in place. He claimed several times that the senior management was supporting their effort, that the project was of strategic importance for the company, and that at the end of the project FSS would have granted everyone an extra bonus.

The phone rang. Athi was calling; she was the boss of Uli, the VP Engineering of FSS.

"Uli, I have a good news and a bad news." "Athi, the good news first, please." "You do not loose your job." Uli imagined the bad news... "The bad news is that J decided to cut the project and to restructure your team." "You must be joking." but he knew she was not it was just to express his anger... "I am not, sorry. I am very sorry, indeed. But there was nothing to do. XXX called J and said that they did not want to go ahead this way any more. They could consider going on with the project, but only if you were replaced. Since your team is loyal to you, J thought it is better to assign the project to another team ... Uli ... Uli ... Are you there?"

1.1 Introduction

Software development has always been problematic, and yet it is.

From the first days of computers, it was acknowledged that telling a computer what to do was not easy. One important step was the abstraction of the complexity of the hardware through programming languages, which helped to improve quality and productivity [3, 10].

Consequently, in the late 1950s and in the 1960s, the research on programming languages flourished, producing several languages that we still use, either as is or in new forms. Among these languages there were BASIC, Fortran, LISP, C, ALGOL, and so on [10].

Indeed, moving to more sophisticated languages improved the situation, but unfortunately, the ultimate, prettiest, and coolest programming languages could not simplify significantly how software was produced [3].

According to the popular wisdom, when people face problems they cannot solve, they establish committees who, in turn, create definitions, and the books say that in 1968, a group of scientists met in Garmisch (Germany), in the mountains near Munich, and coined a few terms, among which are "software crisis," "software gap," and "software engineering" [9].

The doctors sat in front of the sick and analyzed the symptoms: the software gap, the gap that occurred between ambitions and achievements. Based on the symptoms, they diagnosed the software crisis, that is, the inability to produce software as one would have liked. And luckily, they found the therapy: software engineering, the application of sound engineering principles to the production of software.

Still, 40+ years later we are here discussing the software gap, the software crisis, and the role of software engineering. Here below there are some facts.

In a survey of the Cutter Consortium of 2008, about 12 % of the analyzed software projects were canceled before they delivered anything; of the remaining, more than 20 % were found to deliver only poorly or fairly in at least one of the success criteria: user satisfaction, ability to meet budget targets, ability to meet schedule targets, product quality, and staff productivity [6].

A study evaluating 412 projects in the United Kingdom in 2007 found that 9 % of the projects were abandoned, 5 % were over-budget, and 18 % were over time [14].

The Standish Group regularly studies projects in large, medium, and small cross-industry US companies [8]. Their results indicate that in 2012, about 18 % of the projects are canceled before completion or never implemented, and about 43 % of the projects were challenged: completed and operational, but over-budget or over the time estimate, or with fewer features and functions than initially specified [15].

We show some studies in Fig. 1.1.[1] We see that the reported values for canceled and challenged projects are higher in Standish Group reports than in others. Some researchers (e.g., [7,14]) claim that this is because the definitions, e.g., for "success" or "cost overrun," differ between the Standish Group reports and other reports, and other researchers claim that the numbers are wrong [1,8]. Nevertheless, the Standish Group reports are highly cited in the research community, which shows that many researchers think they are relevant.

According to a study performed by Emam and Koru, some reasons why software development projects fail are [6]:

[1]The here reported list of studies is not exhaustive. We want to illustrate that the problem of canceled, over-budget, and overtime projects existed and still exists today.

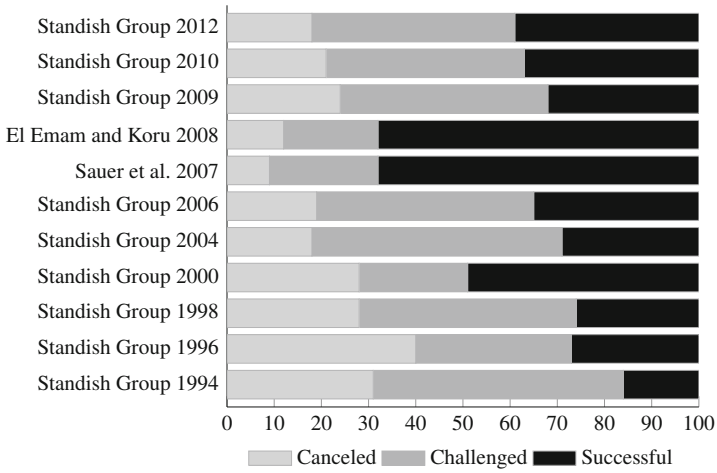


Fig. 1.1 Comparison of the results of studies about the success rate in software development projects [8, 15]

1. the senior management not sufficiently involved;
2. there are too many requirements and scope changes;
3. there is a lack of management skills;
4. the development costs are higher than planned;
5. there is a lack of technical skills;
6. the system is not anymore needed;
7. the development time is longer than planned;
8. there is a lack of experience with a technology;
9. not enough staff;
10. critical quality problems with software; or
11. the end users were not sufficiently involved.

So, what went wrong? The term "engineering" has a lot to say. When we say: "we engineer a device," we often mean that:

1. we understand the (physical) laws underneath the device and its operational environment,
2. we use mathematics to model the (physical) laws and to optimize the behavior of such device in its environment, and
3. we produce a perfected device.

All such activities go in the direction of understanding better what the device is (1), making a plan how to obtain our goals using the device (2), and optimizing its production process based on the previous two steps (3).

There is nothing wrong in the process of engineering a device; on the contrary, it works marvelously well, provided we really follow the three steps above, especially the first, understanding the (physical) laws underneath.

Unfortunately, software engineering has often been done without understanding point 1, that is, without understanding the complexity of the process of creating ideas, that is, programs. Too often people thought that producing ideas was like growing tomatoes. Now, growing tomatoes is really great, but a tomato is not a computer program: growing tomatoes is quite a repeatable process that is based on a sequence of well-known activities that are clearly observable and, for the most part, controllable.

The challenges of building software stem from our intrinsic lack of understanding it, and the essential difficulties of building software can be divided into the following four categories [3]:

1. **Complexity**: software systems consist of a multiplicity of unlike parts that can assume a large number of different states. This makes it difficult to conceive, describe, and test software;
2. **Conformity**: software has to be integrated with previously developed software systems and laws and conform to the way people are used to work. These dependencies increase the likelihood that the software has to be changed in the future;
3. **Changeability**: all successful software gets changed—as new uses of a software product are found, stakeholders push for modifications. Moreover, successful software survives the life cycle of the environment (e.g., operating system, hardware, etc.) it was written for. This means software has to be updated to the new needs or environments it has to run in;
4. **Invisibility**: the invisibility of software and the difficulties in visualizing it make it difficult to reason and to communicate about it.

In his groundbreaking book, Brooks concludes that "building software will always be hard. There is inherently no silver bullet" [3].

We wrote this book because we think that we can do something about it.

As we will see, we will address all four aspects in this book: we will see how Lean software development tackles complexity through an iterative, "customer-value-focused mindset/philosophy [1]." Lean software development is based on an Agile, flexible development approach (see Chap. 4) and therefore alleviates the conformity and changeability problem. We will also see how Lean software development continuously aims to improve, becoming more and more efficient. Indeed, Lean wants to create a learning organization which knows how to measure its own performance and to decide what to do next. In this journey, also the invisibility of software has to be addressed through measurement. We will look at this aspect in Chap. 9.

In respect to the mentioned four problems, this book presents and combines past achievements and—as said in the preface—wants to show a practical implementation of Lean software development, gluing together well-proven tools to provide a way to develop Lean. The message this book wants to convey is the utilization of goal-oriented, automated measurement for creation of a Lean organization and the facilitation of Lean software development.

The first, most significant step that we have to take is to understand the specificity of software development and to acknowledge that traditional engineering approaches fall short of providing suitable solutions to it. The second step is to define a "software-specific" approach to "engineer" the development of software systems.

The next sections of this chapter deals with the first question, and the remaining of the book with the second.

1.2 Tame and Wicked Problems

Planning theory distinguishes tame and wicked problems. In this section we present the decalogue that Rittel and Webber have proposed to classify a problem either as tame or wicked.

Tame problems are those problems that can be easily engineered; they can be formulated exhaustively and stated containing all the information needed for understanding and solving the problem.

Not all problems are tame. There are also wicked problems. The information needed to describe the wicked problem depends on the idea to solve it.

An example for a wicked problem is: "how can we reduce the crime in the streets?" The causes of this problem are not completely clear. For example, a high unemployment rate could be the reason for a high crime rate, as it could be a weak educational system. It is difficult to develop a strategy to solve the problem if the factors influencing that problem are not clear. So in this case, the information needed to implement a given strategy depend on the factors considered influencing the final result.

Thus, the formulation of a wicked problem is the problem itself. Once we know what we need to do to achieve our goal, the solution itself is clear. If we could identify low unemployment as the only reason for crime in the streets, trivially, we will try to rise employment. As a consequence, **wicked problems have no stopping rule**: since we do not know if and how our actions contribute to the goal, it is difficult to understand when to stop with one action and to proceed with another action.

As a result, there is no unambiguous criteria to decide if a wicked problem is solved or not. In fact, for wicked problems, it is not even possible to tell how good the adopted solution will be, since every solution creates a set of consequences that cannot be fully evaluated in advance. Let us consider the case that we propose to lower the unemployment rate to reduce the crime in the streets. We could create new working places by lowering the minimum wage, so it would be cheaper for companies to hire people and unemployment would be lower. But this could lead to a situation where more and more workers are hired with the new minimum wage and the situation could get even worse than at the beginning, as now even working people would not have enough to eat and would turn into gamblers, burglars, thieves, etc. We could end up with even more crime in the streets.

The unforeseeable (and not revertible) consequences occurring after an attempt to solve a wicked problem make it even more difficult to devise a solution.

To simplify the identification of wicked problems, Rittel and Webber have built a decalogue [12], which Mary and Tom Poppendieck also use as a starting point when they discuss Lean software development [11]:

1. Wicked projects cannot provide a definitive, analytical formulation of the problem they target. Formulating the project and the solution is essentially the same task. Each time you attempt to create a solution, you get a new, hopefully better, understanding of the project.
2. Wicked projects have no a stopping rule telling when the problem they target has been solved. Since you cannot define the problem, it is almost impossible to tell when it has been resolved. The problem-solving process proceeds iteratively and ends when resources are depleted and/or stakeholders lose interest in a further refinement of the currently proposed solution.
3. Solutions to problems in wicked projects are not true or false, but good or bad. Since there are no unambiguous criteria for deciding if the project is resolved, getting all stakeholders to agree that a resolution is "good enough" can be a challenge.
4. There is no immediate or ultimate test of a solution to the targeted problem in a wicked project. Solutions to such projects generate waves of consequences, and it is impossible to know how these waves will eventually play out.
5. Each solution to the problem targeted by a wicked project has irreversible consequences. Care must be placed in managing assumed solutions. Once the website is published or the new customer service package goes live, you cannot take back what was online or revert to the former customer database.
6. Wicked projects do not have a well-described, widely accepted set of potential solutions. The various stakeholders may have differing views of what are acceptable solutions. It is a matter of judgment as to when enough potential solutions have emerged and which should be pursued.
7. Each wicked project is essentially unique. There are no well-defined "classes" of solutions that can be applied to a specific case. It is not easy to find analogous projects, previously solved and well documented, so that their solution could be duplicated.
8. The problem targeted by a wicked project can be considered a symptom of another problem. A wicked project deals with a set of interlocking issues and constraints that change over time, embedded in a dynamic and evolving context.
9. The causes of a problem targeted by a wicked project can be explained in several ways. There are several stakeholders who have various and changing ideas about what is the project, its nature, its causes, and the associated solution.
10. The project must not go wrong. Mistake is not an option here. Despite the inability to express the project solution analytically, it is not allowed to fail the project.

Now, the first step to solve a wicked problem is to identify its wickedness and not to try to solve it as if it were tame!

1.3 Software Development Is a Wicked Problem

Software engineers have to understand the problem, design a solution that solves the problem, and implement the solution.

However, in doing so, they are confronted by exactly those aspects that haunt them if they approach wicked problems. It is very hard to understand the problem completely upfront (i.e., collect all requirements) and therefore also hard to devise a complete solution when starting with the implementation. Moreover, it is unknown how users will accept the new solution and which further requirements will be needed later. During the software development, it is difficult to assess how good the implemented solution will be and how close we are to the solution. Also, the languages, the operating systems, the APIs, and the databases on which we plan to develop the software are likely to evolve in an unforeseeable way.

Here are some examples (compare with the indicators of wicked projects above):

1. It is very hard to plan a software development project upfront considering all eventualities.
2. A software product is hardly perfect or finished; as soon as users use it, new requirements will arise.
3. There does not exist the one single solution to a software engineering problem.
4. We cannot determine how well an implementation solves the requirements until we implement it.
5. Choices are sometimes very costly to reverse. The last option is to throw away the existing product and start from scratch.
6. There are infinite ways to solve a software development problem.
7. Every software development project is essentially unique.
8. Software engineering is a wicked problem on multiple levels: choosing the "best" database system, user interface, operating system, hardware, etc. is also a wicked problem.
9. The problem a software system aims to solve is seen differently by the stakeholders of the system: users, developers, maintainers, database operators, etc.
10. Software development (i.e., to try to solve the problem) means to spend resources. To be wrong, i.e., to deliver a solution that the users retain not useful, is not considered an option.

The invisibility of software mentioned above amplifies the difficulties with wicked problems: it makes it difficult to understand if our actions help to progress in the desired direction, as well as it makes it difficult to assess the actual situation. These difficulties arise from the fact that the primary output of a software development process is invisible [13]. An invisible output is hard to describe and also the constraints affecting it. The invisibility of the output and of the constraints that influence the output lead to a number of consequences.

The lack of physical constraints can lead to a perception that everything is possible with IT, although software development is governed by real constraint.

In fact, the nature of constraints in software engineering tends to be abstract and multidimensional, therefore difficult to understand and communicate.

In comparison, the physical constraints and the costs faced when requesting modifications to a building can be communicated and understood by all stakeholders. The difficulty of communication between stakeholders can result in not clearly understanding the limitations of IT and as a consequence in unrealistic expectations and overambitious projects [3].

Moreover, since the output of the process is not tangible, it is very easy for the project to continue for a considerable time before problems become apparent and without the possibility to verify that the project is progressing in the desired direction [13].

The lack of physical constraints does not limit the way software is produced and used as much as in other engineering fields. As a result, software is used to approach problems never solved before. Moreover, software is used to solve old problems in new, more efficient ways, which leads to development embarking into research (as was discovered in 1968 [9]), trying to cope with complexities never approached before. Not knowing which functionalities are needed makes it difficult to meet budget and schedule targets.

In summary, we claim that software development is a wicked problem (see also [4, 11]). When Brooks says that "there is inherently no silver bullet," he just reconfirms the wickedness of software. In this book we want to show how a Lean approach can address the wickedness of software:

1. The iterative, customer-oriented approach reduces the need of detailed planning, which reduces complexity. Moreover, it is also harder to fail the project since the client, the one that defines what "failure" means, keeps the control over the output of the project during the development.
2. An approach based on measurement and collaboration helps to understand if the team is going towards the right direction and to balance the needs of different stakeholders (see Chap. 9).
3. A flexible, Agile approach helps to keep the costs of change low (see Chap. 4) which lowers the need to find the "best" solution immediately and allows to find a solution that fits the specific context.
4. Just-in-time production (see Chap. 2) maximizes the chances to know how well an implementation solves the requirements.
5. An approach that systematically collects experience and reuses it in future projects helps to collect "best practices" for a specific context (see Chap. 8).

1.4 Taylorism and Software Development

The first approaches to software engineering have been mostly guided by the ideas of Scientific Management by Frederick W. Taylor (see Fig. 1.2): the construction of the final outcome based on a thorough plan, the division of a project in different

Fig. 1.2 Frederick Winslow Taylor (image courtesy of Wikipedia [17])

steps, etc. When we talk about "engineering" a solution, we do not do much other than applying Taylorism—Taylor was indeed an engineer.

Frederick W. Taylor did not trust that people can work efficiently on their own. He wrote: "When the same workman returns to work on the following day, instead of using every effort to turn out the largest possible amount of work, in a majority of the cases this man deliberately plans to do as little as he safely can to turn out far less work than he is well able to do in many instances to do not more than one-third to one-half of a proper day's work [16]."

Consequently, he claimed that for any company, the best is when each person produces every day the highest possible efficiency: "no one can be found who will deny that in the case of any single individual, the greatest prosperity can exist only when that individual has reached his highest state of efficiency; that is, when he is turning out his largest daily output [16]."

To achieve such efficiency, he proposes four key activities for managers:

1. They develop a science for each element of a man's work, which replaces the old rule-of-thumb method.
2. They scientifically select and then train, teach, and develop the workman, whereas in the past, he chose his own work and trained himself as best as he could.
3. They heartily cooperate with the men so as to insure all of the work is being done in accordance with the principles of the science which has been developed.
4. There is an almost equal division of the work and the responsibility between the management and the workmen. The management takes over all work for which they are better fitted than the workmen, while in the past almost all of the work and the greater part of the responsibility were thrown upon the men.

Notice that since Taylor does not trust much the individual workers, he thinks that the know-how of each worker needs to be transferred to the managers. The

know-how should be translated into "procedural knowledge" on how to do the work, automated to the highest possible extent so that workers can be easily substituted.

Taylorism has deeply influenced what we consider "good management practices." The following terms are often considered the cornerstones of any sound organization:

- the rigid and detailed upfront planning division of labor and specialization of the workforce and
- the clear formalization of a problem and division of a large problem in small subproblems.

Who objects to divide the work and to specialize the skills of the workforce accordingly? It would go against the common sense. Indeed it is important to have adequate and specialized skills and that people need to master technology. Technical excellence is indeed the prerequisite of an effective software organization. But the growth in technology should not result in the creation of mutually exclusive compartments of knowledge within an organization. Rather, it should cross-fertilize the entire organization, where the individual skills of the individual developers should synergistically build the success of the team.

Who dares to say that defining rules in details to manage the production may disturb and even disrupt and kill an organization? We know that it is good to define rules—the rules save us from chaos; we learned it in kindergarten. However, only rules that naturally emerge from the production process help in organizing a production process. Rules that are imposed externally to make the process "aesthetically clean" and that are verified by an external inspector typically prevent the process to flow naturally towards the end product.

Who dares to say that the formalization of a problem has to follow its understanding and that the understanding sometimes (well, often and nearly always) follows the first attempt to solve the problem? We were taught that we should first analyze and then solve. Well, we have seen that wicked problems exist, so sometimes the understanding of a problem comes with attempts to solve it.

Luckily, we have often broken the rules and we solved problems, real problems. Software engineers are like Sherlock Holmes, the famous detective. Sherlock Holmes faces problems that go beyond the usual complexity, where the typical "divide and conquer" does not work. The solutions to a problem are uncertain, and the causes can be multiple.

And any step he takes is irreversible—there is no way to go back 1 day or even 1 h if the delinquent discovered that Sherlock Holmes is after him.

Likewise, in most situations, software engineers have to be broad in their knowledge, as they face new issues every day. They have to make several attempts before finding the right solution. And if they fail, well, they have no way back. Trying to help them and to make software development more effective by adding rules and superimposing layers of organization is useless or worse; it may be extremely detrimental.

Yet, they are not wizards. Sherlock Holmes is not a wizard; he is a detective. Likewise, software engineers are not hackers; they are serious engineers.

We understand now why the attempt of solving the software crisis by a widespread adoption of tools like formal methods have not led anywhere. The problem is not the inability of software engineers to capture requirements, design, etc. in a formal way. Actually, software engineers use a very elaborate formal language, the programming language. The problem is that in most of the cases, requirements are fully captured only once the software has been shipped—or even later; the design is clear only at the end of the project.

Life would be much easier if a very careful upfront analysis of the need of the user, the widespread adoption of a complex-but-yet-very-sound formal language, and a very detailed planning could solve the software crisis. People have often thought that this was the case, and they have spent enormous effort in finding better ways to collect requirements that would never change, to define a stellar formal language, and to devise the ultimate planning technique and the associated tool. Now, it is definitely good to have a better way to collect requirements, a stellar formal language, and an ultimate planning tool. But it is not the answer to the problem of the software crisis.

Lack of understanding is not caused by the superficiality of analysts, which is addressable by the adoption of the coolest formal language. Unexpected changes are not accidents that occur but are avoidable by good planning. Wrong or misleading decisions are not the result of poor management. Lack of understanding, unexpected changes, and misleading decisions are intrinsic to software development.

And lastly, it is not possible to "engineer" the production so that there would be no need of the "software hero." Good companies are made of heroes! Heroes are not hackers—they are well-trained individuals that know how to play the game in a team and are capable of devising creative and unique solutions that the "regular" folks would not elaborate.

So, are software engineers so strange that the development of software is so different than the production of any other good? Our answer is *NO*.

There are many disciplines that exhibit the same features as software development. We have already mentioned criminology, but there is also medicine. For example, when someone has a serious disease, if he can, he does not go just to the general practitioner or to the specialist suggested by the general practitioner. Everyone wants to go to the **good** doctor, the one that gives the best therapy and solves even the most intricate cases! The process of devising new drugs is also similar to software development, only a few attempts succeed and sometimes the results of an experimentation are totally unexpected, and drugs intended to cure a disease may result to cure totally different diseases.

Batie [2] mentions many other fields in which wicked problem issue areas exist: terrorism, global climate change, nuclear energy, poverty, crime, ecological health, pandemics, genetically modified food, water resource management, trade liberalization, the use of stem cells, biofuel production, nanotechnology, gun control, air quality, sustainable development, biodiversity, environmental restoration, forest fire management, and animal welfare.

Also art restoration seems also to be a wicked problem. When the restoration of the Sistine Chapel (from November 1984 to April 1994) had to be prepared, the restorers knew that they could not plan every eventuality ahead (indicator 1). They had to arbitrarily decide when the restoration is to be considered finished since there is no objective criteria for this (indicators 2 and 4). There were infinite methods, tools, and approaches to be chosen from (indicators 3, 6, and 8). In case of a mistake leading to the dissolution of the initial frescoes by Michelangelo, it would have been impossible (or extremely hard) to restore the initial state (indicator 5) and they would have been blamed for it (indicator 10). Their existing experience was definitely helpful; still every restoration is essentially unique (indicator 7). Finally, the result of the restoration in 1994 was highly controversial; the result was considered as terrific by some, as too aggressive, and, hence, as a destruction of the original artwork by others (indicator 9).

1.5 Summary

The lack of physical constraints does not limit the way software is produced or used as much as in other engineering fields. As a result, software is used to approach problems never solved before. Software is used to solve old problems in new, more efficient ways, which leads to development embarking into research (as was discovered in 1968), trying to cope with (wicked) complexities never approached before. Not clearly knowing which functionalities are needed makes it difficult to meet budget and schedule targets.

"Engineering" as seen by Frederick W. Taylor seems not to really work for software engineering. Brooks [3] found out why: due to the complexity, the need of conformity, the changeability, and the invisibility of software systems it is very hard to understand, model, and track the progress of software development [13]:

- It is hard to understand the current status: an invisible output is hard to describe and also the constraints affecting it.
- It is hard to understand the progress: it is difficult to understand if our actions contribute to the progress in the desired direction. Since the output of the process is not tangible, it is very easy for the project to continue for a considerable time before problems become apparent and without the possibility to verify that the project is progressing in the desired direction.
- Everything seems possible: the lack of physical constraints can lead to a perception that everything is possible with IT, although software development is governed by real constraints.

Software engineering shows properties typically found in wicked problems—and these properties have to be addressed.

Problems

1.1. There is an ongoing discussion whether software development is an art or a craft. Those that see it more as an art think that there are no rules that can help you to develop good software: it is like for every other artist, either you have the ability to paint, dance, or program or you do not. Since there are no rules, software is not something that can be engineered because there are factors that contribute to the success of software development, that depend on creativity, which we (still) cannot engineer.

On the other hand, some see it more as a craft, which also has a creative component, but there **are** rules one can follow to build good software and that one does not need to be born with the ability to program software, but that one can learn it.

What aspects of making software feel like an art, which like a craft to you? Which position to do defend?

1.2. Gerald M. Weinberg once said: "If houses were built the way software is built, the first woodpecker would bring down civilization." Such anecdotes, sayings, and jokes about software are forwarded every day from e-mailbox to e-mailbox. Probably because they bluntly express what many users think about software: "the average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed [5]."

Why, do you think, we could get along with this in the past, and still today we can? A company producing cars with such a low quality would be already bankrupt. Why does this not happen with software? Why do customers still buy software?

References

1. Ambler, S.W.: The non-existent software crisis: Debunking the chaos report. Online: http://www.drdobbs.com/architecture-and-design/the-non-existent-software-crisis-debunki/240165910 (2014). Accessed 30 April 2014
2. Batie, S.S.: Wicked problems and applied economics. Am. J. Agric. Econ. **90**(5), 1176–1191 (2008)
3. Brooks, F.P. Jr.: No silver bullet: Essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
4. DeGrace, P., Stahl, L.H.: Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms. Yourdon Press Computing Series. Yourdon Press, Englewood Cliffs (1990)
5. Dijkstra, E.W.: The end of computing science? Commun. ACM **44**(3), 92 (2001)
6. Emam, K.E., Koru, A.G.: A replicated survey of IT software project failures. IEEE Softw. **25**(5), 84–90 (2008)
7. Jørgensen, M., Moløkken-Østvold, K.J.: How large are software cost overruns? Critical comments on the standish group's chaos reports. Inf. Softw. Technol. **48**(4), 297–301 (2006)
8. Laurenz, E.J., Verhoef, C.: The rise and fall of the chaos report figures. IEEE Softw. **27**(1), 30–36 (2010)

9. Naur, P., Randell, B. (eds.): Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO (1969)
10. O'Regan, G.: A Brief History of Computing, 2nd edn. Springer, London (2012)
11. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley Professional, Boston (2003)
12. Rittel, H.W.J., Webber, M.M.: Dilemmas in a general theory of planning. Policy Sci. **4**(2), 155–169 (1973)
13. Royal Academy of Engineering and British Computer Society: The Challenges of Complex IT Projects: The Report of a Working Group from the Royal Academy of Engineering and the British Computer Society. The Royal Academy of Engineering. Online: http://www.bcs.org/upload/pdf/complexity.pdf (2004). Accessed 4 Dec 2013
14. Sauer, C., Gemino, A., Reich, B.H.: The impact of size and volatility on IT project performance. Commun. ACM **50**(11), 79–84 (2007)
15. Standish Group International: Chaos Manifesto 2013: Think Big, Act Small (2013)
16. Taylor, F.W.: The Principles of Scientific Management. Harper & Brothers, New York. Online: http://www.gutenberg.org/ebooks/6435 (1911). Accessed 4 Dec 2013
17. Wikipedia contributors: Frederick winslow Taylor. Online: http://en.wikipedia.org/wiki/Frederick_Winslow_Taylor (2013). Accessed 4 Dec 2013

Chapter 2
The Lean Revolution

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupery

Less than a second passed when Uli called Athi back. "Fire me, not them. I am responsible for the situation, I take all the blame for it." "Sorry, it would not work. We know you. You are smart, you know how to

Uli ran back to his office and called again a meeting of his four senior architects. Euril, his second, and Perim arrived immediately: they perceived that some important was going on. The other two, Sinon and Elp arrived later, but still within the usual 10'.

At the beginning of the project Uli started the tradition of preparing a coffee to all his senior architects. He did not want to change the ceremony on such day. There was no reason whatsoever to show lack of respects for such group of dedicated individuals. So, he started asking what they wanted: 2 cappuccinos, one Americano, one decaf (for him), and one espresso. Uli went to the coffee machine and started the process. They knew it would have taken a while to complete it, so they sat down and started chatting. After about 10 minutes Uli reappeared and was quite distressed—"Who wants the decaf cappuccino?", he asked. No way, he messed up the orders! He had to restart.

To make the process easier he decided to write down what everyone wanted. Only four choices, but he was nervous (who would not be) and he knew from programming that under stress the memory is not so effective. Once he completed to prepare the beverages, he started to add the sugar. That was easy! That was a peculiar group—all the five of them wanted the same amount and kind of sugar: half spoon of brown sugar! He put all the cups in a raw and started to add it. The telephone rang—an ad! He explained calmly (so to say) to the caller that he have never accepted to have his name inserted in any list for advertisement and that they were violating his privacy! But after the short call he forgot where he had added the sugar and where

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_2

he did not! Since they were large beverages and the amount of sugar was small, he could not possible to recover where he was.

And back to the room of the team, more exhausted, explaining the situation. Euril suggested him to prepare one coffee at a time, asking each of them what she or he wanted. Well, Uli thought it was a waste of time: he knew how to manage parallel threads for preparation of coffees, but at this point he did not want to reply. And so he started back, one at a time from the selection of the powder down to the addition of the sugar and... everything went fine.

Uli started thinking, while drinking his decaf—he looked absent but after a few moment a smile appeared in his face, and he shouted "Eureka!" and run out of the office.

Everyone was shocked.

2.1 Introduction

In this book we focus on management, with specific reference to the production of software, which poses the challenges we discussed in Chap. 1.

Management becomes challenging when the number of people in the team grows. From a historical perspective, some of the first circumstances where this has happened was during hunting and in war. The problem of management also emerged with the first major movements of goods and was regulated by the presence of a marketplace. Lastly, with the industrial revolution it emerged in the firm, where hundreds and then thousands of people worked together to produce goods.

We now notice, especially in the area of mobile phone applications, that concepts of the traditional market place get reintroduced in new ways. Vendors of cellular phones like Apple and Google organize virtual marketplaces where a wealthy trade of applications occurs. Typically, these applications are small and developed by individual developers and small teams [5, 6].

During these first moments in history in which management was born, different management styles emerged and have proven to be successful in different circumstances. It is important to have them in mind, as they reappeared later on also in the software industry, often claiming that they were groundbreaking ideas.

In the first wars, soldiers from opposite armies used to run against each other and to fight. Later, they realized that giving some structuring to the fight improved the results. The Greeks had the idea of forcing the infantry to move neck-to-neck in a large, rectangular shape, to act as a hammer against the enemies. They called such structuring a phalanx (Fig. 2.1).

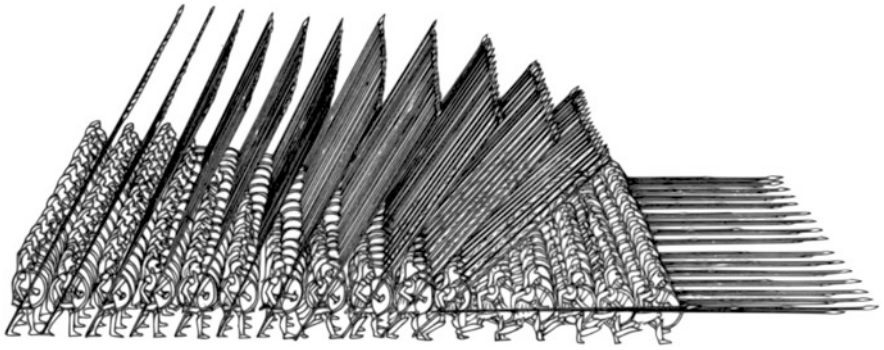


Fig. 2.1 The Macedonian phalanx (image courtesy of Wikipedia [26])

In the fourth century BC, Philip the Second of Macedonia and his son Alexander the Great perfected the structure of the Phalanx [10]. The Macedonian soldiers were armed with a long lance, the sarissa of more than 5 m, and a shield. While approaching the enemies, the first line had the sarissas down and the subsequent lines kept it up. Once they were in touch with the opposing army, also the other lines put the sarissas down, so that the enemies had to face a very dense cluster or dangerous daggers and could not do nothing but to step back, well, in principle.

The phalanx is the quintessence of structuring for the army of the time. It required discipline and a sense of duty. After all, the soldiers in the first line had to stay firm in the lines even if the enemy was overwhelming and they knew that they were going to be dead in a few moments. A soldier in the phalanx had nothing to do but to obey and sacrifice himself.

A totally different approach was taken by the Roman Quintus Fabius Maximus [16]. In the second century BC, the troupes of Carthage under the lead of Hannibal attacked Rome. They came down from the Alps with elephants and repeatedly defeated the Romans in open battles. Rome had a rule that in case of emergency, most of the constitutional rights were suspended and almost all the power went to an appointed dictators for 6 months. In this case, the Romans selected as dictator an experienced man, Quintus Fabius Maximus. Fabius avoided confronting Hannibal in open fields; rather, he tried the best intelligence to predict the moves of Hannibal in Italy and entertained with him short fights when he felt Hannibal was weaker. The long-term goal of Fabius was to put the army of Hannibal under a constant stress so that he would have to leave Italy. Fabius was very successful and Hannibal lost men and motivation.

Despite the winning approach, the Romans started criticizing Fabius. They claimed he was a coward. Someone even suggested that he was a betrayer. Fabius claimed that his goal was not to display courage but to save Rome. Still, his citizens were not happy. At the end, going against all the typical rule of a dictatorship, they

appointed another general, Minucius, to share the command in chief of the Roman army. Minucius and Fabius decided to split the army in two. Minucius did not wait much and at the first occasion attacked Hannibal, and if it were not for the wise and intelligence rescue of Fabius, he would have faced an enormous defeat. Then the Romans accepted the approach of Fabius and continued the war against Hannibal with his wise strategy.

After a while, however, another man, Varro, build his political fortune by criticizing the approach of Fabius, again claiming that it was not honorable, driven by lack of courage, and so on. Varro wanted a face-to-face confrontation with Hannibal. Again, the people liked this bold approach and appointed him as consul. In short, Varro was badly defeated by Hannibal; the Roman reappointed Fabius and eventually won.

Altogether, this short review evidences that even in the army, where obedience and order are considered to be of paramount importance, there are situations where it is better to have a very structured approach but also other situations when it is better to act flexibly. Noticeably, there are people who claim that a flexible approach is an evidence of lack of (management, technical, personal) skills, courage, resources, vision, etc. The history of Fabius is a paradigmatic example. And these people claim this despite all evidences: it looks like that they to apply some sort of aesthetic model to management, where everything has to be ordered, clean, disciplined, symmetric, and bold.

A similar discussion can be made for software development: the initial software development models, such as the waterfall model, originated from the idea to minimize the risk to do something wrong. One does everything step-by-step: plan, think, implement, think again, test, think, and deploy. It is like planning to climb Mount Everest: You do not want to end up somewhere unprepared and die. You plan thoroughly and then carry out the plan. Waterfall was conceived as a risk-reducing technique.

With time, practitioners realized that the waterfall model makes two assumptions that frequently do not hold (e.g., [2, 3]):

- we have all the necessary information to make a plan and
- we face rising and changing costs, i.e., fixing something later in the process costs more than fixing it earlier.

As we will see later, the Agile and Lean communities claim that both assumptions do not always hold: sometimes it is not possible to plan everything. Better than to speculate in the planning phase, it can be advantageous to develop a prototype and then plan again once we find out. Moreover, it is not true that mistakes that are corrected later in the process have extraordinary costs. Technologies like updates over the Internet and programming techniques that keep the adaptability of the code high and its complexity low, e.g., component-based architectures, refactoring, etc., help to keep the costs of change under control.

If we want to compare this approach with the climb to Mount Everest we could say that Agile and Lean are like exploring an uncharted forest: You cannot plan. You carefully walk and chart it, and if you find out you went wrong, you go back a bit and try again.

2.2 Henry Ford

As we mentioned in Chap. 1, Taylorism was the first prominent attempt to supply a structured approach to managing a civil organization. We have already discussed in Chap. 1 his ideas of:

- rigid and detailed upfront planning;
- division of labor and specialization of the workforce; and
- the clear formalization of a problem and division of a large problem in small subproblems

The ideas of scientific management have deeply influenced software engineering, software engineering intended in a broad sense, i.e., the "systematic design, development, and development of software products and the management of the software process [12])."

Most of the ideas of Frederick W. Taylor were put in practice by Henry Ford. Ford was almost a contemporary of Taylor, but it is unclear whether he was inspired by the ideas of Taylor or he started his approach anew.

Ford was different from Taylor in his attention to the welfare of workers. He introduced several improvements on the life of workers: for example, he set the minimum wage to $5 per hour and improved the workplace safety to avoid serious accidents [9]. During World War I and the crisis of the late 1920s also for his factories the situation degraded, but still he did care for those working for him and did not treat them as brainless machines.

In addition to the attention to the life of workers, Ford introduced two key ideas to the production of cars (and, indeed, of any good): economies of scale and just-in-time delivery.

Ford understands that the production of a car goes through several steps aimed at producing the various parts of it. The production of each part requires in turn the preparation of the instrumentations and of the tools, the acquisition of the raw material, the actual work to produce the car, and then the disposal of the tools and the dismissal of the waste. The acquisition of the raw material requires also interactions with suppliers, typically external to the factory.

His idea is that to optimize the production of a car, each step should be optimized, and in addition:

- more than one item at a time should be produced together in batches, so the preparatory work could be factored out of several items and
- each item should be produced by workers specialized in such production.

The idea is neat. Let us call p the time to prepare the instrumentations and the tools, a the time for the acquisition of the raw material, w the time to do the actual work to produce the car, and d the time for the disposal of the tools and the dismissal of the waste. The improvements of the technology and of the process reduce the value of w and of the overall production time.

Moreover, if we produce one item at a time, the total time to produce an item (alone), t_a is

$$t_a = p + a + w + d$$

And the time to produce n cars T_a is

$$T_a = n \times (p + a + w + d)$$

If we produce n cars in one batch, then, roughly speaking, the time to produce the batch T_b is

$$T_b = p + a + n \times w + d = (p + a + d) + n \times w$$

Now, $T_a > T_b$, since for T_b I do $p + a + d$ only once. The larger the number of cars n, the lower the production costs for one car. Figure 2.2 shows the average time to produce one item for two possible values for w, w_a, and w_b, where $w_a > w_b$.

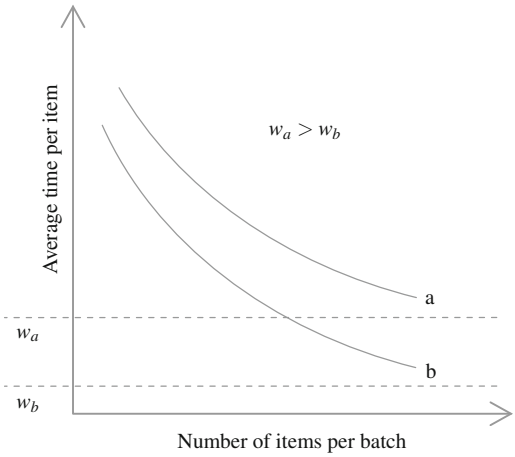


Fig. 2.2 Average time per item when the batch size is increased

Analogous equations can be drawn for the production costs. The phenomenon of the reduction of average cost per item when the number of items increases is called "economies of scale," because scaling up the production results in better economies.

Note that there are limits to the scale-up; these limits are imposed by the physical space available, by the dimensions of the machinery in place, by the possible

deterioration of the raw material, and so on. Beyond such limits, an increase in the production does not result in a reduction of the average costs; in such case there are "diseconomies of scale."

As we mentioned, often the production of goods requires the production of components of such goods or their acquisition from suppliers. A key question is how many components ought to be supplied to produce a good. The obvious answer is exactly the amount needed. However, what if a component is damaged or wrong, or if the production process proceeds faster than needed? Well, the idea is to pile up a few more components "just in case" there is a need for them.

It is like when we prepare a crostata pie. The crostata is a simple, butter-based pie with at the top the jam of choice. So, to bake a crostata we need flower, butter, eggs, water, and, indeed, the jam of choice. When we prepare the crostata, though, we may think that an egg can get broken, the butter may get old, or, worse, that at the end we may decide to change the jam. So we get a few extra eggs, a constant supply of fresh butter, and a variety of jams. This redundancy ensures that we do not get stopped in the process of preparing a crostata. However, the redundancy has an adverse effect: we need to handle an inventory of food for the crostata, which means costs of refrigerations and costs for food that might be never used. Altogether, we have extra costs that are due to our approach of "just in case."

Applying the idea of "just in case" to the production of cars has even higher needs of components, because, as mentioned, we might want to take full advantages of our infrastructure and we might want to have an excess of components so that machines would never stop. This results in even higher costs.

The idea of Ford is to keep the inventory to the minimum, getting all what is needed "just-in-time" for its use and in full shape for it, so that there is no need for the expensive inventory. So he put a lot of effort to coordinate suppliers inside the company and suppliers outside the company.

"Just-in-time production" means that the parts needed during the production process are made available at the right place, at the right moment, and in the right quantity.

Altogether, Ford achieved a remarkable success with his production model. Figure 2.3 evidences such success showing the variation in items produced and in cost of the famous T model from 1908 to 1916.

The Ford model resulted in mass production of goods. When Ford started selling his cars, only a miniscule fraction of the families had a car. His idea was that if he managed to produce a solid and cheap car, he would have sold it—and this really happened, at least initially. He once said: "Any customer can have a car painted any colour that he wants so long as it is black" [9].

However, after World War II, two concurrent phenomena occurred [14]:

- a small but yet significant portion of the market was already saturated and
- the credo "if you make it, you can sell it" had no validity anymore. Economists describe this event as a shift from a seller's market to a buyer's market [4, 19].

In such years people had less cash to spend and started to focus on the value. The Ford model started to decline.

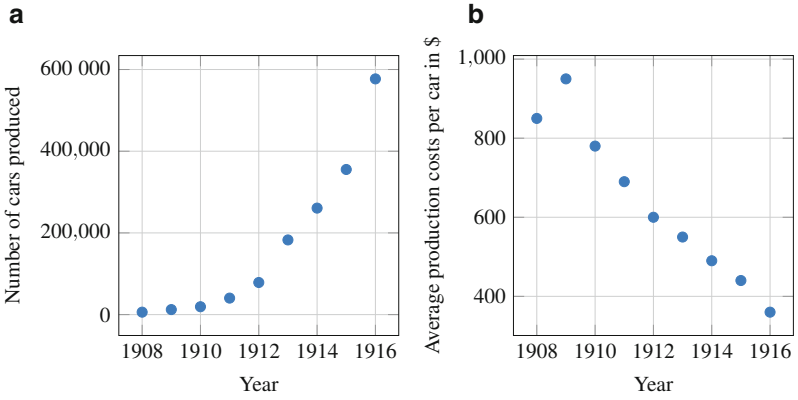


Fig. 2.3 Variation of the production and of the cost of the Ford "Model T" [1]. (**a**) Production volume per year. (**b**) Production costs per year

2.3 Taiichi Ōno and the Toyota Production System

When the Fordist model started to decline, a manager of Toyota, Taiichi Ono, had a revolutionary idea. He stopped to focus on the minimization of costs through the production of high quantities. Rather, he tried to align the production to what the customers effectively required and to strive for the maximum efficiency in producing that. His idea is that higher levels of efficiency require:

- to understand what produces perceivable value to the customers and
- to have a constant monitor of the production process, so that anything that does not produce value is eliminated.

These two actions had three major needs:

- to focus on the customer, who best understands what he needs;
- to empower all the employees throughout the development process, from the manager down to the janitor; and
- the employees who are the best people know what they do and can spot any area of waste.

Instead of focusing on the minimization of costs through the production of high quantities, the objective of Toyota was to align the production to what the market effectively required and to strive for the maximum efficiency in producing that.

The Toyota Production System tries to maximize efficiency by eliminating all activities that do not produce value to the customer. "Value" means the perceived benefit by the customer and influences their satisfaction with the product and, finally, what customers are willing to pay for it. Activities that do not produce value to the customer, i.e., are wasting resources and should be removed.

This philosophy promoted by Toyota came back in the 1990s with the book "Lean Thinking" by Womack and Jones [27]. "Lean Thinking" generalizes the

ideas introduced by Toyota to achieve "Lean manufacturing" and brought the Lean idea into new industries such as the pharmaceutical industry [15] or software development [17].

The Toyota Production System continuously focuses on the activities that provide value for the customer and removes unnecessary ones. This improvement is pursued in a continuous, incremental way. Not radical changes such as in business process reengineering [11] but constant improvements through optimization are the goal.

Womack and Jones identify five steps to enact Lean Thinking [27]:

1. **Specify value from the standpoint of the end customer**: understand what is valuable for the customer, why the customer is willing to pay money for a certain product or service;
2. **Identify the activities along the production process that contribute in creating what is valuable from the standpoint of the end customer**, i.e., identify all the steps in the value stream;
3. **Align the value-providing steps in a way that every product and service is built or provided along a simple, predefined path**, i.e., make the value-creating steps flow toward the customer;
4. **Start an activity only at the moment that it provides value to a concrete customer requirement**, i.e., let customers "pull value" from the next upstream activity; and
5. **Pursue perfection**: continuously improve.

These aspects will be described in detail below: In Sect. 2.4 we describe how to specify value from the standpoint of the end customer (step 1). Later in the chapter we classify activities whether they contribute or not in creating what is valuable from the standpoint of the end customer (step 2). Section 2.5 describes how the Toyota Production System involves workers to constantly improve the production process (steps 2 and 5). Section 2.6 describes the alignment of the value-providing steps using a pull strategy (steps 3 and 4). The Sects. 2.7 and 2.8 describe how activities can be coordinated so that they occur just-in-time to deliver the right parts and information at the right moment at the right place (step 5). Finally, Sect. 2.9 describes which approaches the Toyota Production System uses to continuously improve (step 5).

2.4 Creating a "Radiography" of the Production Process

The first step to understand which activities are not necessary is to understand what is important from the standpoint of the end customer. It is necessary to understand why the customer is willing to pay money for a certain product or service.

For this reason, the Toyota Production System constantly analyzes the ongoing activities to identify:

- **waste of overproduction**, i.e., output that is not required and not valued by the customer;

- **waste of waiting**, i.e., output that unnecessarily has to wait for a limited resource to be further processed. During this time the output can generate a set of costs, e.g., it might block other resources or might become obsolete;
- **waste of transportation**, i.e., unnecessary transportation of goods within the production process. Transportation costs can be lowered by aligning the production process in a way that subsequent processing steps are carried out by collocated machines;
- **waste of processing**, i.e., unnecessary processing activities. Understanding the processing costs is the basis to understand the contribution of each activity to the total production cost and to evaluate possible improvement actions;
- **waste of inventory**, i.e., keeping an unnecessary amount of unfinished products in inventories. Unfinished products that are kept in inventories produce management costs, which grow in relationship to the inventory size;
- **waste of movement**, i.e., the unnecessary movement of people or machines during the production process;
- **waste of making defective products**, i.e., the costs deriving from producing products that cannot be sold due to defects; and
- **waste of talent**, i.e., the under utilization of worker knowledge and skills.

A prerequisite for the identification of wasted resources is a clear understanding of the costs and benefits of the performed activities. In the same way as wastes are analyzed, problems have to be analyzed thoroughly to understand causes and to be able to avoid the problem in the future.

As a starting point, the Toyota Production System advices to analyze the "value stream": all the required steps (value adding and non-value adding) to bring a product from raw materials to the customer are captured. The result of this analysis is used to identify methods to eliminate waste in the current production process.

To understand the value stream includes walking to the place where the production takes place, talking to workers, and closely observing how a product is actually made from start to finish. Figure 2.4 shows an example for a value stream. Four different value streams produce the raw materials that are then used to produce cans of Coke. Table 2.1 shows the analysis of the value stream of one box of Coke cans in detail: for every step it shows the time the raw material waits in the incoming goods inventory, is processed, and waits in the finished goods inventory and how much is spoiled of the initial amount of aluminum.

Looking at Table 2.1, we can already see some wastes:

1. the difference between effective productive work (3 h) and the total cycle time (11 months) is very high: 99 % of the time nothing happens;
2. the aluminum and the cans are carried through 12 storage facilities before they are offered to the customer (cells marked with ①);
3. the cans are packed and unpacked three times on pallets (cells marked with ②); and that
4. 4.24 % of the extracted aluminum is lost because cans are damaged when packed empty at the can maker.

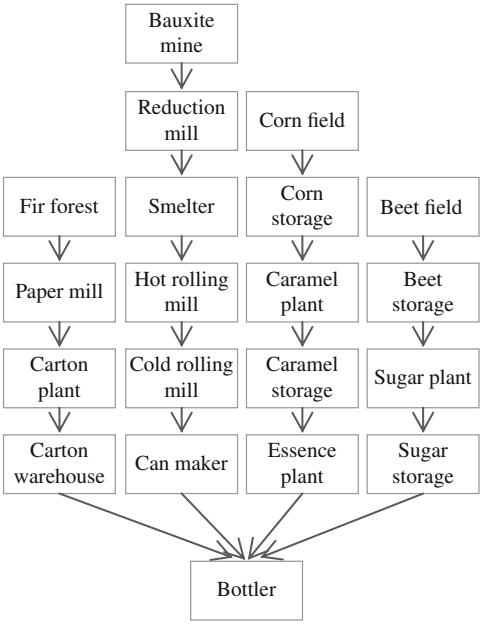


Fig. 2.4 Value stream for Coke [27]

Table 2.1 The value stream for one box of Coke cans [27].

Step	Time spent in			Spoilage in % of the initial amount of aluminium
	Incoming goods inventory (weeks)[a]	Processing duration (min)	Finished goods inventory (weeks)	
Bauxite mine	0	20	2	0
Reduction mill	2	30	2	0
Smelter	12	120	2①	2
Hot rolling mill	2①	1	4①	4
Cold rolling mill	2①	1	4①	6
Can maker	2①	1	4①②	20
Bottler	0,6①②	1	5①②	24
Distributor	0①②	0	0,4①②	24
Store	0①②	0	0,3	24
Household	0,4	5	–	–
Total	5 months	3 h	6 months	24

[a] Including the time needed to transport the goods from the preceding step

The analysis of the value stream provides a picture of the value that the company provides to the end customer. This helps to classify activities into three groups:

- activities that **add value**
- activities that **do not add value**; these activities can be divided into:
  - activities that provide no value but **are needed**, e.g., because of current technologies, law requirements, or production methods;
  - activities that provide no value and **are not needed**, i.e., that can be removed.

The multi-level pie in Fig. 2.5 shows this classification: the inner ring divides all activities into "non-value-adding" activities and "value-adding" activities. The non-value-adding activities are then (in the outer ring) divided into needed and not needed activities.

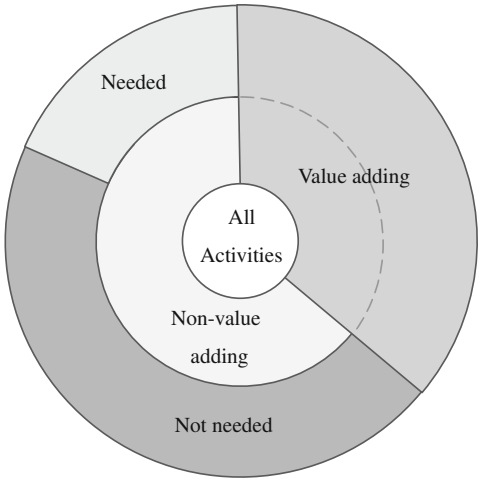


Fig. 2.5 Classification of activities as value adding or non-value adding [14]

The advice that the Toyota Production system gives is summarized in the decision matrix in Fig. 2.6: Activities that do not add value and are not needed should be removed; if they are needed but do not provide value, they should be reduced, reengineered, and performed with the maximum efficiency to minimize the wasted resources. If activities are adding value, they should be constantly improved and performed with improving quality and efficiency.

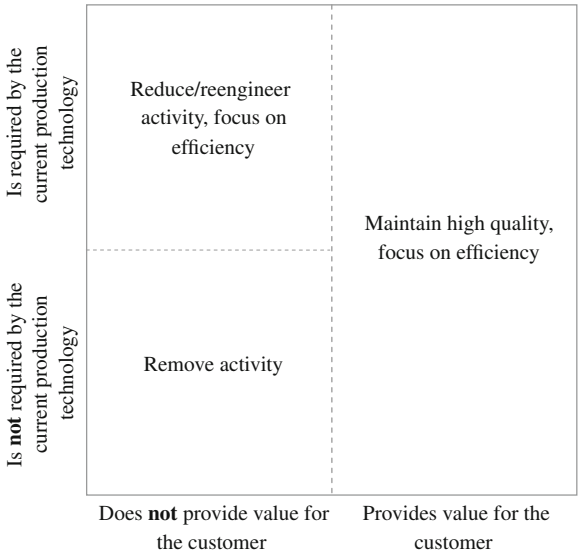


Fig. 2.6 Classification of activities as decision matrix [14]

2.5 Worker Involvement

Workers are directly involved in the production process. They obtain a good understanding of the performed activities and their benefit to the final output. This understanding can be used to improve current practices and to increase the quality of the output.

The content, sequence, timing, and outcome of each production step are stated on a standard work sheet by the workers themselves [14]. Standard work sheets have a twofold function: First, they help to assess the current activities by workers. This creates awareness on how currently time and resources are spent. Second, they function as the instrument to standardize work, i.e., prevent workers from using different approaches (such as a different sequence) to perform their work.

The importance of standardization is pointed out by [23] variations in the way the work is done is seen to hide the link between the performed work and the obtained results. Only if the work follows a well-defined sequence of steps that it is instantly clear when they deviate from the specifications.

If the work, e.g., is organized as a sequence of ten tasks, each of them expected to be completed in 20 s, and one worker is doing task no. 5 before task no. 2, then the sequence is not as planned and something must be wrong. In the same way, if after 50 s, the worker is still busy with task no. 1, then again something is wrong.

In this way, the standard work sheet can be used as a visual control mechanism: when a problem is detected, the worker and supervisor on the occurrence of a

problem can decide what to do to prevent a recurrence of the problem—to change the specifications, to change the production parameters, to retrain the workers, etc.

Since the standard work sheet defines how everybody is working, the analysis of the reason for errors in the process output can be done analyzing the standard work sheet. Through the continuous adaption of the standard work sheet to the working practices, it allows to capture the experience of the workers and to enable learning within the organization.

Figure 2.7 shows a possible way to specify the expected effort for an activity that consists of four steps. The expected duration is marked with a solid line; the movement (walking) to the next station to perform that step is marked with a dotted line.

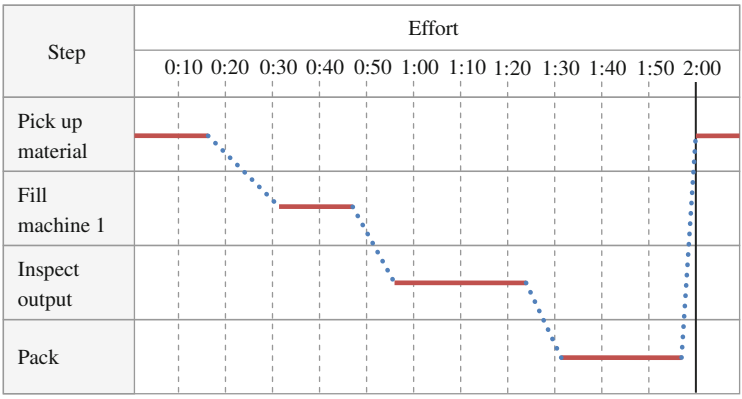


Fig. 2.7 Standard effort sheet [18]

The Toyota Production System categorizes within the categories shown in the circle diagram in Fig. 2.8: all activities performed by a worker are considered either work or waste (the inner ring). All needless activities (the "waste" slice) are superfluous and have to be eliminated.

The remaining part of the activities (the slice "work in general") is separated into value-adding work, i.e., work that is creating the value required by the customer and work that is needed because of the current working conditions. Examples include going to another room to pick up a spare part, opening the package of goods ordered from outside, operating machines, etc. It is the long-term goal to remove all non-value-adding activities.

A particular attention is given to worker movements. Moving is not necessarily providing value. It is part of value-adding but also of non-value adding work. Only if it contributes towards completing the job, it should remain part of the production process.

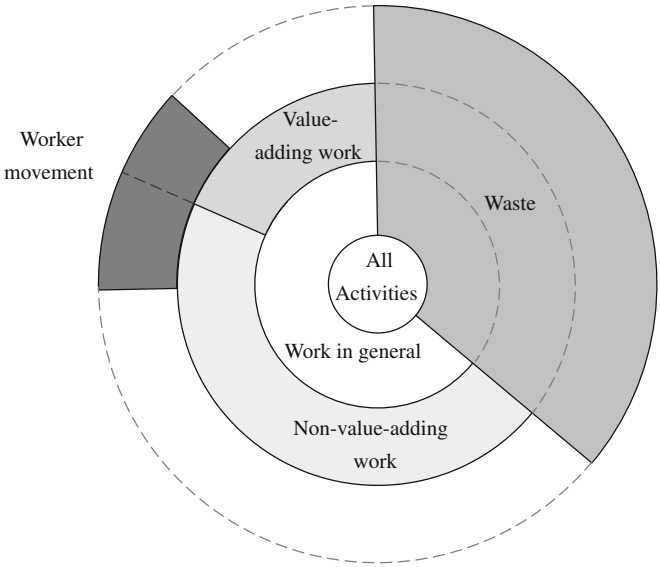


Fig. 2.8 Classification of work as value adding or non-value adding within the Toyota Production System [14]

Workers are organized within teams. The number of workers within each team is determined by the types of problems expected to occur, the level of assistance the team members need, and the skills and capabilities of the team’s leader [23]. Teams are held responsible for the output, not the individual [14].

2.6 “Pull” and Not “Push”

In traditional manufacturing, the parts used during the production process are produced independently by every work station, temporarily stored in inventories, and used from there when needed. This approach has the advantage to decouple the production of one work station from the remaining ones, but it requires the handling of inventories, which produces a series of costs, e.g. [14]:

- opportunity costs: the costs of using resources like a storage room to keep goods that are not yet requested by customers instead of using these resources differently;
- risk costs: the costs that arise to avoid risks or to overcome the unfavorable event, e.g., an insurance against fire;
- conservation costs: the costs of conserving the goods in a good state, e.g., heating;
- shrinkage costs: the costs of shrinkage of the goods present in stock because of reasons different from consumption like theft, administrative errors, or spoilage;

- management costs: the costs of managing the inventory and keeping track of the amount of shrinkage, what is stored, what is consumed, and so on; and
- handling costs: the costs of moving products to, from, and within the inventory.

A decoupled production makes it difficult to understand if and how much value is provided by every single production step. The Toyota Production system changes this approach with the ambition to clearly define a "causality chain" where the contribution of each activity to the final output becomes clear.

To achieve this, a coordination mechanism is sought that triggers all activities that are necessary to deliver a certain product or service. Traditionally, this is done using a "push" approach: the previous processes produce parts and "push" them to the later ones. These, as soon as parts are available, process them further, and so on.

To obtain a coordination mechanism that creates a dependency between those steps that produce value, a system is introduced that inverts the traditional coordination mechanism: not the earlier processes trigger later processes as soon as they are finished, but the later processes, as soon as they foresee a given demand, "pull" (request) the needed parts from the processes producing them.

This coordination mechanism is based on the idea that every process that needs a specific part requests the type and amount of parts needed from the competent upstream process.

This mechanism is introduced using the main production line as a starting point. The production plan indicates the needed output as well as the needed quantity and type of supply parts. To obtain the parts used to produce the final output, earlier processes are notified of this need and only the required amount is produced and provided to the main process.

This inverted coordination mechanism creates a chain of customer-supplier relationships along the activities that contribute to create the value for the end customer. The end customer signals the request for a product or service; this request goes to the last step in the production process which activates ("pulls") the upstream activities necessary to provide the requested item.

The way to trigger an upstream process chosen by the Toyota Production System is to send a message using a card ("Kanban" in Japanese). The example card in Fig. 2.9 shows a retrieval Kanban that is used to retrieve 25 metal rings of size 5 from the production location "Machining M9" and to deliver it to "Assembly A7."

The Kanban coordination mechanism forces the company to create a clear production process with clear connections between each production step. Processes requiring parts for their work retrieve (pull) these parts from preceding processes (see Fig. 2.10).

An example of a production process consisting of three steps (production, packaging, delivery) is shown in Table 2.2. For every production step also who acts as a customer and who acts as a supplier are also specified.

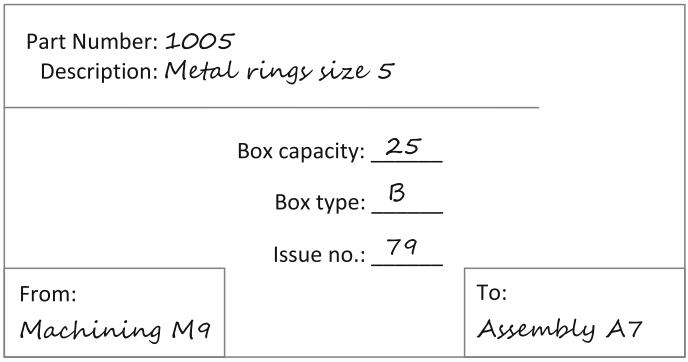


Fig. 2.9 Example of a Kanban [18]

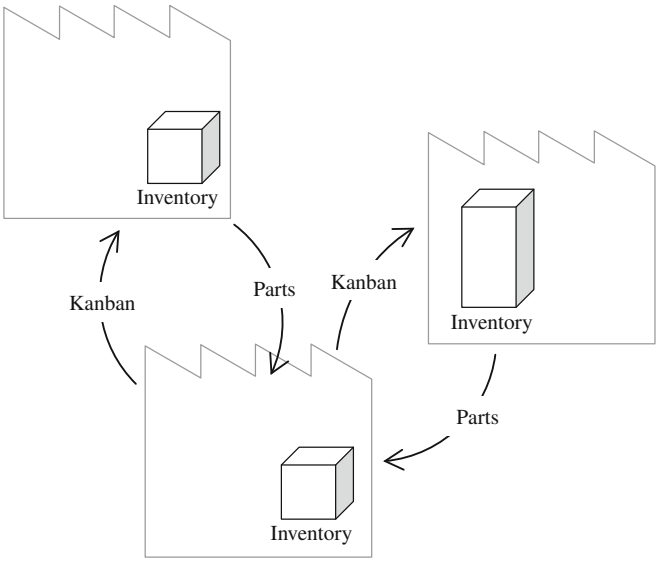


Fig. 2.10 Pulling parts from upstream processes

2.7 The Right Parts at the Right Moment at the Right Place

The Kanban system is used to clearly connect dependent production steps. The inventories shown in Fig. 2.10 represent the in-process inventories, i.e., partially completed goods, parts, or sub-assemblies that are no longer part of the raw materials inventory and not yet part of the finished products inventory.

Inventories per se do not provide value to the customer (he does not explicitly pay for it; they do not contribute directly to the outcome); they are needed because of the current production methods. They are set up because of fluctuations in the

Table 2.2 Example of "pulling" upstream activities.

Step	Activity	Customer	Supplier
1	A customer requires a product.	Customer	Delivery
2	The delivery process obtains the required item from the production process, packages it, and ships it to the customer.	Delivery	Production
3	The production process obtains the necessary raw materials from the suppliers, builds the requested item and ships it to the delivery process.	Production	Suppliers

demand of a certain resource and guarantee an uninterrupted production process. This means that the production of needed parts occurs earlier than required and the in-process inventories are filled.

The amount of items kept in inventories could be minimized or even avoided by finishing the production of the needed parts just before they can be further processed, i.e., just in time. "Just-in-time production" means that the parts needed during the production process are made available at the right place, at the right moment, and in the right quantity.

From a management perspective, designing and realizing such a process are complex, e.g., one mistake could delay the execution of all following activities and would require a revision of the initial plan to compensate the effects of the delay. It would be desirable to organize the activities so that a delay of one activity does not require to revise the production plan by some central authority but that the execution of the remaining activities automatically adapt to the new situation.

One way to achieve this is exactly by decoupling all production steps using inventories and to push the produced items to the next step. To avoid the usage of inventories, a way of organizing the production flow is needed, which is not based on a predefined coordination mechanism but on a mechanism that is able to adapt itself to unexpected events.

To create a self-adjusting system to handle the production of parts "just in time," the Toyota Production System extends the Kanban system to "pull" parts and services from upstream processes (and in this way) to trigger their production.

Just-in-time production aims to perform activities only at the moment and in the amount that is currently needed, i.e., is providing value. Within just-in-time production, Kanban has the function to provide pick-up or transport information, to trigger production, to create visibility, and in this way to prevent overproduction and excessive transport (only items with a Kanban are produced or transported).

Moreover, they prevent the production of defective products: if the pulled or the subsequent processes notice that products are defective, they can immediately stop the production flow and investigate to find a solution. Through the Kanban it will be possible to identify the process that produced the defectives, and since no products are produced in advance, these (now discovered as defective) items do not have to be thrown away.

Ideally, using just-in-time, products are built exactly in the order they are ordered. As a consequence, just-in-time requires a higher flexibility from the preceding processes. The "setup time," i.e., the time a process needs to switch from producing one item instead of another, has to be as short as possible.

Traditionally, because of long setup times, the amount of same items produced has to be as high as possible so that the long setup time pays off (see Fig. 2.11, *first line*).

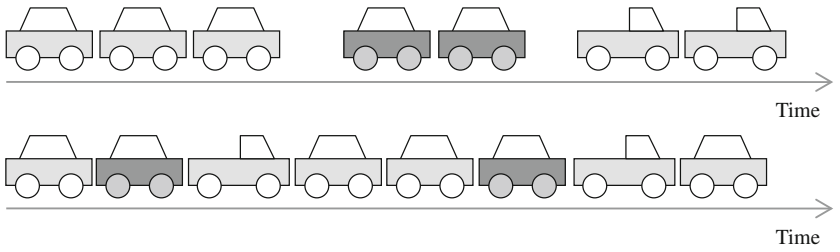


Fig. 2.11 Just-in-time production requires quick setup times

If it is possible to reduce the setup times, it is possible to produce items as they are required by the following processes (see Fig. 2.11, *second line*). This approach avoids overproduction: only those parts effectively used by subsequent processes can be produced. Moreover, the production of defective products is reduced: if the subsequent process notices an incompatibility or a mistake in the previous process, it is not necessary to throw away all the previous items produced on stock [14].

The Toyota Production process goes even further: it deliberately avoids to produce the same type of product in batches to minimize in-process inventories [14]. If items are produced as in Fig. 2.12, items are produced on stock, which produces the costs mentioned above. This example assumes that products 1 and 2 are sold on a constant rate, but they are produced in this way to minimize costs.

If items are produced in small lot sizes, i.e., in small quantities, before switching to the next required item (as shown in Fig. 2.13), the inventory size (and its costs) can be reduced. As above, a prerequisite for this is a constant demand by the customers. If there are peaks in the demand, a buffer inventory is nevertheless needed [8].

The Toyota Production system particularly stresses the importance of avoiding overproduction since it causes a set of consequences like the creation of inventories or the waste of resources because of the production of defective products. Therefore, the alignment of all production steps to the demand is pursued.

Figure 2.14 shows an example in which the production of step 2 is the bottleneck (producing only 30 items per hour), while step 1 is producing 45 items per hour. This means that step 1 will produce items at a faster rate that they are used, which causes costs to handle this excessive production.

To align the production to the demand coming from the end customer, the speed of the just-in-time production is aligned to the speed of the demand. The production

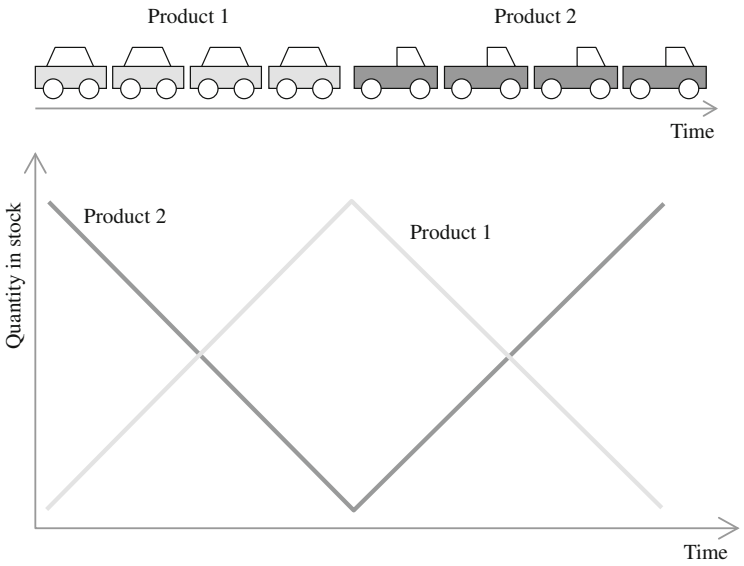


Fig. 2.12 Producing items on stock [8]

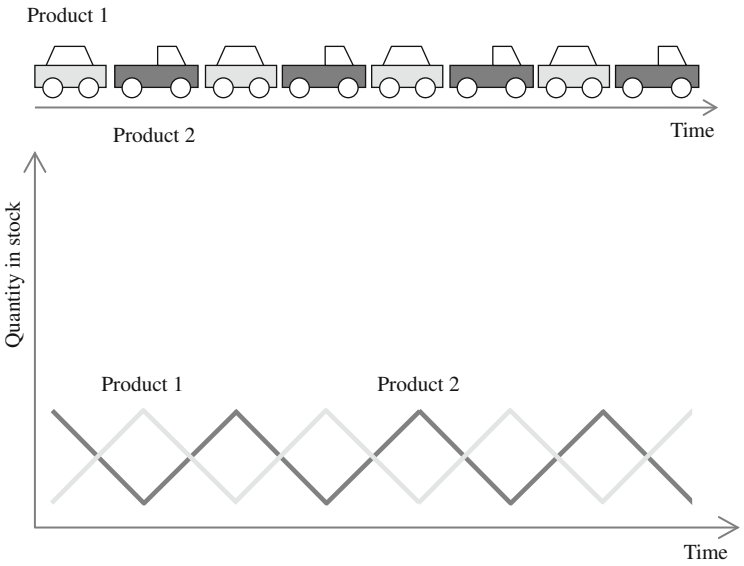


Fig. 2.13 Producing items just in time [8]

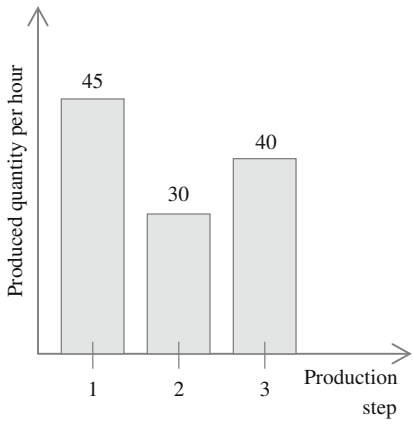


Fig. 2.14 Unleveled production [8]

is aligned to "takt time," i.e., the total amount of products required divided by the total amount of hours available for production. If, for example, the sales department reports that 120 items of a certain product are needed per month and the production is effectively operating 6 h (360 min) per day (this is the available time for work to be done, already without break times and other interruptions such as meetings, etc.), this results in a takt time of 360/120, i.e., 3 min per item. All production steps are now aligned so that every 3 min, one item can be produced.

In the example of Fig. 2.14, the production is unleveled: it will require to create inventories to produce parts at the stated quantities. Figure 2.15 shows this example after leveling the production steps on a takt time of 42. The last production step, the one from which no other step depends, is allowed to be shorter than the takt time since this does not produce in-process inventory [8].

To achieve this, the different processes have to be reorganized (e.g., by slowing down one process but using less workers or expanding the capacity of another process adding workers or machines, and so on) so that all proceed at the same pace.

2.8 The Right Information at the Right Moment at the Right Place

Exactly as the just-in-time production of goods avoids costs, the just-in-time delivery of information is advocated by the Toyota Production System: "Is it really economical to provide more information than we need—more quickly than we need it? [14]."

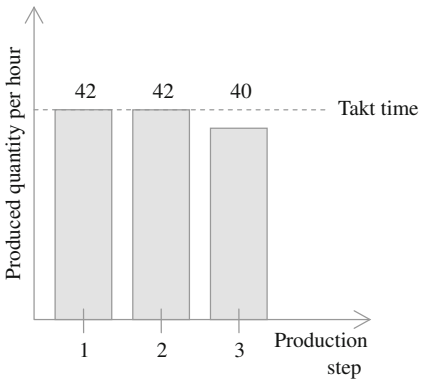


Fig. 2.15 Leveled production [8]

Figure 2.16 shows an automobile assembly line with three subprocesses. Car number 1 is about to exit the main production line and the chassis number 12 just entered. The production schedule is sent to the main production line which attaches to each chassis all the information needed for its production. The subprocesses A, B, and C obtain the needed information through a Kanban as soon as they need it: process A three steps in advance, process B two steps in advance, and process C five steps in advance.

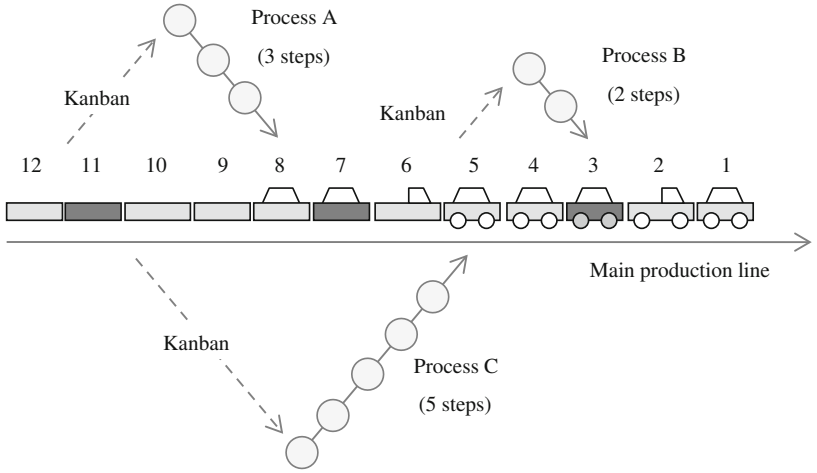


Fig. 2.16 Automobile assembly line [14]

The key idea of the Toyota Production System is to suppress excess information [14]. The production plan is only sent to the main production line. All the needed information is carried by the products being produced while the subprocesses are coordinated through Kanbans.

2.9 Quality Management

Quality plays a central role within the Toyota Production System: the success on the marketplace is seen as a result of the quality of the products and services which are the determining factors for customer satisfaction. This assumption drives the constant search for improvement throughout the company.

Moreover, the quality of the final product is seen as a result of the process producing them. This assumption creates a high attention for a high-quality production process according to the credo "prevention is better than healing" (see Fig. 2.17).

Quality management within the Toyota Production System is done on a continuous basis. The goal is to constantly improve without disrupting the current process [14]. The process used for continuous improvement was first discussed by Shewhart [21] and promoted by Deming [7] as a systematic approach to problem solving.

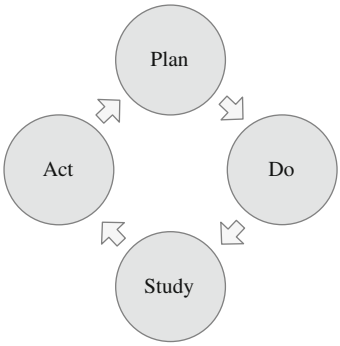


Fig. 2.17 PDSA (Plan-Do-Study-Act) within the Toyota Production System

The process consists of four steps: plan the activities to perform and their expected outcome; execute the plan (do); study the outcome and compare it with the expected outcome, i.e., understand how and why the realized result differs from the expected one; and confirm the plan or adjust it (act).

The expected result of applying the PDSA paradigm is a controlled process, i.e., a process that—within certain limits—produces predictable results. Predictable means that it is possible to state—at least approximately—the probability that the observed phenomenon will fall within the given limits [20].

The PDSA process follows the methodology for conducting experiments used in science: hypothesis (plan), experiment (do), and evaluation (study). Some examples of how people conduct experiments within the Toyota Production System are given in Table 2.3 [23].

Table 2.3 Experiments within the Toyota Production System [23]

Hypothesis	Signs of a problem	Responses
The person or machine can do the activity as specified in the standard worksheet	The activity is not done as specified	Determine the true skill level of the person or the true capability of the machine and train or modify as appropriate
If the activity is done as specified, the good or service will be defect free	The outcome is defective	Modify the activity
Customers' requests will be for goods and services in a specific mix and volume	Responses do not keep pace with requests	Determine the true mix and volume of demand and the true capability of the supplier; retrain, modify activities, or reassign customer-supplier pairs as appropriate

The Toyota Production System advices to ask why five times about every matter, particularly to identify the root cause of problems. If, for example, a machine stopped to work, the way to understand how to solve the underlying problem is to ask five times why as in the following dialog [14]:

1. "Why did the machine stop?" "There was an overload and the fuse blew."
2. "Why was there an overload?" "The bearing was not sufficiently lubricated."
3. "Why was it not lubricated sufficiently?" "The lubrication pump was not pumping sufficiently."
4. "Why was it not pumping sufficiently?" "The shaft of the pump was worn and rattling."
5. "Why was the shaft worn out?" "There was no strainer attached and metal scrap got it."

The knowledge acquired in this way can be used to understand how value is created, where resources are wasted, and what are the underlying causes when problems occur.

Standardization plays a crucial role in the Toyota Production System. It is the form how the company learns. Whenever a new aspect is learned, the standards (such as the standard work sheet) have to be updated to reflect the new method of production. Only through this the company can learn. Figure 2.18 shows a frequent way how this idea is depicted: the Plan-Do-Study-Act cycle helps the company achieve progress, but only standardization prevents the company from falling back to initial quality levels.

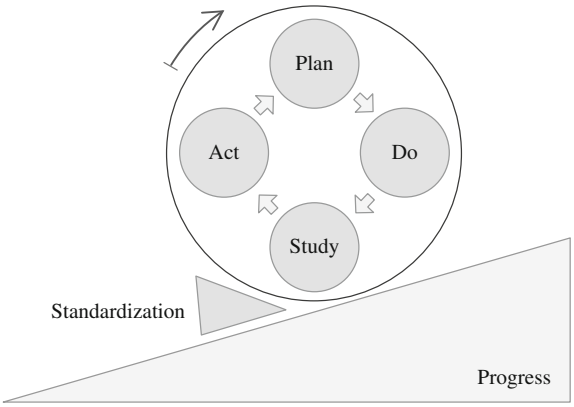


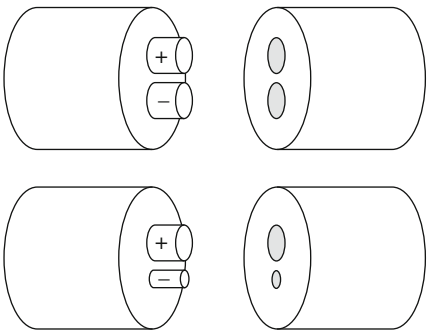
Fig. 2.18 The role of standardization [25]

Standardization is the instrument to enable organizational learning. It is the formalization of knowledge. The adoption to a standard means to benefit from all the experience packaged into it and is seen as the basis to proceed in improving organizational performance.

The Toyota Production System puts a strong focus on the modification of equipment, tools, and processes to embed the standards into it so that the compliance to the standard happens automatically, i.e., is error proof.

Figure 2.19 shows first two connectors that can be attached in different ways; this can lead to mistakes. The second pair of connectors allows only one way to be connected.

Fig. 2.19 Error-proof connectors [18]



Embedding standards within the equipment means to avoid the costs of checking if the work is compliant to the standard as well as the costs to correct existing work, not respecting the standard.

If it is not possible to embed the standard into the equipment, visual control mechanisms are embedded in the tool or equipment so that its user gets aware of a mistake or a regulation without any effort. Figure 2.20 shows an example where the worker immediately sees that the scissors are missing or where they should be put when they have to be returned to the drawer.

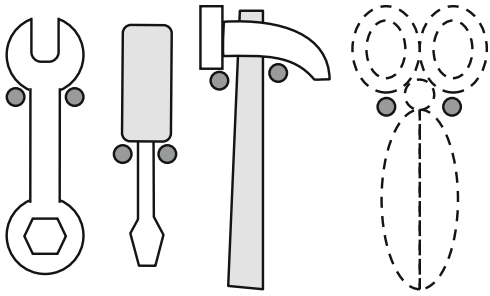


Fig. 2.20 Visual control for equipment [18]

To embed standards and knowledge into processes, the Toyota Production System uses the concept of autonomation, that is, structuring the processes so that a machine or a procedure are able to detect a problem in the produced output and interrupt production autonomously rather than continue to run and produce bad output [13, 14].

Autonomation, together with just-in-time production, is considered one of the pillars of the Toyota Production System [14]. Figure 2.21 shows the concept of autonomation through a simple autonomation system where the stop mechanism is triggered because one part does not meet the size specifications.

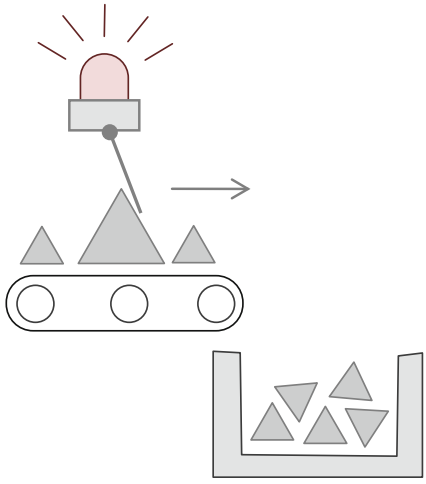


Fig. 2.21 Using autonomation to ensure quality

Some authors use the term "quality-at-the-source" for autonomation [24], meaning that quality is inherent in the production system and is not checked after the process. Autonomation means that a machine is complemented with two additional parts [13]: a mechanism to detect problems, i.e., abnormalities or defects, and a mechanism to interrupt the production line or machine when a problem occurs.

The initial idea came from the construction of a weaving machine by Toyoda Sakichi, the founder of the Toyota Motor Company, that could stop instantly if any one of the threads broke. This machine contained a mechanism that could detect problems and stop the machine to prevent the production of defective products. In this way, the quality control after the production as well as discarding defective output was avoided.

The use of autonomation has various organizational effects: no operators are needed to constantly check the quality of the output of a given process and only when an error occurs an operator has to be available. In this way one operator can be in charge for several machines. Moreover, stopping the machine when a problem occurs creates visibility and motivates everyone to clearly understand the problem.

On the basis of this thought, the Toyota Production System advises that every worker should be given the possibility to push the stop button to halt the production if any abnormality occurs.

The goal of quality management is to obtain the "ideal" output, i.e., the output of an ideal person, group of people, or machine [23]:

- is defect free;
- can be produced and delivered just in time in the version requested;
- can be produced without wasting any materials, labor, energy, or other resources (such as inventories); and
- can be produced in a work environment that is safe physically, emotionally, and professionally for every employee.

2.10 Summary

The "Leitmotiv," the guiding theme for the Toyota Production System, is that an activity should be only carried out, if it provides value to the customer. The consequences of this are manifold:

- it is necessary to understand the contribution of each activity to the value for the customer (see Sect. 2.4);
- it is necessary to question current production methods to find new approaches that allow to eliminate all those activities that are not needed or only needed because of the current production method (see Sect. 2.4);
- it is necessary to involve workers in this assessment process because they are directly involved with the work and know the best of what is really needed and what is not (see Sect. 2.5);

- it is necessary to create a visible link between the creation of the value for the customer and all the carried out activities within the company: this is achieved implementing a "pull" mechanism (see Sect. 2.6); and
- the goal is to make only "what is needed, when it is needed, and in the quantity that is needed" just-in-time production (see Sect. 2.7) to minimize overproduction;

The Toyota Production System favors quality management based on the Plan-Do-Study-Act approach to promote organizational learning through standardization. This approach promotes the creation of quality control mechanisms like the standard work sheet or autonomation and accepts to stop the entire production when an error has to be identified and fixed in exchange for a low defect rate of the output.

Problems

2.1. Compare a street crossing based on traffic lights with a roundabout (see Fig. 2.22).

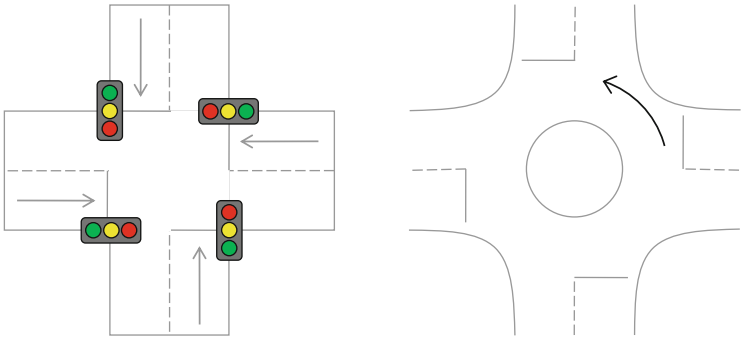


Fig. 2.22 Traffic lights vs. roundabout

Which one is safer and which has a higher throughput? What are the parallels between the waterfall software development model and a Lean model?

2.2. Harry Sneed has come up with a model called the "devil's square" [22] to express how hard it is to provide the right quality and the right amount of features and be on time and on budget (see Fig. 2.23).

The devil's square claims that it is not possible (or at least hard) to arbitrarily choose the quality, scope, development time, and the development costs of a software project since the productivity (represented by the gray surface area) is supposed to stay invariant in the short term.

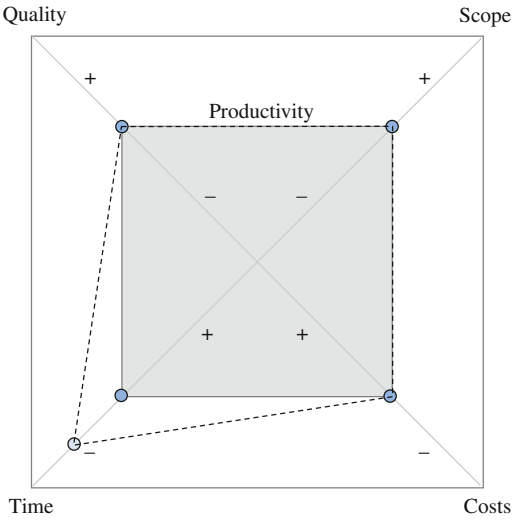


Fig. 2.23 The devil's square [22]

Imagine now a client wants that a project is finished earlier than planned. It is like he wants to drag a corner of the productivity rectangle on one side—the "time" side—towards the minus (depicted as the dashed rectangle in Fig. 2.23). The devil's square tells us that the surface of the gray area has to remain the same; therefore, one of the other edges has to move towards the center:

- either the quality diminishes, or
- the scope diminishes (fewer features), or
- the costs increase.

What is the goal of Lean in this context? What are the expected effects to the surface of the rectangle when a company introduces Lean?

References

1. Ayers, E.L., Gould, L.L., Oshinskyand, D.M., Soderlund, J.R.: American Passages: A History in the United States: Since 1865. Cengage Advantage Books. Cengage Learning, Boston (2009)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley, Reading (1999)
3. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Comput. **21**(5), 61–72 (1988)
4. Boone, L., Kurtz, D.: Contemporary Marketing, 2013 Update. Cengage Learning, Mason (2012)
5. Corral, L., Janes, A., Remencius, T.: Potential advantages and disadvantages of multiplatform development frameworks–a vision on mobile environments. Procedia CS **10**, 1202–1207 (2012)

6. Corral, L., Janes, A., Remencius, T.: Potential advantages and disadvantages of multiplatform development frameworks–a vision on mobile environments. In: Proceedings of the International Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments (SUPE), Ontario, Canada (2012)
7. Deming, W.E.: Quality, productivity, and competitive position. Massachusetts Institute of Technology Centre for Advanced Engineering Study (MIT-CAES), Cambridge (1982)
8. Drew, J., McCallum, B., Roggenhofer, S.: Unternehmen Lean: Schritte zu einer neuen Organisation. Campus-Verlag, Frankfurt am Main (2005)
9. Ford, H., Crowther, S.: My Life and Work. Doubleday, Page & Company. Online: http://www.gutenberg.org/ebooks/7213 (1922). Accessed 4 Dec 2013
10. Keegan, J.: A History of Warfare. Vintage, New York (1994)
11. Malhotra, Y.: Business process redesign: an overview. IEEE Eng. Manag. Rev. **26**(3), 27–31 (1998)
12. Mills, H.D.: The management of software engineering, part i: principles of software engineering. IBM Syst. J. **19**(4), 414–420 (1980)
13. Monden, Y.: Toyota Production System, 2nd edn. Industrial Engineering Press, Norcross (1993)
14. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)
15. Petrillo, E.W.: Lean thinking for drug discovery — better productivity for pharma. Drug Discov. World **8**(2), 9–16 (2007)
16. Polybius: Histories. Macmillan, Reprint Bloomington 1962. Evelyn S. Shuckburgh, translator. Online: http://www.perseus.tufts.edu/hopper/text?doc=urn:cts:greekLit:tlg0543.tlg001.perseus-eng1:1.1 (1889). Accessed 1 May 2014
17. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, Upper Saddle River (2006)
18. Russell, R.S., Taylor, B.W.: Operations Management: Quality and Competitiveness in a Global Environment. Wiley, Hoboken (2005)
19. Savitt, R.: What they wrote about world war ii: the journal of marketing 1939–1946. In: Neilson, L.C. (ed.) Proceedings of the Biennial Conference on Historical Analysis and Research in Marketing (CHARM). CHARM Association, New York (2011)
20. Shewhart, W.A.: Economic Control of Quality of Manufactures Product. D. Van Nostrand Company. Online: https://ia601607.us.archive.org/8/items/bstj9-2-364/bstj9-2-364.pdf (1931). Accessed 4 Dec 2013
21. Shewhart, W.A., Deming, W.E.: Statistical Method from the Viewpoint of Quality Control. Dover Books on Mathematics Series. Dover, New York (1939)
22. Sneed, H.M.: Software Management. Verlagsgesellschaft Rudolf Müller, Köln (1987)
23. Spear, S., Bowen, K.H.: Decoding the dna of the toyota production system. Harv. Bus. Rev. **77**(5), 96–108 (1999)
24. Standard, C., Davis, D.: Running Today's Factory: A Proven Strategy for Lean Manufacturing. Hanser Gardner Publications, Cincinnati (1999)
25. Stewart, J.: The Toyota Kaizen Continuum: A Practical Guide to Implementing Lean. A Productivity Press Book. CRC Press, Boca Raton (2011)
26. Wikipedia contributors: Macedonian phalanx. Online: http://mk.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D0%B5%D0%B4%D0%BE%D0%BD%D1%81%D0%BA%D0%B0_%D1%84%D0%B0%D0%BB%D0%B0%D0%BD%D0%B3%D0%B0 (2013). Accessed 4 Dec 2013
27. Womack, J.P., Jones, D.T.: Lean Thinking: Banish Waste and Create Wealth in Your Corporation, 2nd edn. Free Press, New York (2003)

Chapter 3
Towards Lean Thinking in Software Engineering

Quod potest fieri sufficienter per unum, superfluum est si per multa fiat.
(If a thing can be done adequately by means of one, it is superfluous to do it by means of several)

Thomas Aquinas, Summa Contra Gentiles, Ch. 70

On the street went Uli without thinking, and then to Nausicaa park, a few yards away. He picked up his mobile phone. He sat on a bench, made sure that there was none around, took a deep breadth and called XXX. What?! Yes, he called XXX.

The phone rang. XXX looked at the display and was astonished to see the name of Uli. He knew that Uli had already received the bad news. He was hesitant to answer. But then, the hell! "Uli, what do you want?" He expected to receive a sequence of insults, well, it was not the first time nor it was likely to be the last: it is the part of the life of a good manager, he thought. "Relax XXX! I am not here to complain for your decision nor to talk to you of the miserable life of my folks who are going to be be laid off." "So, what?" "I want to make a deal with you. I want to prepare a coffee for you, actually, an espresso, or, better, a lot of espressos, but one at a time." "Uli, are you OK?" "Yes, indeed, come and you will see. I am at Nausicaa Park, right where Zeta Str. crosses with 17^{th} Ave. Let us meet there in about one hour." XXX pondered a bit what to do... "well, I am a big boy and it is an open park... and, at the end, the situation is so bad that wasting one hour or so won't make it worse..." Still he let his secretary know where he was going, giving her instructions in case he would have not be reachable on the phone in a couple of hours.

XXX arrived right on time. There was not point to let Uli wait... Uli was waiting for him anxiously and greeted him smiling. "This guy may not be well," XXX thought "I have just fired his team and he smiles!" "Hi XXX, let us go to Phaeacians', they serve the best espresso in town, you will drink and understand." The reader can imagine what XXX thought. Still, Phaeacians' is a big place, with a lots of folks, so everything was still fine.

"XXX what would you like to have" "An espresso." "So, please, 5 espressos and 3 decafs." "Uli, are you OK? We are only the two of us?" "XXX, take it easy."

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_3

Uli started to talk: "Look at how espressos are prepared. They are made one at a time, according to the specs of each individual customer. It is not a coffee that is brewed, say, once per hour. The reason is that each customer might want it a bit different. Still it is coffee, but it is a peculiar coffee. And each customer really like it her or his way. So our 5 espressos and 3 decafs are prepared one after the other. The team is optimized to produce the specific, very specific value that each customer wants, and every customer is unique.

Also notice that all the workers are capable of doing almost everything, espressos, cappuccinos, lattes, but also regular coffees and stay at the cash, this is essential, as at any time of the day someone may come to ask for a very specific product. And it is not easy to make a good espresso or a good cappuccino. People need to be properly instructed when they get hired, but then they need to learn from each other. And, lastly, practice makes perfection. The difference between a good cappuccino prepared by a talented and experienced person and a cappuccino prepared by a novice is immense, the same difference between a piece of code wrote by, say, Kent Beck and by a freshman. The funny thing is that also the tools used here require experience. The coffee machines require a tuning and only after a while they reach their top performances.

So, we have three terms here: value, knowledge, and improvement. I want to struck a deal with you. You let us work again with you, and we will only concentrate on those feature that create direct value for you, one at a time, and we will deliver them to you one at a time, say, every two weeks, and you, indeed will pay only for the one you accept one at a time, with no upfront investment and the possibility to stop the work at any time with no penalties. It will be hard to convince Athi and J but this will be my task, and, clearly, I will make very clear with them that we will have the same opportunity of stopping the contract if it turns out that a given time is not any more beneficial for us. We will just ask you to be patient and not to interfere to much with the work. Well, at the beginning of each sequence of two weeks we will get together and discuss your priorities and the amount of work we can do in the two weeks, but then we will ask you to trust our ability to make informed and reasonable decisions within the two weeks so that we could concentrate on producing value for you. Still you will have the chance to come and see us and the system being developed as much as you want—I mean all the system, including the source code. And, I promise you, you will get every two weeks a release of something that provides value to you. I promise you." "Uli, you promised not to beg for another chance…" "Come on, XXX, this is not another chance! Consider how much you have already invested on us. We know your organization, we know your requirements, we know you, and we have a high esteem in you, even if we often disagreed on technical choices. But, if we both commit on releases every two weeks, you will be able to tell us immediately whether you like our espresso or we need to change something in the way we prepare it—the machine, the blend, the temperature, the cup…"

XXX considered that to get another team up to speed he would have needed at least one month and that Uli was not asking for money, unless he had produced a useful feature. So, the situation appeared weird but no risk was evident, just a bit of extra time to interact with these folks in this new funny way, but, if anything went wrong, well the agreement was that he could say "Stop". Also he liked the idea to be allowed to jump in their time at any time without any advance notice—he has always been curious to see how these nerds were developing software.

"OK, Uli. Let us try for two weeks, and then... But you talk to your bosses and explain them honestly what we discussed—no negotiation on my side and the right to stop at any time without any liability, paying only for what you have delivered to me and I have liked." "Deal!"

3.1 Introduction

Since its conception in the mid-1950s, Lean Thinking has been very successful in manufacturing: it has helped organization to focus on value-providing activities, to identify unnecessary ones, and therefore to increase the overall efficiency and effectiveness of development. So the obvious idea was to extend it also to the production of IT goods in general and specifically to software engineering. There are several proposals on how to translate Lean principles into software engineering practices, e.g., the Agile Manifesto [11], the pioneering work of Kent Beck [10], the work of Mary and Tom Poppendieck [41], and so on.

In theory, it would be a good point now to present existing approaches to Lean software development. We decided to do that only in Chap. 6 because before we want to look at the roots of Lean and see how they already in the past influenced different organizational practices. In part, this is because we do not want to present Lean software development as the next holy grail, but we want to look at its constituent parts or the perspectives one can look at Lean. After doing this, we can avoid the problems we encounter today with Agile Methods, which we will describe in Chap. 5.

The application of the principles of Lean management is a way to obtain software development processes that are focused on the value to provide to the customer and the constant aim to increase efficiency and effectiveness. To enable Lean software development, the company has to become aware of its internal processes, identify the value provided to the customer, and keep its own processes aligned to them to produce what was identified as valuable.

This idea is not new. Several approaches, from the software engineering domain and not, have already been proposed to help in this endeavor. Some—this list is not exhaustive—of them are listed in Table 3.1.

Table 3.1 Approaches that focus on similar values as Lean

Year	Proposal
1970s	Rapid application development (RAD) [35]
1985	Quality improvement paradigm [6]
1986	Scrum (product development) [49]
1986	Spiral software development [13, 15]
1987	Six sigma [39]
1987	Goal question metric approach [8]
1989	Experience factory [7]
1991	Capability maturity model (CMM) [40]
1992	Balanced scorecard [33]
1995	Dynamic systems development method (DSDM) [48]
1999	Extreme programming (XP) [10]
1999	Feature driven development [19]
2000	Adaptive software development [28]
2000	Personal software process [30]
2001	Agile manifesto [11]
2002	Capability maturity model integration (CMMI) [18]
2004	Scrum (software development) [44]
2004	Crystal clear [20]
2007	GQM$^+$strategies [9]
2009	DevOps [31]
2011	Disciplined agile delivery [4]

In this chapter we will look at Lean from three perspectives and relate them to the approaches of Table 3.1:

1. **Value**: methods that support the organization to focus on the understanding and maximization of the delivered value;
2. **Knowledge**: methods that focus on the creation of a shared understanding of the know-how, know-where, know-who, know-what, know-when, and know-why within the company [42]; and
3. **Improvement**: methods that help to instill a culture of constant improvement.

We look at each perspective in the following three sections.

3.2 Value

Lean Thinking was developed to overcome the scarcity of resources that occurred after World War II; its aim was to increase the efficiency [37]. Agility, also trying to focus on processes that consist only of essential, value-adding activities, was developed to address software development risks [10].

Agility, on the other hand, was developed to address the intrinsic variability and randomness of software development. In both cases the key idea is to focus on value, i.e., on activities that deliver business value to the customer. Such focus implies a

substantial evaluation of activities, a reduction that addresses the problem of scarcity of resources and also addresses the unpredictability of software development.

In fact, not addressing risks is inefficient if the costs of avoiding the risks are lower than the expected costs if the risk occurs (see the next section). In particular, changing requirements are a risk factor since they require the allocation of effort to provide value to the client. To address this risk, Agile Methods encourage the use of practices to "harness change for the customer's competitive advantage [11]."

One such practice is delivering prototypes early in the process to obtain feedback and to better know what provides value for the client. Rapid Application Development [35] was one of the first approaches proposing this.

As for Lean Thinking, the highest priority for the "Manifesto for Agile Software Development" [11] (which will be presented in detail in Chap. 4) is to deliver value to the customer.

Concrete instantiations of the principles stated in the Agile Manifesto include Scrum [44] (which originated from the product development method Scrum [49]), Crystal Clear [20], Extreme Programming [10], Adaptive Software Development [28], Feature Driven Development [19], and Dynamic Systems Development Method [48]. Extreme Programming is taken as an example to show how these principles are transformed into practices and will be presented in Chap. 4.

Around 5 years ago, DevOps [31] started to become popular, which builds on the Lean idea that upstream processes have to be aligned with what is needed by downstream processes. DevOps aims to close the gap between **dev**elopment and **ope**rations and integrates both sides on three dimensions [2]:

- **Process integration**: the processes, in which development and operations are both involved, e.g., the development of a solution into production or the solution of a problem discovered in production are integrated. In this context the term "continuous delivery" became popular, which describes a team that is able to continuously (i.e., very often) integrate, build, test, and deploy its software into production. The way how this is achieved is based on [29]:
  - the creation of a repeatable, reliable process for releasing software,
  - a high degree of automation,
  - a high degree of traceability,
  - continuous integration (see Chap. 4),
  - autonomation,
  - a clear definition of what "done" means,
  - a shared responsibility of the delivery process, and
  - continuous improvement.
- **Tool integration**: the tools that development and operations use are different. Through an integration of the tools, the collaboration can be increased. For example, a plugin written for the integrated development environment of the developer could access production configuration information to simulate a test using production parameters.
- **Data integration**: data that is collected by either team are shared or federated. For example, usage statistics that operations track as well as configuration/in-

stallation details could be shared with developers. Through the federation of this data, the consistency, accuracy, and therefore relevance of the data can be increased.

Based on various Agile Methods, also hybrid methods such as Disciplined Agile Delivery [4] were proposed. Scott Ambler defines Disciplined Agile Delivery (DAD) as "a people-first, learning-oriented hybrid Agile approach to IT solution delivery. It has a risk-value life cycle, is goal-driven, and is enterprise aware [3]." Among others, DAD is based on Scrum, Extreme Programming, and Kanban (see Chap. 10).

3.2.1 Risk as a Value-Maximizing Strategy

Risk is an event or a condition that may affect the outcome (i.e., the delivered value) of a project [47]. It influences the expected value; therefore, it is important to consider it during software development.

There are several risks in software production. Some of these include:

- the risk of not understanding the requirements of the customers;
- the risks that the customer changes her/his mind;
- the risks of choosing the wrong technology; and
- the risks of writing defective software.

The history of software development can be read as the history of how managers and developers tried to mitigate such risks.

Risk can be characterized by two distinctive elements: probability and impact [38]. One way to quantify the risk is to calculate the risk exposure [14]:

$$\begin{aligned}\text{risk exposure} = {} & \text{probability of an unsatisfactory outcome} \\ & \times \text{ loss to the parties affected if the outcome is unsatisfactory}\end{aligned}$$

The waterfall model was the first to address risks: the risk to develop a product that does not correspond to the requirements. The design was done only after the requirements were clear, the implementation was done after the design was completed, the system was then verified and tested at the end to again ensure that it did what it should do.

The waterfall model acts on the first part of the equation: its intention is to lower "the probability of an accident occurring" through a rigorous development process encouraging thorough planning to foresee every eventuality.

The argumentation behind this model is similar to that in the construction of houses: it happened that newly built houses had to be torn down after discovering that their foundations were not built on stable land; analyzing the ground before building the foundation would have helped to avoid this.

This approach works best when we know the probability of the different risks. In the case of construction, if we know that we are building a new house on a land that may not be stable, we reduce our risks analyzing the ground.

There is a joke circulating on the Internet with the title: "If architects had to work like software developers," which illustrates the difficulties of unclear requirements; see Appendix A. Such jokes unfortunately represent the reality for many developers and consultants and shows that the risk that the customer does not know what he wants is high.

Other plan-based approaches propose solutions that are not too different than the waterfall. In essence, the first term of the multiplication (the probability of an accident) seems hard to reduce.

The promoters of Agility work also on the second term of the multiplication, the impact of the accident. They take advantage of tools that have been used for more than 20 years in risk management (see also [36]) such as prototyping, risk analysis, and iterative development combined as in the spiral development model [13, 15].

Boehm defines the spiral development model in the following way [15]: "The spiral development model is a risk-driven process model generator. It is used to guide multi-stakeholder concurrent engineering of software-intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions."

The spiral software development model is an iterative software development approach in which each cycle shows the following characteristics [15]:

1. Concurrent rather than sequential determination of artifacts.
2. Consideration in each spiral cycle of the main spiral elements:
   - critical stakeholder objectives and constraints,
   - product and process alternatives,
   - risk identification and resolution,
   - stakeholder review, and
   - commitment to proceed.
3. Using risk considerations to determine the level of effort to be devoted to each activity within each spiral cycle.
4. Managing stakeholder life cycle commitments.
5. Emphasis on activities and artifacts for system and life cycle rather than for software and initial development.

The principles of Extreme Programming can be linked to risks they address (see Chap. 4 for a detailed description of the principles). We listed some risks that each principle addresses in Table 3.2.

Table 3.2 Risks addressed by extreme programming principles

Principle	Risk
Humanity	Costs and disruption of high turnover and missed opportunities for creative action
Economics	Costs of developing something with features or qualities not required by the customer (obtaining a "technical success" but an "economic failure")
Mutual benefit	Having win-lose situations in which not all stakeholders benefit from the collaboration, which will result in present or future costs [16]
Self-similarity	Cost of "reinventing the wheel over and over again"; in fact there is a widespread usage of patterns [27] in the Agile community
Improvement	Cost of repeating the same mistakes over and over because the team does not try to improve present practices
Diversity	Cost of overlooking strengths, weaknesses, opportunities, and threats
Reflection	Cost of repeating the same mistakes over and over because the team does not learn from past projects
Flow	Cost of wasting resources to develop a product that does not produce value for the customer because the team was waiting too long for feedback
Opportunity	Cost of wasting resources because the team does not improve its practices
Redundancy	Cost of "betting all on one card," i.e., one approach, and to have to redo everything after discovering that it does not match the requirements
Failure	Cost of not improving since the team does not risk to fail
Quality	Cost of wasting resources because of bad source code quality (leading to less predictable delivery) and the cost of losing customers because of bad quality (compare with "win-lose" situations in [16])
Baby steps	Cost of wasting resources because of an increase overhead of big changes compared to small changes
Accepted responsibility	Cost of wasting resources because of too much centralization

Let us recall the definition of risk exposure:

$$\begin{aligned}\text{risk exposure} = {} & \text{probability of an unsatisfactory outcome} \\ & \times \text{ loss to the parties affected if the outcome is unsatisfactory}\end{aligned}$$

Agility leverages on both sides of the equation: it tries to avoid activities that are not creating business value—so the probability of an accident is zero for those activities that are never performed at all; on the other side, it tries to prepare for the impact of the accident if it occurs (e.g., a change in requirements) to keep the loss as low as possible.

3.3 Knowledge

The problem of managing knowledge is strongly related to the problem of managing value.

Software is immaterial. We cannot touch it; the only way to handle it is to try understanding it, that is, knowing it. Lean management focuses on an effective management of the knowledge associated to the production.

Lean management in software focuses on methods that promote the creation of a shared understanding of the know-how, know-where, know-who, know-what, know-when, and know-why within the company [42]. The collection, storage, organization, processing, and distribution of this understanding throughout the software organization are essential to deliver value.

An approach that collects data about what the single developer does and relates it to his performance is the Personal Software Process (PSP) [30].

Using the PSP, a developer keeps track of what he produces (looking at the size of the source code), how much time he needs to produce it (measuring the required time), the quality of the produced code (the number of defects found in the product), and his estimation precision (comparing planned and actual completion dates).

The Goal Question Metric paradigm [8] is a methodology to specify measurement models; it provides a mechanism to define a measurement strategy, and it defines a measurement goal and the means to achieve this goal, i.e., which data that has to be collected. The GQM$^+$Strategies approach is an extension of the Goal Question Metric paradigm which "provides mechanisms for explicitly linking software measurement goals, to higher-level goals for the software organization, and further to goals and strategies at the level of the entire business [9]." The Goal Question Metric paradigm will be presented in Chap. 7, the GQM$^+$Strategies approach in Chap. 8.

Particularly since software is invisible, measurement is needed for understanding, control, and improvement of software. Measurement helps us to quantify things we observe in the real world and is useful in three ways [25]:

- Measurement helps to understand what is happening during the various activities in which programmers are involved; they make aspects of process and product more visible, giving a better understanding of relationships among activities and the entities they affect.
- Measurement allows to control what is happening on our projects. The collected data are used to predict what is likely to happen and to make changes to processes and products that help us to meet our goals.
- Measurement helps to understand how to improve the processes and products. For instance, we may adopt a new software development technique, based on measures of its impact on the software quality.

The GQM approach will be described in detail in Chap. 7 since it plays a crucial role in the context of the creation of a Lean software development process:

- it contributes creating organizational standards: it documents how the fulfillment of the stated objectives is measured;
- it creates visibility about the degree of fulfillment of the stated objectives;
- it contributes to the "study" step within Plan-Do-Study-Act providing data about the performed process; and
- it contributes to create visual control mechanisms within software engineering (e.g., to show that a certain class has a cyclomatic complexity that is over a given threshold or above average)

The Experience Factory [8], described in detail in Chap. 8, makes use of the Goal Question Metric paradigm to set up an adaptation of the Plan-Do-Study-Act paradigm that is tailored to software engineering. It combines the GQM paradigm with the Plan-Do-Study-Act philosophy to optimize processes based upon models of the business and the experience about the relationship between process characteristics and product characteristics [5].

The Balanced Scorecard approach [33] is an approach to structure key performance indicators within an organization in form of scorecards and to focus on a set of scorecards that cover all parts of the organization to obtain a "balanced" picture: "Think of the balanced scorecard as the dials and indicators in an airplane cockpit. For the complex task of navigating and flying an airplane, pilots need detailed information about many aspects of the flight. They need information on fuel, air speed, altitude, bearing, destination, and other indicators that summarize the current and predicted environment. Reliance on one instrument can be fatal. Similarly, the complexity of managing an organization today requires that managers be able to view performance in several areas simultaneously [33]."

Kaplan and Norton propose four scorecards (see Fig. 6.5) but point out that they should be extended or new scorecards should be added if this gives a more complete picture of the organization:

- **financial**: assesses the organization from the shareholders' viewpoint: shareholder value, liquidity, revenue growth, productivity, resource utilization, efficiency, etc.;
- **customer**: assesses the organization from the customers' viewpoint: perceived quality of the product, reputation of the organization, customer loyalty, etc.;
- **innovation and learning**: assesses the capability of the organization to innovate and continuously improve its processes looking at the available employee assets, information systems capabilities, and organizational infrastructure; and
- **business process effectiveness**: assesses the maturity of the business processes in efficiently and effectively providing the planned output.

The Balanced Scorecard approach implies causal links between the scorecards: e.g., it is assumed that the financial scorecard is the most important; all other scorecards contribute to the fulfillment of the financial goals. The Balanced Scorecard paradigm defines the set of measurements that have to be collected to assess the single scorecards and in this way defines the strategy of an organization in terms of goals and measurements.

Both the Balanced Scorecard and the GQM approach aim to create visibility within the organization providing feedback through the measurement of key performance indicators. Unlike the GQM approach, the Balanced Scorecard approach links the measurement goals to the organizational strategy. The GQM approach does not provide explicit support for integrating its software measurement model with elements of the larger organization, such as higher-level business goals, strategies, and assumptions, nor does it provide explicit support for dealing with goal dependencies [9].

The misalignment of software development goals with the organizational goals can have the following consequences [12]:

- the strategy is now known or used in the development of project goals and measurements;
- the data collection at the project level does not reflect organizational goals;
- financial indicators tend to drive corporate decision making;
- measurement is done mechanically with no clear purpose; or
- measurement is done in isolation of other projects.

Different approaches have been proposed to overcome this gap, e.g., [12] propose to integrate the Balanced Scorecard with GQM, or "GQM$^+$Strategies" [9], which is an approach that extends the GQM paradigm providing a methodology to link high-level business goals to software measurement data.

3.4 Improvement

Constant improvement is also related to the problem of maximizing value: it is essential to constantly verify if current working methods are still valid or if new technologies, new methods, past experiences, etc. can be used to improve.

The previously mentioned approaches to collect knowledge like the GQM approach and the GQM$^+$Strategies approach contribute to the improvement perspective as they provide the necessary input to improve.

The development of the Shewhart cycle [46] represents a milestone in the idea of constant improvement through standardization, which is based on the use of feedback from previous iterations. The Shewhart cycle was later refined by Edwards Deming as a systematic approach for problem solving within Total Quality Management [24], i.e., quality management that is applied to all areas of the organization.

The cycle consists of four steps: plan the activities to perform and their expected outcome; execute the plan (do); study the outcome and compare it with the expected outcome, i.e., understand how and why the realized result differs from the expected one; and confirm the plan or adjust it (act). The expected result of applying the Plan-Do-Study-Act paradigm is controlled processes, i.e., processes that—within certain limits—produce predictable results. Predictable means that it is possible to state—at least approximately—the probability that the observed phenomenon will fall within the given limits [46].

The "Guide to the Software Engineering Body of Knowledge" [1] mentions Plan, Do, Check, and Act (a synonym for Plan-Do-Study-Act) as an instrument to meet quality objectives within software engineering, based on the assumption that the quality of a product is directly linked to the quality of the process used to create it [22, 23, 32].

Six Sigma, invented at Motorola 1987 [39], uses an approach, which is derived from the PDSA-cycle, the DMAIC-cycle:

- Define (identify the problem),
- Measure (measure key attributes of the problem),
- Analyze (identify root causes),
- Improve (develop ideas to remove root causes), and
- Control (establish standard measures to maintain performance).

Also the Quality Improvement Paradigm [6], presented in Chap. 8, is based on the PDSA-cycle and is used within the Experience Factory.

A process improvement approach that is based on the works of Shewhart and Deming is the "Capability Maturity Model Integrated" (CMMI) [18] and its predecessor "Capability Maturity Model" (CMM) [40]. While organizations have several dimensions to improve the business of organizations (to improve the skills, training, and motivation of people, to improve the tools and the equipment, and to improve procedures and methods defining the relationship of tasks), the authors of CMMI put their focus on the third possibility that "the quality of a system or product is highly influenced by the quality of the process used to develop and maintain it [18]."

The CMMI is a family of reference models covering the development and maintenance activities applied to both products and services. In other words, CMMI does not provide a process, but it describes the characteristics that processes should contain (recommending best practices, proven to be effective through experience) to create processes that effectively achieve its goals. The CMMI for development foresees 22 process areas. A "process area" is a grouping of related best practices in an area, which when implemented collectively satisfy a set of goals considered important for making significant improvement in that area [18]. In CMMI, depending on which process areas are implemented, a process can achieve a maturity level from 1 to 5. The maturity levels are:

1. Initial,
2. Managed,
3. Defined,
4. Quantitatively Managed, and
5. Optimizing.

The recommended process areas for each maturity level are shown in Table 3.3.

Table 3.3 Recommended process areas for the CMMI for development

Practice	Maturity	Objective
Configuration management	2	Define work products, track and control changes to them, and maintain their integrity
Measurement and analysis	2	Define and set up measurement and analysis activities, provide measurement results
Project monitoring and control	2	Monitor the project against the plan and manage corrective actions
Project planning	2	Establish estimates, develop a project plan, and obtain commitment to the plan
Process and product quality assurance	2	Verify and communicate the process and product quality
Requirements management	2	Ensure alignment between project work and requirements
Supplier agreement management	2	Establish and satisfy supplier agreements
Decision analysis and resolution	3	Ensure that all possible decisions are evaluated
Integrated project management	3	Coordinate the defined processes with all stakeholders
Organizational process definition	3	Define how processes should be executed, tailored, and assessed
Organizational process focus	3	Determine, plan, and implement process improvements
Organizational training	3	Determine the training needs, deliver training, and assess the training effectiveness
Product integration	3	Ensure that the final product, assembled from different components, conforms to the requirements and deliver it
Requirements development	3	Elicit customer requirements, develop product requirements, and analyze and validate requirements
Risk management	3	Identify, analyze, and mitigate risks
Technical solution	3	Design a product that implements the identified requirements
Validation	3	Assure that the output meets the needs of the client
Verification	3	Assure that the output was constructed according to the requirements
Organizational process performance	4	Establish performance baselines and models
Quantitative project management	4	Monitor the performance quantitatively, analyze the root causes of problems
Causal analysis and resolution	5	Determine and address causes of selected outcomes
Organizational performance management	5	Select and deploy innovations

The CMMI approach is different from the approaches described so far. It prescribes **what** should be present to achieve a certain maturity certification, but it does not describe **how** this can be achieved. It is based on best practices that the researchers at the Carnegie Mellon University identified in industry. Deming frequently said: "You cannot inspect quality into a product [23]," meaning that quality comes from improvement of the process [34]. According to this view, the process has to be modified so that quality is part of the process itself.

So far we looked at larger proposals, methodologies, i.e., at "systems of methods and principles [21]" that were proposed for specific problems and contexts. We want to give two examples that also single practices coming from Lean can be used in one's software development process.

In the next two sections we give two examples of commonly used software development practices, which adopt a "push" approach and analyze a possible "pull" alternative: first (see Sect. 3.5), the practice of beginning the development collecting requirements and pushing them into to the development process, and second (see Sect. 3.6), the practice of developing code "bottom-up."

3.5 "Push" vs. "Pull" in Software Engineering: "Requirements-First" Development

The "push" approach is present whenever activities are triggered as soon as other activities finish their work, obtaining their output as input. The risk involved with the "push" approach is that the product or service, once it is "pushed" to the next activity, does not correspond to what is required for further processing.

To begin the software development process, collecting requirements is practiced in nearly all software development approaches. Some approaches try to collect all requirements upfront, and some iteratively collect a part of the requirements and implement them into an intermediate solution until all requirements are implemented; still, both approaches "push" requirements to the development process.

The definition of requirements converts business objectives into requirements that take into consideration the technical possibilities and their costs. This step requires considering the benefits and the costs of a technological solution in solving a business problem. Usually the originators of requirements lack this knowledge, which causes a large number of change requests during and after development [43]. This causes costs related to the following situations:

- the requirements do/do not optimally contribute to fulfill the business objectives;
- requirements change because old requirements have shown not to contribute to the fulfillment of the business objectives;

- the implementation of the requirements costs more than an alternative implementation that fulfills the business objectives in the same way;
- requirements that would contribute to the business objectives and that would not be costly to implement are not requested.

A possible "pull" approach can be achieved through an approach in which goals are collaboratively identified before setting the requirements and used to "pull" requirements and their priorities from the stakeholders [43]. Knowing the goals, technicians can clarify issues earlier, offer ideas on what could be done to achieve these goals, inform the business side on the different costs of possible alternatives to allow prioritization of goals, and generate the technical requirements to fulfill the overall goals.

The result is a development approach in which requirements are not "refined down to an implementation," i.e., taken as the starting point to develop an implementation that represents those requirements, but where the business objectives are mapped to the capabilities of the technical platform to "equally consider and adjust business goals and technical aspects to come to an optimal solution corresponding to the current situation [43] (see Fig. 3.1)."

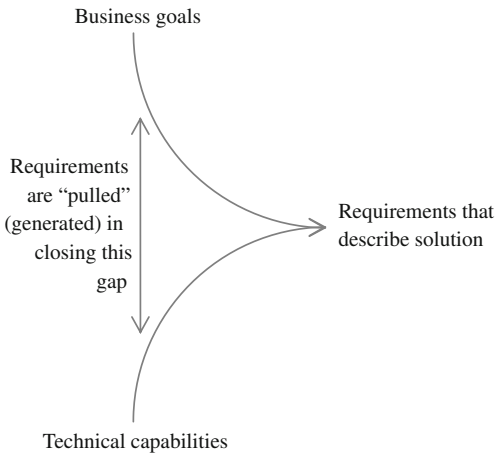


Fig. 3.1 Convergence of technical capabilities and business goals towards a solution [43]

Another similar approach is proposed by Briggs and Grünbacher [17] which involves all success-critical stakeholders in a collaborative requirements negotiation process to define win conditions (i.e., descriptions of business goals), issues (e.g., conflicts, technical difficulties, etc.), options (possibilities to solve the issues), and agreements (agreed solutions to overcome issues). The goal is to obtain a "concisely worded, non-redundant, unambiguous list of win conditions [17]."

3.6 "Push" vs. "Pull" in Software Engineering: "Bottom-Up" Development

In "bottom-up" development, the development begins with building blocks that will be used to create those parts that the end customer requires, similar to the process of building a house: the foundation is built first. When developing top-down, "the designer begins by determining what overall functions will be performed by the software system, in what order, and under what conditions. He then proceeds to develop a working top-level computer program, containing all the logic controlling the sequencing between functions, but inserting dummy programs or "stubs" for the functions. The succeeding steps consist of fleshing out the stubs into a lower-level sequence of control logic, computation, and sub-functions, each of which is again represented by a stub [45]."

Developers intuitively embark into the "bottom-up" paradigm since it is less complex to design, implement, reuse, and test the single building blocks and then continue to construct parts of the code using these building blocks instead of the opposite approach.

The building blocks represent unfinished parts put on inventory producing costs because of risks, e.g, often, different assumptions have been made during the development of the individual components, which can lead to a high-level control and data structure that are simply "kludged up," which makes it difficult, i.e., costly to maintain such an architecture if requirements change [45]. Moreover, a bottom-up development approach makes it difficult to understand if and how much value is provided by the single development activities.

Introducing a "pull" approach can be achieved adopting a top-down approach as described above and introducing "pull" mechanisms such as test-driven development [11]: test cases that are developed before the development define the goals of the upcoming development step. These test cases are motivated by the relative business objectives and "pull" the minimum amount of code that is necessary to fulfill them. In other words, they describe the needed capability and "pull" the minimum amount of code that is necessary to fulfill them.

The activities performed during test-driven development are as follows:

1. The programmer writes a test case that tests the desired functionality as if it were already present in the code.
2. The initial test case fails, but it describes the expected outcome of the new capability.
3. The next step is to make the test pass implementing the necessary code.
4. Refactor and pick the next requirement (go back to step 1).

Incremental additions can result in a confusing design and inhomogeneous or difficult to read code parts. For this reason, the last step consists of refactoring, i.e., the improvement of the design under preservation of the existing functionality [26].

3.7 Summary

This chapter presented three perspectives under which we analyze Lean concepts: value, knowledge, and improvement. We used the approaches presented in Table 3.1 to show that the values behind Lean already influenced a number of other concepts. We repeat the table below (Table 3.4), adding the perspectives to which, we think, the single approaches contribute the most.[1]

Table 3.4 Approaches that focus on similar values as Lean, together with the perspective to which, we think, the single approach contributes the most

Year	Proposal	Value	Knowledge	Improvement
1970s	Rapid application development (RAD) [35]	×		
1985	Quality improvement paradigm [6]			×
1986	Scrum (product development) [49]	×		
1986	Spiral software development [13, 15]	×		
1987	Six sigma [39]			×
1987	Goal question metric approach [8]		×	×
1989	Experience factory [7]		×	×
1991	Capability maturity model (CMM) [40]			×
1992	Balanced scorecard [33]		×	
1995	Dynamic systems development method (DSDM) [48]	×		
1999	Extreme programming (XP) [10]	×		
1999	Feature driven development [19]	×		
2000	Adaptive software development [28]	×		
2000	Personal software process [30]		×	×
2001	Agile manifesto [11]	×		
2002	Capability maturity model integration (CMMI) [18]			×
2004	Scrum (software development) [44]	×		
2004	Crystal clear [20]	×		
2007	GQM$^+$strategies [9]		×	×
2009	DevOps [31]	×		
2011	Disciplined agile delivery [4]	×		

[1] In fact, for almost all the approaches, one could say that they contribute to all three perspectives, because the three perspectives influence each other. For example, a better understanding of the production processes helps to increase the delivered value, constant improvement also improves the ability to collect and reuse knowledge, and so on.

We classified Rapid Application Development, spiral software development, the Agile Manifesto, and proposals that are based on the Agile Manifesto as mainly contributing to the understanding and maximization of the delivered value. They also have elements to increase knowledge and improvement, but we think this is not their primary goal.

We classified instruments like the Balanced Scorecard, the Experience Factory, GQM, GQM$^+$Strategies, and the PSP to the knowledge category, i.e., as methods that focus on the creation of a shared understanding of the know-how, know-where, know-who, know-what, know-when, and know-why within the company.

Finally, we classified methods that help to instill a culture of constant improvement such as the Quality Improvement Paradigm, Six Sigma, the Capability Maturity Model, and the Capability Maturity Model Integration, as mainly supporting improvement.

The two examples of existing approaches that translate Lean concepts into software engineering show that this translation can occur in different ways. In the first example, "pull" was used to organize the requirements engineering process and in the second example, to devise a development strategy. In the following chapters we develop a methodology to support such translation efforts.

Problems

3.1. Tag each process area of the CMMI (see Sect. 3.4) for development as:

- **value**, if its primary goal is to identify what has value and what has not;
- **knowledge**, if its primary goal is to increase the understanding of what happened, what is happening, and what will happen; and
- **improvement**, if its primary goal is to improve the status quo.

3.2. Compare Define-Measure-Analyze-Improve-Control with Plan-Do-Study-Act: link each step of the Define-Measure-Analyze-Improve-Control-cycle to one or more steps of the Plan-Do-Study-Act-cycle.

References

1. Abran, A., Moore, J.W., Bourque, P., Dupuis, R. (eds.): Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, Los Alamitos (2004)
2. Ambler, S.W.: Disciplined agile delivery and collaborative devops. Cutter IT J. **24**(12), 18–23 (2011)
3. Ambler, S.W., Lines, M.: Disciplined agile delivery: an introduction. IBM Software Design and Development, Thought Leadership White Paper. Online: http://public.dhe.ibm.com/common/ssi/ecm/en/raw14261usen/RAW14261USEN.PDF (2011). Accessed 3 May 2014

4. Ambler, S.W., Lines, M.: Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise. IBM Press Series. IBM Press, Upper Saddle River (2012)
5. Basili, V.R.: The experience factory and its relationship to other improvement paradigms. In: Sommerville, I., Paul, M. (eds.) Proceedings of the European Software Engineering Conference (ESEC). Lecture Notes in Computer Science, vol. 717. Springer, Berlin (1993)
6. Basili, V.R., Rombach, H.D.: Tailoring the software process to project goals and environments. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society Press, Monterey (1987)
7. Basili, V.R., Caldiera, G., McGarry, F., Pajerski, R., Page, G., Waligora, S.: The software engineering laboratory: an operational software experience factory. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, Melbourne (1992)
8. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1. Wiley, New York (1994)
9. Basili, V.R., Heidrich, J., Lindvall, M., Munch, J., Regardie, M., Trendowicz, A.: Gqm$^+$Strategies — aligning business strategies with software measurement. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Computer Society, Madrid (2007)
10. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley, Reading (1999)
11. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for agile software development. Online: http://www.agilemanifesto.org (2001). Accessed 4 Dec 2013
12. Becker, S.A., Bostelman, M.L.: Aligning strategic and project measurement systems. IEEE Softw. **16**(3), 46–51 (1999)
13. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Comput. **21**(5), 61–72 (1988)
14. Boehm, B.W.: Software risk management: principles and practices. IEEE Softw. **8**(1), 32–41 (1991)
15. Boehm, B.W.: Spiral development: experience, principles, and refinements. Technical Report CMU/SEI-2000-SR-008, Carnegie Mellon University, Software Engineering Institute, Pittsburgh (2000)
16. Boehm, B.W., Papaccio, P.N.: Understanding and controlling software costs. IEEE Trans. Softw. Eng. **14**(10), 32–68 (1988)
17. Briggs, R.O., Grünbacher, P.: Easywinwin: managing complexity in requirements negotiation with gss. In: Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS), vol. 1. IEEE Computer Society, Big Island (2002)
18. Chrissis, M.B., Konrad, M., Shrum, S.: Cmmi: Guidelines for Process Integration and Product Improvement. The SEI Series in Software Engineering. Addison-Wesley, Boston (2003)
19. Coad, P., Luca, J.D., Lefebvre, E.: Java Modeling Color with UML: Enterprise Components and Process. Prentice Hall PTR, Upper Saddle River (1999)
20. Cockburn, A.: Crystal Clear a Human-Powered Methodology for Small Teams. Addison-Wesley, Boston (2004)
21. Collins: Collins English Dictionary — Complete & Unabridged, 10th edn. HarperCollins. Online: http://www.collinsdictionary.com (2009). Accessed 4 Dec 2013
22. Crosby, P.B.: Quality is Free: The Art of Making Quality Certain. New American Library, New York (1979)
23. Deming, W.E.: Out of the Crisis. Massachusetts Institute of Technology Centre for Advanced Engineering Study (MIT-CAES), Cambridge (1982)
24. Deming, W.E.: Quality, Productivity, and Competitive Position. Massachusetts Institute of Technology Centre for Advanced Engineering Study (MIT-CAES), Cambridge (1982)
25. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, Boston (1998)

26. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading (1999)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
28. Highsmith, J.A.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, New York (1999)
29. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional, Upper Saddle River (2010)
30. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Professional, Reading (1996)
31. Hüttermann, M.: DevOps for Developers. Apressus Series. Apress, New York (2012)
32. Juran, J.M.: Juran on Leadership for Quality: An Executive Handbook. Free Press, New York (1989)
33. Kaplan, R.S., Norton, D.: The balanced scorecard: measures that drive performance. Harv. Bus. Rev. **70**(1), 71–79 (1992)
34. Kasse, T.: Practical Insight into CMMI. Artech House Computing Library. Artech House, Boston (2008)
35. Martin, J.: Rapid Application Development. Macmillan Publishing, New York (1991)
36. Nyfjord, J., Kajko-Mattsson, M.: Commonalities in risk management and agile process models. In: International Conference on Software Engineering Advances (ICSEA). IEEE Computer Society, Cap Esterel (2007)
37. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)
38. Padayachee, K.: An interpretive study of software risk management perspectives. In: Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology (SAICSIT). South African Institute for Computer Scientists and Information Technologists, Port Elizabeth (2002)
39. Pande, P.S., Neuman, R.P., Cavanaugh, R.R.: The Six Sigma Way: How GE, Motorola, and Other Top Companies are Honing Their Performance. McGraw-Hill Education, New York (2000)
40. Paulk, M., Curtis, B., Chrissis, M.B., Averill, E.L., Bamberger, J., Kasse, T.C., Konrad, M.D., Perdue, J.R., Weber, C.V., Withey, J.V.: Capability maturity model for software. Technical Report CMU/SEI-91-TR-24, Carnegie Mellon University, Software Engineering Institute, Pittsburgh (1991)
41. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley Professional, Boston (2003)
42. Rus, I., Lindvall, M.: Knowledge management in software engineering. IEEE Softw. **19**(3), 26–38 (2002)
43. Schnabel, I., Pizka, M.: Goal-driven software development. In: Proceedings of the Annual IEEE/NASA Software Engineering Workshop (SEW). IEEE Computer Society, Columbia (2006)
44. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press, Redmond (2004)
45. Selby, R.W. (ed.): Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research. Practitioners Series. Wiley, Hoboken (2007)
46. Shewhart, W.A., Deming, W.E.: Statistical Method from the Viewpoint of Quality Control. Dover Books on Mathematics Series. Dover, New York (1939)
47. Software Engineering Standards Committee of the IEEE Computer Society: IEEE standard for software life cycle processes: risk management (2001). IEEE Std 1540-2001
48. Stapleton, J.: DSDM Dynamic Systems Development Method: The Method in Practice. Addison-Wesley, Reading (1997)
49. Takeuchi, H., Nonaka, I.: The new product development game. Harv. Bus. Rev. **64**(1), 137–146 (1986)

Chapter 4
Agile Methods

Entia non sunt multiplicanda praeter necessitatem.
(Entities must not be multiplied beyond necessity.)

William Ockhamo

Very tired but happy, Uli moved back toward his office. Yes, he made it.

He just recollected that he left his senior architects in the middle of the meeting and... it was 4 hours ago. Now it was about 6PM, so he would have not expected to find anyone in.

He was wrong. Everyone was there waiting for him. The four of them had already heard the bad news, but not yet the good one. They all thought that Uli had gone out to shout and to let his anger dissolve and they would have never imagined the reality.

When Uli entered his office, they were there. The atmosphere was surreal. Elp was drinking a beer, Sinon and Euril were discussing a new paper by Tom De Marco, where it looked like he was retracting all his previous beliefs, Perim was doing her email on a new Android phone. Uli started to talk, but he cold not go ahead. His guys were smart. They did not want to wait to be laid off, so actually all of them but Elp have already sent out their resumes and Euril even got an offer. Elp was relaxed, as he had already planned for about an year to move to the consulting industry. They informed Uli of the news and reassured him of their everlasting friendship and loyalty—they all thought he was a great boss and he had no liability for what happened. Altogether, the cut of the project was perceived with a sense of freedom.

So, perhaps the good news was not such a good news! But, anyway, he was proud of his guys. They were able to react very fast and very effectively.

"Look guys, I am not sure whether what I am going to tell you is going to affect your decisions..." he paused for a while "but XXX is ready to let us to go ahead on the project if we are capable of changing the way we work. Substantially, we need to provide him the required functionalities piecewise and frequently. I know that this is going to affect the way we work, but it could be a fun experimentation. Rather than dividing and scheduling the work based on technical choices, we need to focus on what he perceives as a valuable element. Also, he will have the right to come to see what we do at any time and also to quit the project at any time. He ensured me

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_4

that he will not interfere with the specific design choices that we will take, provided that we are able to supply him a constant flow of new and valuable functionalities. And worse, these proposals come from me—I elaborated them from some readings I made recently and from what I consider effective software engineering practices, and adapting them to our specific environment." Sinon smiled "this reflects also some of the ideas that I was discussing with Euril on the paper I showed you yesterday." "Yes, indeed. Tom De Marco made me reflecting on something that has been up in my mind for a long, very long time. It appears to me that when we develop software we do not focus enough on the real value we provide to the customer, the running functionalities. We have put together a sequences of methods and practices and now we are slave of such things. It often looks more important to follow such methods and practices rather than delivering the running system. Such practices make us comfortable, let us feeling as if we were doing the right thing. If the project fails, we are sorry, but we feel in a sense good, because we followed such practices—so adverse and unavoidable events made the project failing, not us. We become martyrs for the good (practices), not just people who failed. But the practices and the methods are to provide value to the customer and not the other way around.

And we are very skilled software engineers. We are here, we did this careers, we studied for so many years, and, after all, we were born to follow virtue and knowledge, to grow in competence, not to follow blindly the practices. So, I said—stop." This was one of the great speeches for which Uli was known. Sinon had tears in her eyes. Elp thought he could gain this extra experience before going on is own.

Euril was a bit more hesitant: "Well, but proceeding in this way has not necessarily produced better results than the old way. After all Tom De Marco refrains explicitly from proposing a specific method." "I do not want to stick to a specific method. If the point were to devise a method, well, I would stay with the ones we have and we know. But we do our work because we have values and principles to follow—because we are proud to be software engineers. So, what I propose, rather, is to define what are the core values of our work and then, based on such values, to determine how to apply such values to our daily work with our customers. Needless to say, the application will depend from customer to customer. Talking with XXX I thought that the best way to proceed was the one we discussed, but this does not prevent that in a different situation we could have a different application, actually, this is very likely, almost certain."

Perim looked convinced. Her eyes were shining over the translucent dark skin of her face. Well, she was beautiful and people looking at her were nearly always hypnotized. She started to talk softly: "I am with you." She did not intend to do so, but her statement reminded everyone the time when she dated Uli, so the folks were even more concentrated on her. "But there are two key issues to address. The first is that if we have such frequent releases and we focus on developing valuable items in such time frames, we risk to loose the big picture of the system we are

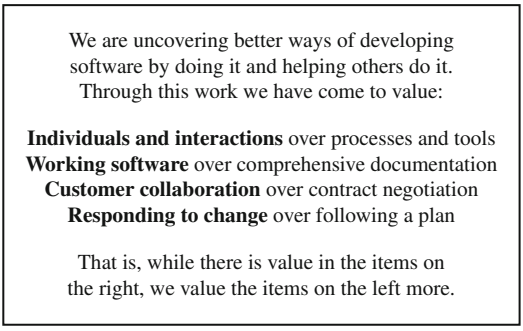
developing, and the overall architecture may go nuts. Then, currently we divide our tasks based on our skills, working on such short time frame requires to forget somewhat such skills, as there will be the time when we focus, say, on developing a database related feature and the time on a network issues. How can we handle such problems?" "Strike Perim!" Uli thought. He had these two issues also in his mind, but he did not want to make evident that still not everything was clear in his mind. "Perim, we need to develop some more situational awareness, so that we learn how to react faster to external stimulus. Also, we need to strengthen our horizontal skills; it is now evident that all of us should be not super-specialized in a specific branch of software engineering, but we ought to be broad." Uli had no idea of what "situational awareness" meant, but he heard such term from a consultant and thought that it was fuzzy enough to look like that he had an answer when he had not. Also, he was not very convinced of all these issues related to horizontal skills, actually, he was not even sure whether they applied also to software engineering or only to some of the usual socio-psico-philo-disciplines whose experts were paid big bucks to sell (what he thought was just) vapor.

"Uli, I am not sure what you mean but…" Perim replied. Uli felt a stroke on his stomach—was she going to uncover him? "but I think you are right. After all, we have nearly always spent long times in building large architectures, but then we have often ignored them and such time actually has only be a waste. I also think that we are all smart people. Probably, it is better that we acquire all a good knowledge of everything, and we can easily do it, so that we can learn better what the system is as a whole, rather than to have the overall view of the system dispersed in our individual understanding based on our very specific skill, so that everyone has a really deep picture of the system from his viewpoint, the viewpoint of the technology he masters, but there is none with a solid global view and, perhaps, the individual perspectives even do not match. So, if we go this way, I would even propose that we collectively share the responsibility for the overall system and for each piece of code, that we let everyone modify each part of the code she or he think it is suitable—after all we will (or should) all be competent in all the details of the system." "Fine, fine, fine" said Elp, "but to be on the safe side, let us have a good suite of tests pervading the whole code, so that if anyone makes a change that actually breaks the code, it will become immediately evident. We should aim at 90% coverage, at least for the parts not related to user interfaces. With such constraint, with the fact that none wants to look stupid making nonsense modifications ad with the usage of a good version control system, so that we can revert any unlikely wrong modification that may occur, I feel comfortable that we can proceed."

Uli felt now tired but relaxed. He still had to get the go from Athi and from J, but the team was with him. So he called Athi asking for a meeting with her and J the day after and then took everyone out for dinner at Calypsos, where they would have forgot the troubles in front of a good moussaka followed by a generous portion of baklava a la mode.

4.1 Introduction

It was the year 2001. A significant number of software engineers had already been trying to apply the concepts of Lean management to software engineering. A group of them decided to formalize their approach. They identified a set of values and principles and synthesized them in the "Manifesto for Agile Software Development" [5], also known as "Agile Manifesto" (see Fig. 4.1). We will now go into the details of the Agile Manifesto and of its Principles to unveil some of its facets. We also evidence how some such ideas were not conceived out of the blue—actually they represent the synthesis of years of research and development in this area.


We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Fig. 4.1 The Agile Manifesto [5]

The "Agile Manifesto" identifies two sets of values: the values lying on the left of the document and the values lying on the right of the document (see Table 4.1).

It then claims that the values of the right are important, but they consider even more important the values on the left. So they value individual and interactions more than processes and tools, etc. It is worth noticing the graphic metaphor they use: the values on the right are the values of the rational thinking, of the top-down, Fordist/Taylorist, waterfall-ish development; after all, they are on the right side! The values on the left sit on the side of the heart and the side of the emotions, of the interactions and of the collaborations, of what often can be lost despite being so

Table 4.1 Set of values identified by the Agile Manifesto [5]

Values on the left	Values on the right
Individual and interactions	Processes and tools
Working software	Comprehensive documentation
Customer collaboration	Contract negotiation
Responding to change	Following a plan

important, since we are in a knowledge-intensive field and we know how many defects come for misunderstanding requirements, people, code, documentations, etc.

Right after the "Agile Manifesto," the "Principles behind the Agile Manifesto" were formalized. The values are translated into twelve principles to guide software engineers in their everyday's work (see Fig. 4.2).

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances Agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Fig. 4.2 The 12 principles behind the Agile Manifesto [5]

The first two principles describe how an Agile software developer sees the world: customer satisfaction is the highest priority, and the best way to achieve it is to adapt to the evolving needs of the customer. In this context, "Agility" acknowledges the impossibility to predict all the desired system upfront and provides support for the natural evolution of requirements.

Moreover, while technical excellence is of paramount importance (Principle 9), it is not the ultimate end of the project, which is in turn customer satisfaction.

This is further emphasized by Principle 7, where we acknowledge that any measure that does not relate to the customer satisfaction has little sense. Only the working software that can be supplied to the customer can be the measure of how much we have progressed in the project. This rules out using as milestones our own definitions of how the project should be done, for instance, including our analysis

and design document, the amount of discussions we had internally, and so on. Note that it is not said that such things are useless; it is written that they cannot be an objective measure of how much we have progressed, because they reflect our perception of the project and not the value that we provide to the customer, that is, what matters at the end.

Principle 10 is further attached to this matter. Saying that we strive for simplicity means that we eliminate anything that does not carry a value and so, in turn, that we first focus on customer satisfaction, going back to the tenets of Lean Management.

The remaining principles describe how an Agile software developer chooses to achieve Agility. These practices are not all new; some of them can be linked to existing best practices; this evidences the evolutionary processes that lead the creation of Agile Methods, that actually are rooted in years of software engineering research and development, and that represent a comprehensive "fusion" of ideas.

Table 4.2 lists some concepts that seem to have inspired the principles of the manifesto.

Table 4.2 Ideas that can be related to principles in the Agile Manifesto

Year	Principle	Idea
1954	11	Management by objectives [16]
1968	5	Job enrichment [23]
1970s	3	Rapid application development [28]
1971	6	Communication improvement by collocation of workers [35]
1975	4	Involvement of stakeholders [26]
1978	7–12	Toyota production system [30]
1986	12	Plan-Do-Study-Act [15, 34]
1988	3	Prototyping and iterative development [9]

The previous chapter mentioned Rapid Application Prototyping and Spiral development which seem to have inspired Principle 3, i.e., to deliver working software frequently.

Lucas pointed out in the 1970s that "the major reason most information systems have failed is that we have ignored organizational behavior problems in the design and operation of computer-based information systems [26]." He models failure based on three classes of variables: user attitudes and perceptions, the use of systems, and user performance [6]. The need to involve users, see what they expect, how they use the system, how they perform, etc. throughout the project seems to have influenced Principle 4.

Job Enrichment (see below) seems to have influenced Principle 5.

Weinberg in "The psychology of computer programming" [35] already describes the communication improvement by collocation of workers; this might have influenced Principle 6.

The principles 7–11 can be linked to the ideas described in the previous chapter: the idea to focus on what produces value (Principle 7), to involve and respect workers (Principle 8), to focus on quality in the process (Principle 9), to focus on simplicity (Principle 10), to let the team assume responsibility (Principle 11), and to constantly improve (Principle 12). Also the approach of "Management by Objectives [16]" proposed by Drucker might have influenced Principle 11.

"Best practices" are a popular way in software engineering to package and transfer experience and knowledge. Table 4.3 shows examples of best practices; according to Fraser [18], the practices in the table below were combined to what is now known as "Agile Methods."

Table 4.3 Software best practices are not new [18]

Best practice	Year	Introduced by
Requirements	0s	(Exists since the beginning of time)
Pair programming	1950s	John Von Neumann (IBM)
Project planning	1960s	Mercury project (NASA)
Risk management	1960s	Mercury project (NASA)
Software architecture	1960s	Frederick P. Brooks, Edsger W. Dijkstra, David L. Parnas
Software reuse	1960s	Malcolm D. McIlroy (AT&T)
Test-driven design	1960s	Mercury project (NASA)
Coding standards	1970s	Brian W. Kernighan, P. J. Plauger
Collective ownership	1970s	Unix, open source
Continuous integration	1970s	IBM federal systems division
Data hiding and abstraction	1970s	David L. Parnas
Documentation	1970s	David L. Parnas
Incremental releases	1970s	Victor R. Basili, Albert J. Turner
On-site customer	1970s	Harlan D. Mills (IBM federal systems division)
Simple design	1970s	Victor R. Basili, Albert J. Turner
Software measurements	1970s	Tom Gilb, Maurice H. Halstead
Evolutionary design	1980s	Tom Gilb
Patterns	1980s	Tom DeMarco, Timothy Lister, Gang of Four
Peopleware, sustainable pace	1980s	Tom DeMarco, Timothy Lister
Use cases	1980s	Ivar H. Jacobson

(continued)

Table 4.3 (continued)

Best practice	Year	Introduced by
Software economics & estimation	1980s	Barry W. Boehm
Metaphor	1990s	Kent Beck, Martin Fowler, Howard G. Cunningham
Refactoring	1990s	William F. Opdyke, Martin Fowler
Retrospectives	1990s	Norman L. Kerth, Linda Rising

4.2 Keeping the Process Under Control

At the heart of a development process, there are the mechanisms by which the work of people is composed, synchronized, and synthesized together to produce goods, that is, how to organize, control, and direct the activities of the workforce to the ultimate success of the operations. We have already discussed how this was implemented in the original ideas of Taylor and Ford and how this changed in the early Lean Management approaches.

It appears therefore obvious that at the very heart of Agility, there is a different way of organizing the work of people.

A key aspect of such organization is the "control" of the work of people to ensure that they do what they are supposed to do. Often, controlling the work of an employee means that the employee has to follow a sequence of (often written) rules of behavior and that there is a supervisor that checks at given interval of time whether the employee has done what he was supposed to do. This is true in the Fordist/Taylorist approaches, and it is also true in the early Lean management approaches. Having a fixed schedule of work with the need of a written permission of the supervisor for any change, punching cards when entering and exiting the office, writing reports of each own activities to be approved by the supervisor, etc. are all mechanisms to control.

The fact is that there are several ways to implement the concept of "controlling the work," especially when dealing with motivated and skilled people. Michael L. Harris [22] proposed an effective way to classify different types of control and in which circumstances these types work best.

The selection of the most effective control mechanism depends on two factors:

1. the ability to measure the output, the "measurability," and
2. the ability to specify in details the steps required to accomplish a given task, the "specifiability."

So if we put these two factors in a two-dimensional diagram (see Fig. 4.3), we obtain four areas:

1. an area with high level of measurability and of specifiability;
2. an area where we have a high level of measurability of the output but little specifiability;
3. an area where we have a high level of specifiability but little measurability; and
4. an area where we can neither measure directly the output nor specify the steps of the work.

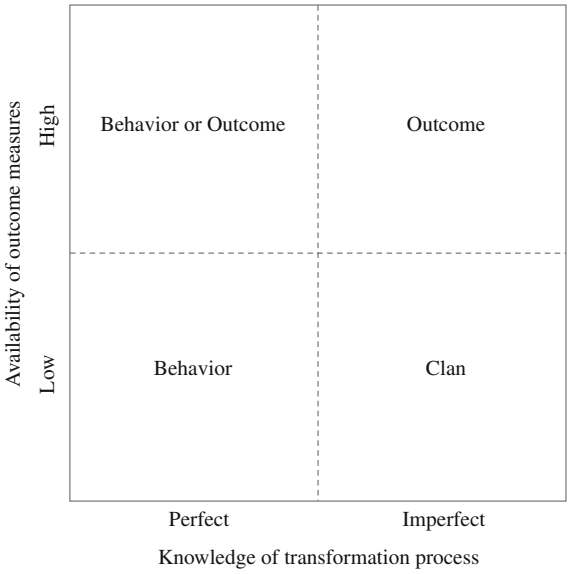


Fig. 4.3 Organizational use of control types [22]

When we are in the area with a high level of measurability and specifiability, we know both how to produce an asset and how to measure whether it has been produced correctly. A typical example of workers falling in this area is secretaries, who can be controlled, for instance, either by checking their output, e.g., the documents they have prepared, the arrangements they made, etc. Therefore, we can apply either a control of the outcome or a control of the steps.

When we are in the area with a high level of measurability of the output but little specifiability, we can control the process only by measuring the outcome. Cookers are a clear example of works that are hardly controllable but can be controlled simply by assessing the quality of the food that they prepare. Workers are evaluated based on a comparison of their output with an expected output.

Output control is in line with Management by Objectives [16]: a "process whereby the superior and subordinate managers of an organization jointly identify its common goals, define each individual's major areas of responsibility in terms of

the results expected of him, and use these measures as guides for operating the unit and assessing the contributions of each of its members [29]."

Altogether, outcome measures define how the optimal output should look like. If a task outcome cannot be described or if an individual's contribution to the outcome cannot be easily determined, then using outcome-based control will not be efficient and management by objectives becomes less effective.

Output control is related to Autonomation (see Chap. 2): autonomation requires a mechanism to detect problems, i.e., requires that the output is measurable and comparable to an objective.

Certain industries, among which is the software industry, produce goods with an evolutionary pattern (see, e.g., [1]), that is, the outcome emerges gradually together with its constraints: scope boundaries (given through the user stories) and ongoing feedback (continuous course corrections through the interaction with stakeholders). In this case we can use the term "emergent outcome," which gives the sense of how, for instance, Agile processes build gradually and piecewise systems towards a final outcome. Measuring the emergent outcome is not easy, and it is often accomplished by a direct intervention of the customer.

In the area with a high level of specifiability of the process but little measurability of the output, we can control the steps that are accomplished but hardly measure the output. But we trust that following the process does lead to the correct answer. Clerical workers are often localized here. They accomplish steps that, composed altogether, can produce successful results, but each step individually is hardly measurable (filling a form, running an interview, checking items against a to-do list, etc.). In this case we can have a behavioral control: workers are evaluated based on a comparison of their performance to a pre-specified behavior that is known to transform inputs to desired outcomes.

The area where we can neither measure directly the output nor specify the steps of the work characterizes most knowledge-intensive works. Consider the case of a medical doctor; the effectiveness of his work cannot be measured simply in terms of the adherence to a set of steps. Otherwise, all doctors would be equally good, while we know that there is a huge difference between a "regular" doctor and a "good" doctor, our Dr. House. Moreover, also the output is not easy to measure. On one side we do not want to wait for the patient to die to determine whether a task was performed correctly. On the other side, treatments often have long-term effects, so a comprehensive measurement would require years. This is exactly the same for software engineers: it is not possible to trivialize the work of building software systems in terms of a sequence of simple steps—otherwise there would not be the huge well-known difference in productivity between a "regular" programmer and a "good" programmer as reported by [10, 12–14].

Altogether, this calls for a different kind of control, the so-called clan control, in short, the control by the peers.

Clan control mechanisms are based more on the promotion of a shared set of values that result in a positive, efficient, and effective attitude towards the work rather than in monitoring the actual work product. Once this attitude alignment is achieved, it is expected that clan members self-regulate based on those common

values and their individual decisions are consistent with the interests of the organization.

The implementation of an effective clan control is not trivial. This can be achieved by selecting the correct individuals and by socializing with them to share the values and the objectives of the organization [22]. Then, when clan control emerges, rituals and ceremonies are used to promote common values among clan members.

In software engineering it is not possible to know in advance which transformation process will lead to the desired outcome; the outcome is often produced incrementally, and also the measurement of the output is often hard. Altogether, emergent output control and clan control appear the most effective means to govern the software development process. More about this topic will be discussed later.

4.3 Job Enrichment

A peculiar set of techniques can be very effective in the implementation of clan control. The early Lean Management has gone beyond the simplistic concept of division of labor. It recognized the advantages of motivated, responsible employees through the adoption of techniques that promote motivation through non-monetary remunerations to the workers, especially valuable for those who enjoyed the work they do. Such set of techniques is called "Job Enrichment." A summary of them is listed in Table 4.4.

Job Enrichment can be an effective mechanism to promote an effective process control, especially if an approach based on Clan Control is adopted. Therefore, it is important to review some of the principles of Job Enrichment, as presented by Herzberg [23].

Herzberg claims that it is of paramount importance to increase the accountability and responsibility of employees, so that they themselves become motivated and interested that their work has a high-quality outcome.

The "seven useful starting points for consideration" stated by Herzberg are shown in Table 4.4. These suggestions are based on Herzberg's studies that identified the factors creating motivation, and those creating dissatisfaction are different (i.e., the opposite of motivation is no motivation, and the opposite of dissatisfaction is no dissatisfaction), as shown in the figure below.

The identified factors (see Fig. 4.4) causing motivation, the "motivator factors," are "achievement, recognition for achievement, the work itself, responsibility, and growth or advancement." The factors that can cause dissatisfaction when disadvantageous for the employee are "company policy and administration, supervision, interpersonal relationships, working conditions, salary, status, and security." These factors—when favorable for the employee—which do not cause motivation but avoid dissatisfaction are called hygiene factors [23].

Table 4.4 Techniques of job enrichment [23]

Techniques	Motivators involved
Removing some controls while retaining accountability	Responsibility and personal achievement
Increasing the accountability of individuals for own work	Responsibility and recognition
Giving a person a complete natural unit of work (module, division, area, and so on)	Responsibility, achievement, and recognition
Granting additional authority to employees in their activity; job freedom	Responsibility, achievement, and recognition
Making periodic reports directly available to the workers themselves rather than to supervisors	Internal recognition
Introducing new and more difficult tasks not previously handled	Growth and learning
Assigning individuals specific or specialized tasks, enabling them to become experts	Responsibility, growth, and advancement

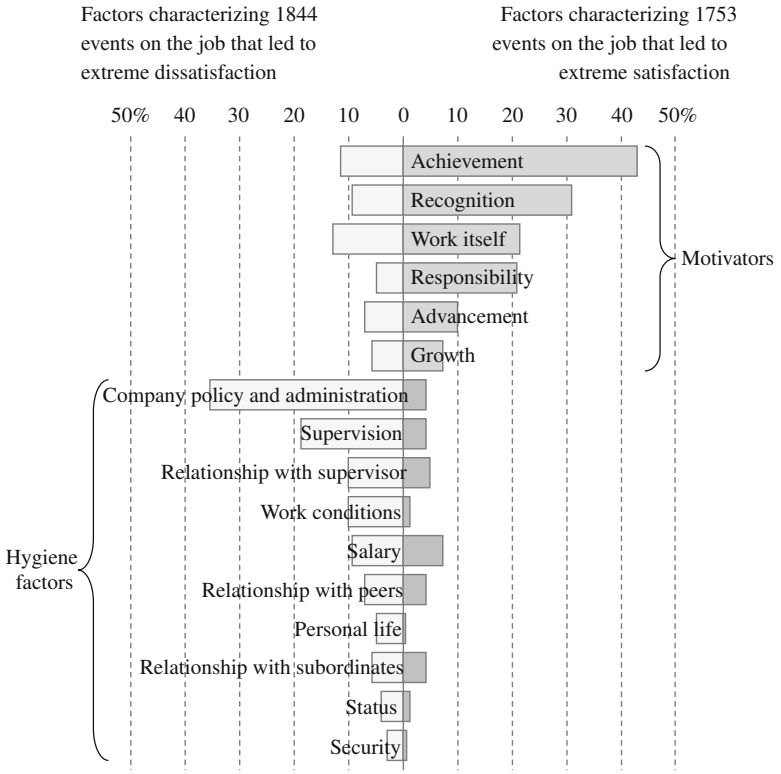


Fig. 4.4 Factors affecting job attitudes [23]

Now we understand Herzberg's recommendations:

- to remove some controls while retaining accountability reduces a hygiene factor (supervision) and increases motivator factors (responsibility and personal achievement);
- to increase the accountability of individuals for own work reduces hygiene factors (supervision, security; more accountability is connected to a higher risk to fail) and increases a motivator factor (responsibility and recognition);
- to give a person a complete natural unit of work reduces hygiene factors (company policy and administration, supervision) and increases motivator factors (responsibility, achievement, and recognition);
- to grant additional authority to employees in their activity reduces hygiene factors (company policies, supervision) and increases motivator factors (responsibility, achievement, and recognition);
- to make periodic reports directly available to the works themselves rather than to supervisors reduces a hygiene factor (supervision) and increases a motivator factor (internal recognition);
- to introduce new and more difficult tasks not previously handled reduces a hygiene factor (company policies and administration) and increases motivator factors (growth and learning); and
- to assign individuals specific or specialized tasks, enabling them to become experts, increases motivator factors (responsibility, growth, and advancement).

At this point of the book, it should be clear that producing software is not like producing cars: there is not a high availability of reliable and solid measures of the output and there is a lack of a sound knowledge of the transformation process. Software engineering cannot define rules to control effectively the transformation process; as mentioned, this calls for a Clan Control mechanism, which, in turn, requires a high motivation of the individual through techniques such as Job Enrichment.

4.4 Endogenous and Exogenous Control Mechanisms

Whether a control mechanism is clan control, process control, or output control, it should be exercised with two different kinds of mechanisms: endogenous mechanisms and exogenous mechanisms.

What we call "endogenous mechanisms" subsumes all methods that embed control mechanisms into the process. We gave already examples of endogenous control in the "Quality management" section of Chap. 2; the second connectors of Fig. 2.19 are made in a way that it is not possible to connect them wrongly—the tool box shown in Fig. 2.20 does not allow to put the scissors on a wrong place since they would not fit.

Autonomation is another example; it implements endogenous control indirectly—through its mechanism that verifies a certain property of the produced output and the alert mechanism, it enforces a given condition. Autonomation—by alerting and stopping the machine automatically in case of an error—implements endogenous control: it makes it impossible to produce something that does not fulfill the condition verified through autonomation. Autonomation allows to verify more complex conditions than to verify the correct size or location of objects (as in the examples of Figs. 2.19 and 2.20).

The opposite of endogenous control is exogenous control: control that is not embedded into the process but is coming from the actors involved in the process.

What is now endogenous control within a software methodology? We consider the control endogenous if the application of a practice forces also others to fulfill it. It is like in Fig. 2.20: once we all decide to put the scissors in the box shown in the figure, there is only one way to put it. An example is the shared code practice of Extreme Programming (see below): if we decide that everybody has the right to change any part of the code, all team members are forced to, e.g., follow some coding standards. If not, other team members will complain as soon as they want to change that code.

4.5 Synchronizing the Flow of Work of Multiple People

Coordinating work means to manage dependencies between activities. Malone and Crowston [27] consider different kinds of coordination, i.e., "managing dependencies between activities":

- shared resource,
- producer/consumer,
- simultaneity, or
- task/subtask.

A shared resource dependency is given when actors require access to the same limited resource. An example would a be "first come/first serve" approach or to assign time slots for each worker when accessing a resource.

In a producer/consumer dependency, one activity produces something that is used by another activity. Coordination mechanisms for this type of relationship are that the first activity is a prerequisite for the second or that a specific good has to be transported from one place to another place to be further processed.

We talk about simultaneity dependency when activities have to occur on the same time (or cannot occur at the same time). Scheduling a meeting is an example where all participants have to be available. Another example is to find a free time slot for a lecture at the university.

This problem is quite complex since several simultaneity constraints apply: students cannot attend more than one lecture at the same time, lecturers cannot give more than one lecture at the same time, rooms cannot be occupied by more than one class at a given time, all students of 1 year have to have time to attend (no other compulsory lecture at the same time), and so on.

A task/subtask relationship exists when a set of subtasks contribute to the achievement of a higher task or goal. The subtasks represent parts of the higher-level task that are decomposed according to some criteria, e.g., by function, by product, by customer, by geographical region, etc.

4.6 Extreme Programming (XP): A Paradigmatic Example of Agile Methods

Extreme Programming (XP) is a software development methodology conceived and refined by Kent Beck [3, 4] based on the principles of the Agile manifesto and that tries to implement effective software development using the ideas of Lean management.

The control is mostly exercised with Clan Control, using whenever possible endogenous mechanisms and with an extensive use of simultaneity constraints synchronization methods.

XP springs from a criticism of one of the axioms of traditional software engineering: "The cost of changes grows exponentially as the project progresses." Since the work of Boehm [7, 8], there has been a common belief that the cost of fixing a defect or to change a feature increases exponentially with the time of fixing it, that is, if the cost is 1 during requirement elicitation, it becomes 10 during analysis, 100 during design, 1,000 during coding, and so on.

If this curve is true, then it is essential, very essential to anticipate all decisions and to collect all requirements initially and try to plan as detailed as possible all the steps to perform. In other words, this curve demands a very Fordist model. However, we have already discussed the inefficiencies of the Fordist models and the impossibility to do the planning. So, in his groundbreaking work of 1999, Beck questions whether this curve is really the only way to go or whether it is possible to devise a different way to develop software, where the curve remains flat throughout development (see Fig. 4.5).

If we are able to have defined a development model that supports a flat curve, then the development process becomes "Agile," that is, it is possible to apply modifications also later in the development phases and non-Fordist development models become applicable.

XP constantly is looking to adopt practices, methods, or technologies that contribute to obtain a flatter cost of change curve.

XP wants to establish a way to develop software where the cost of a change is constant throughout the development. This means a change during the early phases of development costs as much as a change made during maintenance. Such a curve means that the development team is able to accept modifications also late in the development project at a cost that does not depend on the point in time when the modification is requested.

This goal implies a radical change on the culture of software development.

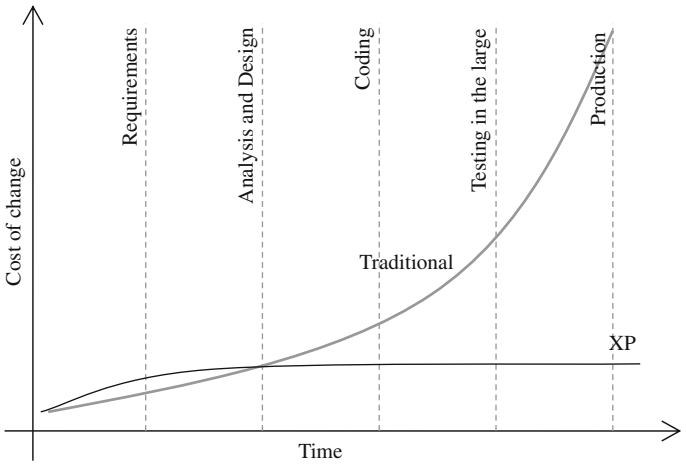


Fig. 4.5 Traditional and Agile cost of change curve [4]

4.7 The Building Blocks of XP

As mentioned, XP is grounded on Clan Control. Therefore, XP is organized in terms of values, principles, and practices (see Fig. 4.6), three levels of abstraction that explain the way of thinking promoted by XP. The values represent the "constitution" of the clan doing XP. The constitution is the base for all laws; still it remains at a very high level. The constitution is then translated into laws.

The principles are the "laws" of the clan. The principles are then implemented with specific operational rules of the clan, the practices. There are five values in the constitution of XP: communication, simplicity, feedback, courage, and respect.

Communication is seen as the most important ingredient for effective cooperation within a clan, which is enhanced through **respect**: an attitude to accept different viewpoints, constructively evaluate every possibility, and care about the contributions of each person of the clan.

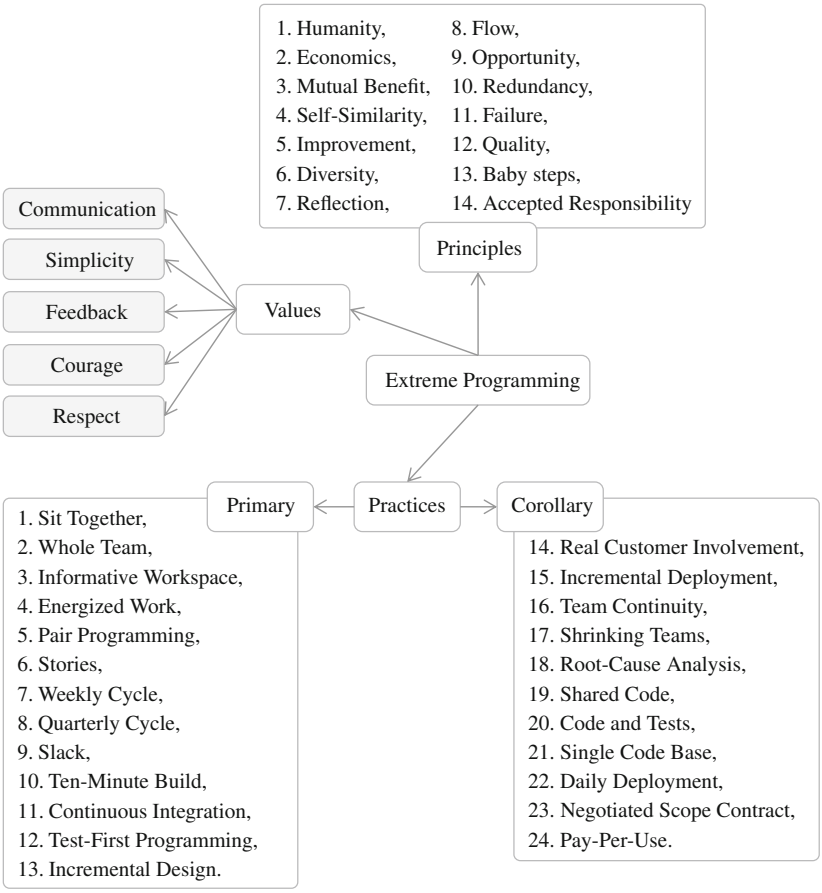


Fig. 4.6 The building blocks of extreme programming

Simplicity emphasizes that a team should try to be efficient: achieve the maximum business value with the minimum effort. During development, the "simplest thing that could possibly work" [4] should be used.

The clan gathers **feedback** on everything it does. It includes discussing alternative ideas with clan members, evaluating the performance of a system through a test, confronting customers with intermediate versions of a solution, and so on. Finally, **courage** is necessary to be ready "to speak truths, pleasant or unpleasant," to "discard failing solutions," and to "seek real, concrete answers" [4] within the clan.

The values are interrelated, for example, simplicity diminishes the need for communication and the need for feedback: requirements that are not needed do not need to be discussed, constructing a test case to analyze the behavior of the system in a tricky situation can anticipate future problems, etc (see Fig. 4.7).

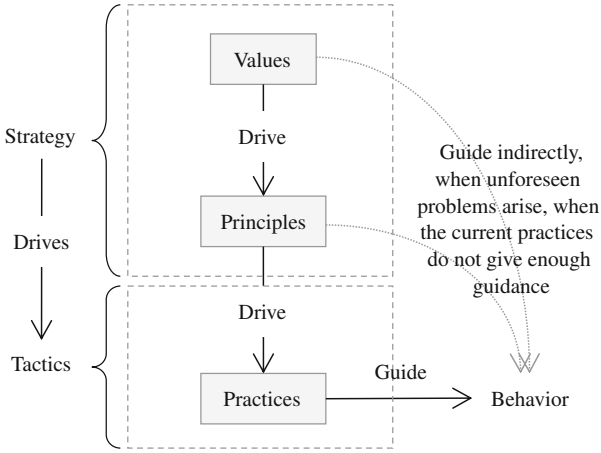


Fig. 4.7 Values, principles, and practices in XP

The law of every XP clan is based on the following 14 principles [4]. We list them here below, bolding their original name.

Clan members shall:

- Avoid the "costs and disruption of high turnover and missed opportunities for creative action" [4] (**humanity**).
- Remember that the goal is to create business value, meeting business goals, and serving business needs. It is not the technically brilliant solution (**economics**).
- Seek "win-win" situations whenever possible (**mutual benefit**).
- Develop the capability to extract patterns from existing solutions to apply them in new contexts, also at different scales (**self-similarity**).
- "Do the best you can today, striving for the awareness and understanding necessary to do better tomorrow" [4] (**improvement**).
- Try to bring together a team with a "variety of skills, attitudes, and perspectives to see problems and pitfalls, to think of multiple ways to solve problems, and to implement the solutions" [4] (**diversity**).
- Reflect regularly about past successes and failures to institutionalize the lessons learned from these experiences (**reflection**).
- Deliver a constant flow of valuable software to anticipate feedback and to split complex tasks (such as integration) into many, easier-to-accomplish, iterations (**flow**).
- See problems as opportunities for learning and improvement (**opportunity**).
- Solve critical, difficult problems from different angles, using different approaches to overcome them (**redundancy**).
- Be prepared to risk to fail in order to learn from it (**failure**).
- Strive for high quality (**quality**); it increases predictability, productivity, and effectiveness, but also acts as motivational factor, since people need to do work they are proud of.

- Improve in small steps (**baby steps**). Together with the concept of flow above, we see how change is done in XP in Fig. 4.8: often and in small steps.
- Accept responsibility (**accepted responsibility**). To be responsible requires also to obtain the authority to be able to fulfill the duties (this follows the principle of congruence, which states that task, competence, and responsibility have to be congruent to motivate and to be able to delegate [31]).

The principles shown above have their roots in the inspiring ideas that we have previously described. As mentioned, these inspiring ideas were developed before and contributed to define XP. In Fig. 4.8 we linked the principles to the six previously developed ideas shown in Table 4.2.

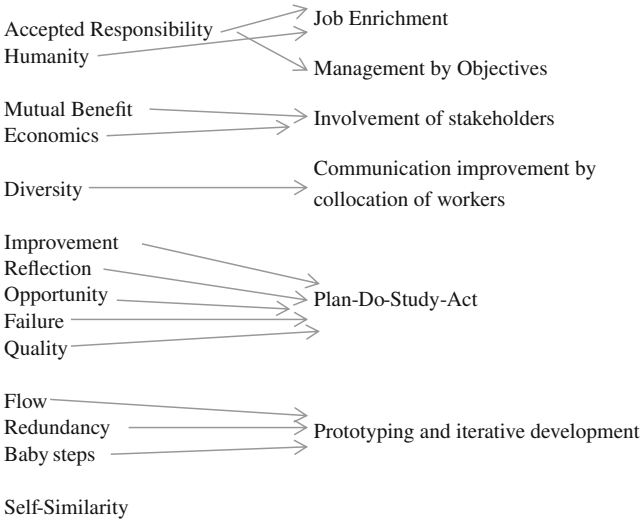


Fig. 4.8 Links between practices and inspiring ideas

4.8 The XP Practices

Practices describe the practical rules that the clan has to follow in its everyday life. They are divided into primary and corollary practices to recommend adopters to start with the primary ones and only, when they have an XP process in place, experiment with the corollary ones.

Table 4.5 shows the practices as well as the classification recommended by [4].[1]

In the following sections (one per classification), we describe each practice in detail.

[1] We categorize unclassified practices as "generic."

Table 4.5 XP practices with their classification [4]

Practice	Business	Integration	Planning	Programming	Team
Primary practices					
Continuous integration		×			
Energized work					
Incremental design				×	
Informative workspace					
Pair programming					
Quarterly cycle			×		
Sit together					
Slack			×		
Stories			×		
10-min build		×			
Test-first programming				×	
Weekly cycle			×		
Whole team					
Corollary practices					
Code and tests				×	
Daily deployment	×				
Incremental deployment					
Negotiated scope contract	×				
Pay-per-use	×				
Real customer involvement					×
Root-cause analysis					×
Shared code				×	
Shrinking teams					×
Single code base				×	
Team continuity					×

4.8.1 Business Practices

The business practices recommend the adoption of practices that maintain Agility and remove unnecessary activities in the negotiation and distribution process.

Daily deployment advises to put new software into production every night, providing customers with value every day.

A **negotiated scope contract** aims to reduce the risk of writing software that does not correspond to what the customer values. This practice recommends to foresee contracts in which the time, costs, and quality are fixed but with a negotiation of the precise scope of the system. Keeping the scope open allows customers to continuously select which parts should be delivered next and allows developers to fulfill the time, costs, and quality requirements.

Finally, **pay-per-use** recommends to charge the customer for every time the system is used and not to let him buy the release. Pay-per-use allows to obtain faster feedback from the market about the acceptance of the product than with pay-per-release.

4.8.2 Integration Practices

The integration practices deal with the step of merging a modification with the rest of the code and to test (i.e., integration testing) the newly combined code.

The practice of **continuous integration** advises to integrate and test changes "after no more than a couple of hours [4]." The motivation is that the complexity to integrate a modification with a code base that is itself changing because of other team members applying modifications is rising over time.

The **10-min build** practice supports continuous integration asking for a build process and a run of all tests that does not take longer than 10 min. A build process that takes too long discourages developers from using it often to obtain feedback.

4.8.3 Planning Practices

Planning involves all activities that help to decide who, how, when is going to do what in the upcoming iterations. XP foresees the practices of **weekly cycle** and **quarterly cycle** to plan activities, where the activities are described in the form of **stories** and grouped by **themes**.

Requirements are written in the form of **stories**: short descriptions of customer visible functionalities together with an estimation given by a programmer of the effort to implement this story (see Fig. 4.9). Beck and Andres claim that according to their experience, "every attempt [. . .] to computerize stories has failed to provide a fraction of the value of having real cards on a real wall" (see the practice "informative workspace").

Weekly cycle advises to plan work weekly. Planning includes reviewing the progress of the previous week (also analyzing discrepancies between the planned and effectively done amount of work), having the customers pick the stories to implement this week, and letting team members sign up for the tasks and estimate them (e.g., using "story points"). Weekly planning allows to estimate the velocity of the team, i.e., how many story points per week can be done on average, and to use this estimation for future planning (see Fig. 4.10).

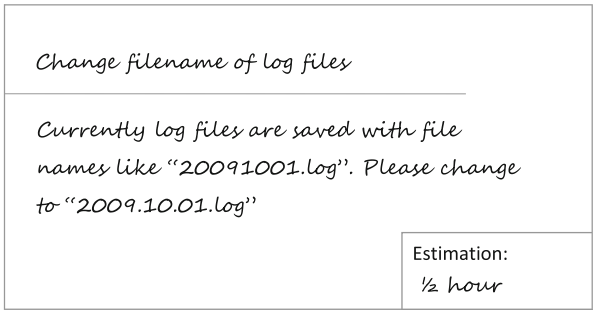


Fig. 4.9 An example of a user story

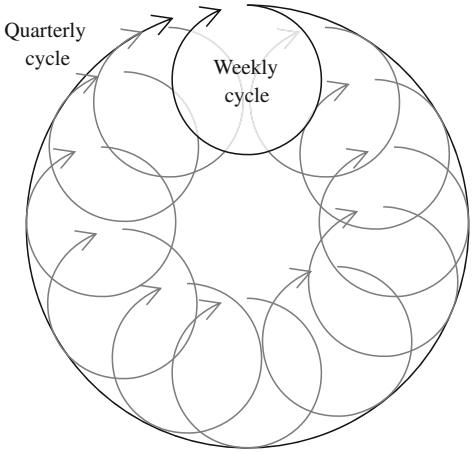


Fig. 4.10 Illustration of a weekly and quarterly planning cycle

Every quarter—so the practice **quarterly cycle**—the team reviews the past work from a broader perspective, not focusing on optimizing team performance internally, but considering the entire organization, identifying external bottlenecks, etc. Following the business priorities, the theme/s (sets of stories addressing a similar topic) are picked by the customers and planned for the upcoming quarter (see Fig. 4.10).

The last practice of this section—**slack**—advises to reserve time for secondary activities that can be used if the team gets behind or—if not needed—to allow programmers to dedicate part of their time to work on a project of their choosing [19].

4.8.4 *Programming Practices*

Programming—to focus on working software—is one of the core values of XP.

The practice of **code and test** reflects this: only the code and the tests should be maintained. Other artifacts such as class diagrams and design documents should be generated from the code and tests. Only artifacts that contribute to "what the system does today and what the team can make the system do tomorrow" are valuable [4].

Incremental design is based on the assumption that it is possible to keep the cost of changing the software at a similar level as in Fig. 4.5. Under this assumption, designing as late as possible reduces the risk of being wrong.

In incremental design "refactoring" plays a crucial role. Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [17]."

Shared code states that everyone in the team that finds that something is wrong with the system has the right to fix any part of the code, at any time. This practice requires that everybody acts with responsibility.

Single code base states that there should be only one code stream. A temporary branch should not be used more than a few hours since it will increase the complexity of integration as time moves on.

Finally, **test-first programming** advises to write a failing automated test before coding. Test-first programming affects programming on multiple levels: writing the test before the code itself helps to focus on the goal of the current programming task, encourages to write code that is easy to test and acts as a form of documentation since it states the intentions of the tested piece of code.

The win-win negotiation model

Frequently software development evolves into one of the situations shown in Table 4.6 [11], in which there is always one party that loses (the developer writes the solution, the customer buys it, and the user uses it).

Table 4.6 Winners (👍) and losers (👎) for the different solutions

Proposed solution	Developer	Customer	User
Cheap, Sloppy Product ("buyer knows best")	👍	👍	👎
Lots of bells and whistles ("cost-plus")	👍	👎	👍
Driving too hard a bargain ("best and final offers")	👎	👍	👍

According to this model, those that "win" on the short term lose on the long term since the losing party will not trust them anymore and maybe choose somebody else for future projects.

(continued)

Barry Boehm proposes a win-win negotiation model shown in Fig. 4.11 [2, 11, 20]: the win-win system is considered to be in equilibrium if all stakeholders agree on the win conditions (i.e., stakeholder requirements) and there are no outstanding issues.

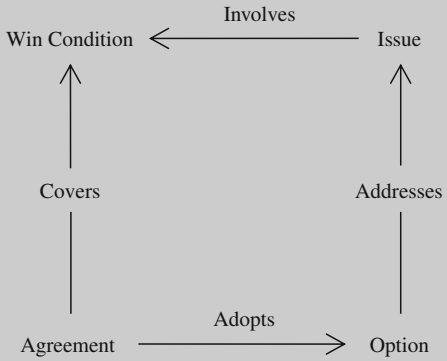


Fig. 4.11 The win-win negotiation model [33]

During the negotiation process stakeholders enter their win conditions, identify conflicts, i.e., issues, devise options to propose solutions to an issue, and agree on options to solve issues (the option linked to the issue, that is adopted through the agreement, helps to obtain agreement among all stakeholders on a win condition). This process continues until all stakeholders have agreed on all win conditions (taking into consideration the adopted options) or until the stakeholders decide that no consensus can be found and the project should be canceled [33].

4.8.5 Team Practices

The practices dealing with team management describe how teams should be formed, work, and reformed into new teams.

Real customer involvement means to consider all stakeholders as part of the team and involve them in the quarterly and weekly planning. As no customer at all makes it impossible to deliver value, having a "proxy customer," i.e., a representative of the real customer, can be as bad since this can end up in a "Chinese whispers" game.

Root-cause analysis advises how a team should work: it should definitively eliminate a defect together with its cause every time a defect is found. This is accomplished writing an automated test that demonstrates the defect and fixing the system afterwards so that if the problem occurs again, the test shows this.

Moreover, this practice advises to—as in the Toyota Production System to ask five times "why" (see Chap. 2)—identify and address the root cause of the problem, find the responsible person, and develop a solution.

Shrinking teams recommend to keep the workload of teams that improve their efficiency over time constant but gradually remove people from this team to form other teams.

Team continuity advises to keep effective teams together. This does not mean that teams are static, i.e., that never ever people change since the rotation of team members helps to consistently spread knowledge and experience.

4.8.6 Uncategorized, Generic Practices

The here mentioned practices support the other practices.

Energized work means to work only as many hours as one can be productive and only as many hours as one can sustain. This practice warns from overworking resulting in spending time inefficiently and finally removing value from the project instead of adding value to it.

Incremental deployment advises to deploy large systems not in one time (also called "big bang deployment") but gradually, shifting the workload gradually from the old to the new system.

Informative workspace recommends to arrange the workspace in a way that an "interested observer is able to walk into the team space and get a general idea of how the project is going in fifteen seconds" [4]. One possibility is to use a task board as, e.g., in Fig. 4.12: using a task board and stickers representing tasks the team can use to keep track of the ongoing work as well as keep the work plan visible to everybody that steps in within seconds (see for example Fig. 4.13).

The pair programming practice states: "write all production programs with two people sitting at one machine" [4]. The aim of this practice is to keep each other on task, inspire each other's creativity, motivate each other, and hold each other accountable to the team's practices [24, 25].

Sit together advises to have the entire team in one open space to encourage communication, having small private spaces nearby where team members can work privately if needed.

While **sit together** aims at ensuring the proximity of people to ensure communication, the practice of **whole team** aims at ensuring the proximity of skills, i.e., it means to include on the team people with all the skills and perspectives necessary for the project to succeed (obtaining a cross-functional team). This "proximity of skills" has the goal to supply the project with an easy access to the resources necessary to succeed (Fig. 4.13).

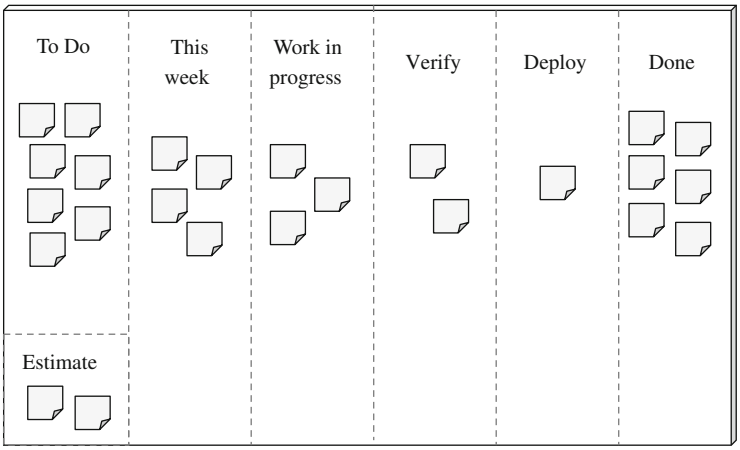


Fig. 4.12 Example of a task board

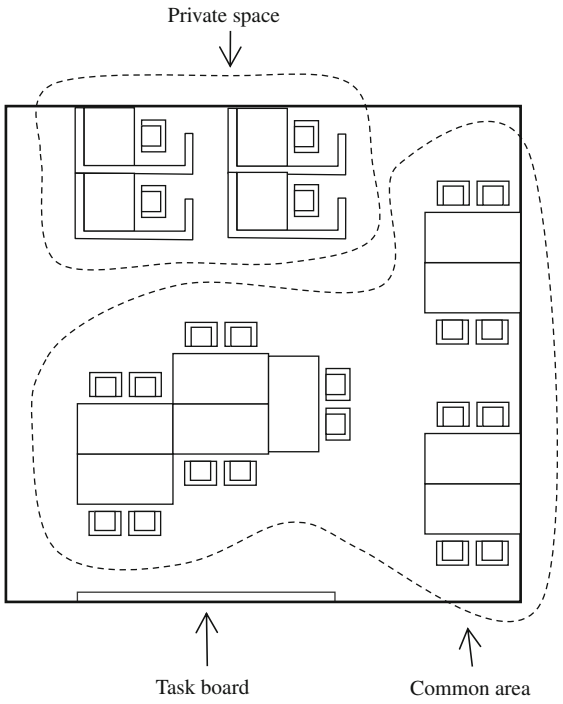


Fig. 4.13 Example of an Agile work space

4.9 Control and Coordination Mechanisms

At the beginning of this chapter, we analyzed that it is possible to consider three forms of control: outcome, behavior, and clan control. Harris [21] analyzed Agile processes[2] whether they adopt outcome, behavior, or clan controls in respect to the following taxonomy:

- **Outcome controls**: measure performance against a priori specifications.
  - Emergent outcome controls: manage outcomes in an evolutionary fashion.
    - Scope boundaries: constrain creativity to insure focus on key areas.
    - Ongoing feedback: provides corrective feedback as development occurs.
- **Behavior controls**: measure adherence to behaviors that transform inputs to outputs.
- **Clan Controls**:
  - Team: Team members provide task feedback to support coordination and communication.
  - Attitude: Individuals make decisions based on attitudes and values that are congruent with organization's attitudes and values.

We can analyze Extreme Programming from two different point of views: first, as the whole methodology, asking what makes the team fulfill the different tasks, and, second, analyzing the practices in detail asking how these practices use forms of control to achieve their goal.

On the level of the whole methodology, Extreme Programming is driven by Clan Control mechanisms, i.e., the objective that team members make decisions based on attitudes and values that are congruent with organization's attitudes and values. Subsequently, Agile Methods deliberately choose outcome controls as a driver for their work: "Working software is the primary measure of progress" (see Principle 7 of the Agile Manifesto).

Extreme Programming uses endogenous control mechanisms to create a high visibility of every practice: everybody notices if somebody is programming alone (instead of using Pair Programming); if one would not continuously integrate, the entire development would stop and if one does too much overtime and is not able to begin his work the next day together with the rest of the team, this is also noticed.

Harris [21] points out that on the practice level, different forms of control are used to ensure that a given practice is successfully carried out (see Table 4.7).

[2]Harris speaks about "flexible processes" instead of Agile ones to include also development organizations that are not using formal Agile methods but that use controlled, flexible processes that manage towards emergent outcomes instead of managing towards a predefined specification.

Table 4.7 Forms of control in the primary extreme programming practices [21]

Practice	Outcome Controls: Traditional outcome	Outcome Controls: Emergent Outcome: Scope Boundaries	Outcome Controls: Emergent Outcome: Ongoing Feedback	Behavior Controls	Clan Controls: Team	Clan Controls: Attitudes	Note
Continuous integration		×					Always be ready to demo latest product.
					×		Fix defects as they occur.
						×	Must maintain synchronization with others. No surprises.
Energized work						×	When we are at work, we will do our best.
Incremental design		×					Each iteration focuses on only a few things.
			×				Each iteration is ready to use or demonstrate to the market.
Informative workspace	×						Visible results.
			×				Outcomes are observable.
				×			Daily Progress is visible.
					×		Team can readily see each other's progress.
Pair programming			×				New ideas are tested with partner.
				×			Work together.
					×		Activities transparent to team members.
Quarterly Cycle		×					Place business constraints as well as market constraints.
			×				Review with management. Guard against feature creep.
Sit together			×				Progress observable by team members.
					×		Team members accessible for advice.
Slack				×			Meeting weekly cycles more important than features.
Stories		×					Broad statements of intent focus efforts.

(continued)

Table 4.7 (continued)

Practice	Outcome Controls: Traditional outcome	Outcome Controls: Emergent Outcome: Scope Boundaries	Outcome Controls: Emergent Outcome: Ongoing Feedback	Behavior Controls	Clan Controls: Team	Clan Controls: Attitudes	Note
10-min build			×	×	×		Make it easy to demonstrate. Don't break overall system. Code must work well with others.
Test-first programming		×					Develop a detailed goal for each feature.
Weekly cycle			×	×	×		Limit amount of change that can occur in each iteration. Market feedback every 1–3 weeks. Work to short deadlines. One-day slips result in miss of weekly target.
Whole team			×		×		Feedback including customer representative. Team is self-sufficient and unified. No mavericks.

Analyzing the coordination mechanisms of Extreme Programming, as a consequence of choosing "Working software is the primary measure of progress," the dominating coordinating mechanism is "Simultaneity," i.e., the need of working together on a common outcome—source code—results in coordination problems that have to be dealt with ensuring that all can work together avoiding possible conflicts.

We analyzed which coordination mechanisms are faced on the practice level. Table 4.8 shows our results for the main and corollary practices of Extreme Programming. We indicate with a single dot when a practice faces a specific coordination mechanism and has to deal with it, and we indicate with a circle when a practice helps to alleviate a specific constraint since it makes it easier to deal with it. Some examples:

- Continuous integration faces shared resources coordination mechanism: it is forced to deal with a code base that is used also by others.

- Sit together faces simultaneity constraints because it requires that all have time to work nearby.
- Energized work alleviates the simultaneity constraint since it assumes that workers that avoid working overtime are able to come on time in the morning to work together.

Table 4.8 Coordination mechanisms for the practices of extreme programming

Practice	Shared resources	Producer/Consumer	Simultaneity	Task/subtask
Primary practices				
Continuous integration	+			
Energized work			–	
Incremental design	+	+	+	+
Informative workspace	–	–	–	–
Pair programming	+		+	
Quarterly cycle	+	+	+	+
Sit together			+	
Slack		+	+	+
Stories		+	+	+
10-min build		+	+	+
Test-first programming		+		
Weekly cycle	+	+	+	+
Whole team			+	
Corollary practices				
Code and tests			–	
Daily deployment	+	+	+	+
Incremental deployment	+	+	+	+
Negotiated scope contract			–	
Pay-per-use		–		
Real customer involvement			+	
Root-cause analysis				+
Shared code	+			
Shrinking teams			–	
Single code base	+			
Team continuity			+	

4.10 Summary

Agility wants to help an organization to build teams and software that are able to cope with the (changing) requirements of the client.

There are different interpretations of how Agility can be achieved, e.g., Extreme Programming defines 5 values, 14 principles, and 13 primary and 11 secondary practices to guide developers, managers, and clients towards the values identified in the Agile manifesto.

This chapter shows that Extreme Programming is not just coding. It is a complex framework that requires a disciplined approach to software engineering. The questions are now: Is this enough? Is this framework complete? The next chapter deals with open issues in Agile software development and how these issues can be approached.

Problems

4.1. Looking at Fig. 4.5, can you give examples of software development practices or activities that lower the traditional cost of change curve?

4.2. Imagine you had to introduce Extreme Programming in a software development team that follows a waterfall process [32], such as in Fig. 4.14.

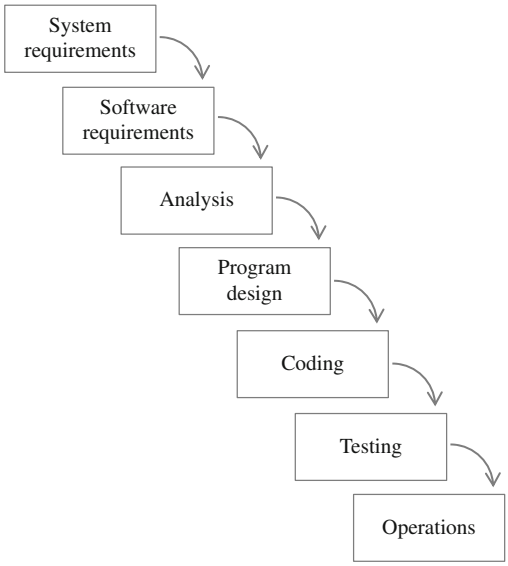


Fig. 4.14 The waterfall software development process [32]

The waterfall process is based on the traditional cost of change curve and assumes that it is possible to collect all requirements upfront. Starting from the requirements, it is conceived to help the development team to engineer the envisioned software in a controlled way.

Which problems do you foresee? How will the clients react (that until now are used to work with a team that used the waterfall process)? How would you address them?

References

1. Bain, S.L.: Emergent Design: The Evolutionary Nature of Professional Software Development. Net Objectives Lean-Agile Series. Addison-Wesley Professional, Upper Saddle River (2008)
2. Barry, W., Boehm, R.R.: Theory-W software project management principles and examples. IEEE Trans. Softw. Eng. **15**(7), 902–916 (1989)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading (1999)
4. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Boston (2004)
5. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for agile software development. Online: http://www.agilemanifesto.org (2001). Accessed 4 Dec 2013
6. Beynon-Davies, P.: Information systems 'failure' and 'risk' assessment: the case of the London ambulance service computer-aided dispatch system. In: Doukidis, G.I., Galliers, R.D., Jelassi, T., Krcmar, H., Land, F. (eds.) Proceedings of the European Conference on Information Systems, Athens (1995)
7. Boehm, B.W.: Software engineering. IEEE Trans. Comput. **25**(12), 1226–1241 (1976)
8. Boehm, B.W.: Software Engineering Economics. Prentice-Hall Advances in Computing Science and Technology Series. Prentice-Hall, Englewood Cliffs (1981)
9. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Comput. **21**(5), 61–72 (1988)
10. Boehm, B.W., Papaccio, P.N.: Understanding and controlling software costs. IEEE Trans. Softw. Eng. **14**(10), 1462–1477 (1988)
11. Boehm, B.W., Bose, P., Horowitz, E., Lee, M.J.: Software requirements as negotiated win conditions. In: Proceedings of the International Conference on Requirements Engineering (ICRE). IEEE, Colorado Springs (1994)
12. Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D., Steece, B.: Software Cost Estimation with Cocomo II. Prentice Hall International, Upper Saddle River (2000)
13. Curtis, B.: Substantiating programmer variability. Proc. IEEE **69**(7), 846 (1981)
14. DeMarco, T., Lister, T.: Programmer performance and the effects of the workplace. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society Press, London (1985)
15. Deming, W.E.: Out of the Crisis. Massachusetts Institute of Technology Centre for Advanced Engineering Study (MIT-CAES), Cambridge (1982)
16. Drucker, P.F.: The Practice of Management. Harper & Row, New York (1954)
17. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading (1999)

18. Fraser, S.: Software "best" practices: agile deconstructed. In: Bomarius, F., Oivo, M., Jaring, P., Abrahamsson, P. (eds.) Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES). Lecture Notes in Business Information Processing, vol. 32. Springer, Berlin/Heidelberg/Oulu (2009)
19. Gargiulo, S.: Route to the top: how employee freedom delivers better business. CNN. Online: http://edition.cnn.com/2011/09/19/business/gargiulo-google-workplace-empowerment/index.html (2011). Accessed 4 Dec 2013
20. Grünbacher, P., Boehm, B.W.: Easywinwin: a groupware-supported methodology for requirements negotiation. In: Proceedings of the European Software Engineering Conference (ESEC) Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, Vienna (2001)
21. Harris, M.L.: Using emergent outcome controls to manage dynamic software development. Ph.D. thesis, University of South Florida (2006)
22. Harris, M.L., Hevner, A.R., Collins, R.W.: Controls in flexible software development. In: Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS), Kauai, vol. 9 (2006)
23. Herzberg, F.: One more time: How do you motivate employees? Harv. Bus. Rev. **46**(1), 53–62 (1968)
24. Janes, A., Russo, B., Succi, G.: Use of pair programming for experience exchange in a distributed internship project. In: Fraser, S., Williams, L. (eds.) Proceedings of the International Workshop on Pair Programming Explored. ACM, Seattle (2002)
25. Janes, A., Russo, B., Zuliani, P., Succi, G.: An empirical analysis on the discontinuous use of pair programming. In: Marchesi, M., Succi, G. (eds.) Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering (XP). Lecture Notes in Computer Science, vol. 2675. Springer, Genova (2003)
26. Lucas, H.C.: Why Information Systems Fail. Columbia University Press, New York (1975)
27. Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Comput. Surv. **26**(1), 87–119 (1994)
28. Martin, J.: Rapid Application Development. Macmillan Publishing, New York (1991)
29. Odiorne, G.S.: Management by Objectives: A System of Managerial Leadership. Pitman Publishing Corporation, New York (1965)
30. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)
31. Reiß, M.: Das kongruenzprinzip der organisation. Wirtschaftswissenschaftliches Studium **11**, 75–78 (1982)
32. Royce, W.W.: Managing the development of large software systems: concepts and techniques. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society Press, Monterey (1987). Reprinted from Proceedings, IEEE WESCON, August 1970, pp. 1–9, originally published by TRW
33. Selby, R.W. (ed.): Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research. Practitioners Series. Wiley, Hoboken (2007)
34. Shewhart, W.A., Deming, W.E.: Statistical Method from the Viewpoint of Quality Control. Dover Books on Mathematics Series. Dover, New York (1939)
35. Weinberg, G.M.: The Psychology of Computer Programming. Computer Science Series. Van Nostrand Reinhold, New York (1971)

Chapter 5
Issues in Agile Methods

Quel sol che pria d'amor mi scaldò 'l petto,
di bella verità m'avea scoverto,
***provando e riprovando**, il dolce aspetto;*

e io, per confessar corretto e certo
me stesso, tanto quanto si convenne
leva' il capo a proferer più erto;

(That Sun, which erst with love my bosom warmed,
Of beauteous truth had unto me discovered,
*By **proving and reproving**, the sweet aspect.*

And, that I might confess myself convinced
And confident, so far as was befitting,
I lifted more erect my head to speak.)

Dante Alighieri, Divina Commedia, Paradiso, Canto 3, 1–6

Eventually, the morning came. Uli could not sleep much. He had an intense day and he managed to recover a desperate situation just buy making promises whose impacts was almost completely unknown. And then, at the end, Perim stroke him with the key question. And such question was still in his head. Situational awareness, what the hell! The big picture, this was the point. "How can I be driven by the request of the customers and still keep a cohesive view of the system I am developing?" Uli kept asking himself.

The meeting with J and Athi was at 8:15. Uli did a fast breakfast at his favorite coffee shop and he was ready for the discussion. Uli arrived at the office of J at 8:10... better to be a bit earlier, he thought. Athi was already there—she was actively discussing with J the new ideas of Uli. "How are you Uli?" asked J, and, without waiting for a reply "let us go to the Olympus conference room. What you propose us is something novel, we ought to discuss it with a few other folks." Something important was going on, thought Uli. Olympus was the largest and best equipped conference room of FSS. The four most senior technical managers were there, Ari, Helios, Eor, and Mes. In a few seconds also Hera, the CIO, arrived.

"Gentlemen, J started, I wanted all of you here to get your advice, started J. Here we have a man who managed to rescue a contract that was almost gone, well almost gone for his and his group. Rescue a contract, but still we do not know whether it will be profitable for us to continue in such contract, as his rescue cost several additional

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_5

commitments to the customer who are not customary in our tradition. What we are discussing is not whether he is competent and capable to continue the contract. We all pay a lot of respect to Uli. Rather, we are discussing whether it is profitable for us to continue the contract with the approach proposed by Uli. So, please, stay focused. Uli, please go ahead." Uli started presenting the situation, what happened with XXX, the morale, the break down, his idea. "It is not just a contract—he concluded—it is the opportunity to assess if it is possible to develop software differently and whether this different approach is suitable for us. And by saying this I do not mean that we should risk the money of our shareholders undertaking dangerous endeavors, I mean that there is a great opportunity to improve our operation and to generate value for our company and we have to assess whether this opportunity is for us. Please note that the fact that XXX may stop the contract at any time is a major pros also for us as we will have the same right. So, if you will end up feeling that the contract is not any more profitable for us, well, you will let us know and in less than two weeks it will be over with no litigation, pain etc."

Mes jumped up as if he had wings in his feet: "I like it Uli, yes I like it. We need to assess if and how we can do better; our competition is assessing a new fad—they call them Agile something, and I thing we should give a try to this as well." Helios nodded his head. He and Uli never went along particularly well. He thought Uli was not enough "structured" to handle complex projects. He was a nice guy, with good technical skills, but then, if there was the need of someone capable of playing hard balls... well it was a different story. "Look Uli, I like your attempt to keep your team united and not to lay off few of your fellows. This is indeed nice of you. However, we need to make sure that we maintain sound engineering practices—someone else could have to deal with the code you are producing now and would need a good overall view of the system, including a well structured architecture, sound design documents, commented code etc. And, especially, a nice overall perspective of what we are producing." Uli felt again a stroke on his stomach. He could not play the card of "situational awareness" with Uli. Helios was an old fashioned software engineer, with a very solid mathematical background and a deep, instinctual repulsion for "all those folks making a hell of a lot of money out of useless blah blah blah." So, he preferred to pass and hope to get some help from some other questions. But when Ari raised his hand he felt even more uncomfortable.

"It is nice to hear that we have wasted our time in useless activities such as analysis documents, design documents, and, particularly, you know, activities that never get recognized for doing good things for us (he intentionally mentioned Paul Simon here as J was a fan of him), such as maintenance, training, scouting for new ideas and so on." Well Ari was not a nice teammate. He wanted to be in the position of J but never managed. And he loved fights. He continued, "but what Uli proposes does not create knowledge in the company, produces only some folklore in the team. How can we ensure a long-lasting life of a project if we do not store the experience we had in written format, such as in analysis and design documents?"

Eol coughed and got the attention from all the people in the room "Come on Ari, these documents are Write Once, Read Never documents. They become immediately obsolete, as they are not updated and it is pointless to say that we should be stricter

with our developers. If we have a harsh deadline in a few days, we work 24/7 on such deadline and we forget about everything else. And once the deadline is passed we are too tired to fix what we did outside the rule. Overall, I think that the idea of Uli deserves some serious consideration. The facts that we always did something should not mean that we keep doing it forever. Also, it is embarrassing that we never experiment novel approaches to software development. We need to gather some quantified elements that can guide us in how we will develop software in the next decade. Still the problem of sharing and transmitting knowledge is indeed a key problem. Not that we are doing it right now, but I would like to see it better managed."

Uli stood, joined his hands "Ladies, gentlemen" he started "we have to acknowledge that our profession is still mostly a territory of unknown. We do not have yet books with consolidated practices and everlasting principles. Only, via a constant process of proof and refutation we can find the truth, the truth we love. So, why are we scared to do it?"

J nodded. Uli thought that he did it... but Hera looked at J and then at Uli. She did not like him, or, perhaps, she liked him too much—but she was the wife of J and the CIO of the company! So she started by saying "Uli, you are a very respectable manager, still I am not convinced by your words. Knowledge is a key issue. But also for me there is an issue of value of what we do and the associated cost, which we need to be able to quantify, and with your approach this becomes impossible. Ari continued "and without measuring our value how can we work on well known issues of constant process improvement, on getting CMM level 2, on being certified by quality assessors?" Athi was ready to start talking, but then she realized that Hera had been not too negative, and Hera would have been only bitterer if she tried to defend Uli—Hera knew that she had a crush on Uli...

J nodded again. This time Uli was not happy. "Well, guys, here we raised several issues that deserve a more comprehensive understanding. I am particularly concerned on how we manage our most relevant asset, the knowledge, and then how can we track the value we produce to our shareholders and how can we constantly improve to keep our leadership on our competition. Still, I am intrigued by the words of Uli, so the best way would be to ask Uli to prepare for 11AM a few slides in which he explains how to plan to handle these three things: experience, value, and improvement. The meeting is adjourned at 11AM in the East Meeting Room. It is 9, so we have 5' for a good espresso."

5.1 Introduction or "the Hype of Agile"

Innovations often go through a phase of overenthusiasm or "hype" and—when the expectations turn out to be unrealistic—subsequent disappointment. New technologies offer new opportunities, so the hope to overcome unsettled problems with the existing solutions is high and generates excessively optimistic expectations.

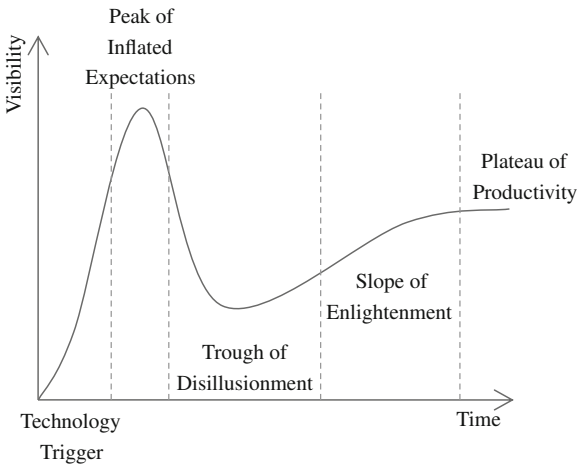


Fig. 5.1 Gartner's Innovation Hype Cycle

Gartner's Innovation Hype Cycle is based on this idea (see Fig. 5.1): it visualizes the phases that an innovation goes through before becoming productive. Those innovations that manage to go through Gartner's Hype Cycle (not all do; see [20] for some examples) pass the following phases [11]:

1. the Technology Trigger;
2. the Peak of Inflated Expectations;
3. the Trough of Disillusionment;
4. the Slope of Enlightenment; and
5. the Plateau of Productivity.

The Hype Cycle begins with the technology trigger—an innovation that generates significant attention in field. As it happened, for example, with the e-Business hype of 1999, resulting in the burst of the dot-com bubble of 2000, it can happen that the innovation is seen overly optimistic and leads to unrealistic expectations. In Gartner's Hype Cycle, this phase culminates in the "Peak of Inflated Expectations." In this phase, there may be some successful applications of a technology, but there are typically more failures.

According to Gartner, there is a point where it is clear that the excessive expectations cannot be met, and so the innovation gets less and less attention and enters the "trough of disillusionment"—it becomes unfashionable. Still some companies continue to use the technology to further explore the real benefits of the technology, and in doing this, they pass through the "slope of enlightenment." This learning phase identifies where the technology is useful and where not, how it should be used, and which disadvantages exist.

The final phase—for those technologies that do not become obsolete before reaching it—is the "Plateau of Productivity," a phase in which the benefits of the innovation are widely demonstrated and accepted. The final height of the

plateau (the visibility of the technology) depends on how applicable the innovation eventually is.

Agile Methods brought a new perspective to software development: the remarkable focus on Agility and the ability to adapt easily to a variation of the requirement or of the environment and to provide value to the customer. Thus, right after the publication of the Agile Manifesto, Agile Methods have been proposed as the universal solution to a common problem: the strong demand to higher flexibility during the development of software to increase productivity and customer satisfaction through the entire development of a software system. The problem is indeed extremely hard, and advocates of Agile Methods have been very effective in their propositions, so Agile Methods have immediately gained an enormous popularity and still they are popular; as an example you can count the hits on Google for software-related terms combined with the word "Agile" (see Table 5.1).

Table 5.1 Hits on http://www.google.com searching for generic software engineering terms in combination with the term "Agile" as of January 22, 2013

Search term	Hits on Google
"Agile software development"	2,630,000
"Agile product management"	830,000
"Agile testing"	508,000
"Agile web development"	230,000
"Agile management"	209,000
"Agile architecture"	143,000
"Agile database development"	108,000
"Agile planning"	98,200
"Agile modeling"	78,600
"Agile AJAX"	64,700
"Agile game development"	24,600
"Agile documentation"	18,400
"Agile offshoring"	16,400
"Agile SOA"	14,400
"Agile requirement engineering"	12,700

However, if we look more closely at the evolution of one of the most popular of Agile Methods, Extreme Programming (see Fig. 5.2), we notice that their popularity has followed quite closely the Gartner's Hype Cycle: after the first hype, there has been a period of disillusion.[1]

In this chapter we conjecture reasons for the rise and the fall of Agile Methods. The experience of this last decade evidences that Agile Methods are often praised for virtues they do not have and are often blamed for deficiencies that they also do not have. Using a well-known expression, Agile Methods are often criticized for

[1]This chart just shows how much Internet users were searching for the term "extreme programming"; the interpretation that the decreasing interest is because of a decreasing interest in the practice "extreme programming" is ours.

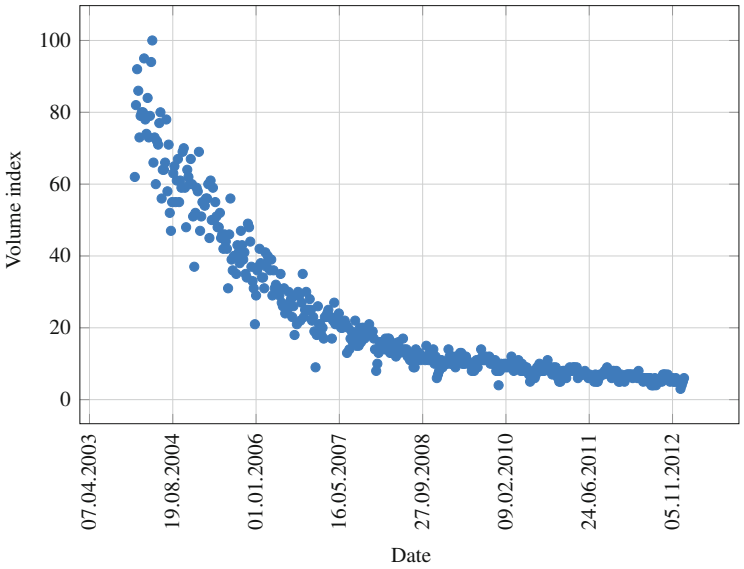


Fig. 5.2 Google Trends search volume index for the search term "extreme programming" as of January 22, 2013

their virtues and not for their vices. Understanding the real key limitations of Agility and the alleged limitations that are just wrong reading of the works of the "founding fathers" is an essential step to take full advantage of the excellent ideas present there to build solid software engineering processes and products.

We think that we passed the peak of inflated expectations for Agile Methods—the hype is over.[2] The attention to the topic "Agile" has reduced. See, for example, the number of papers with the terms "Agile," "extreme programming," or "scrum" in the title or abstract published in the years in the most important conference in Software Engineering, ICSE, the International Conference on Software Engineering (see Fig. 5.3).[3]

According to Garner's report "Hype Cycle for Application Development, 2013," "Project-Oriented Agile Development Methodology" is "sliding into the trough" [39] (see Fig. 5.4), which it was already in 2012 [12]. To begin to enter the slope of enlightenment and to reach the plateau of productivity, it is now necessary

[2]We are looking at the initial hype of Agile Methods after the Agile Manifesto. New methods are coming.

[3]The chart in Fig. 5.3 is quite selective. We looked only at one conference and searched for three search terms. Nevertheless, Scrum and Extreme Programming (or a combination of them) are the most popular Agile methods [26, 27, 29, 34, 37] and we think that the topics discussed at the top conference in software engineering reflect the interests of the community. Moreover, we think that papers published in ICSE represent a significant sample of the overall scientific production in software engineering.

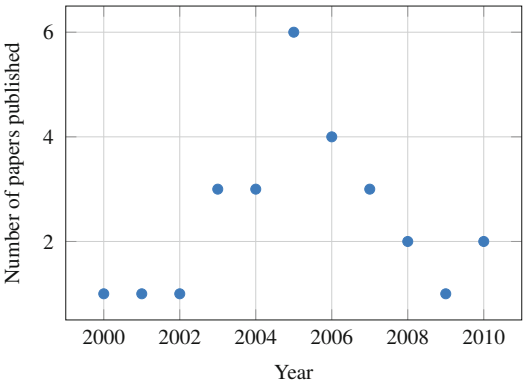


Fig. 5.3 Published articles (research and educational papers) containing the terms “Agile,” “extreme programming,” or “scrum” in their title or abstract on the International Conference on Software Engineering from 2000 to 2010

to understand the underlying principle of Agile Methods and to determine the time, the scope, the means, and the extent of their applicability. In this way software engineers and software companies can take full advantage of them, clearly when they happen to be useful.

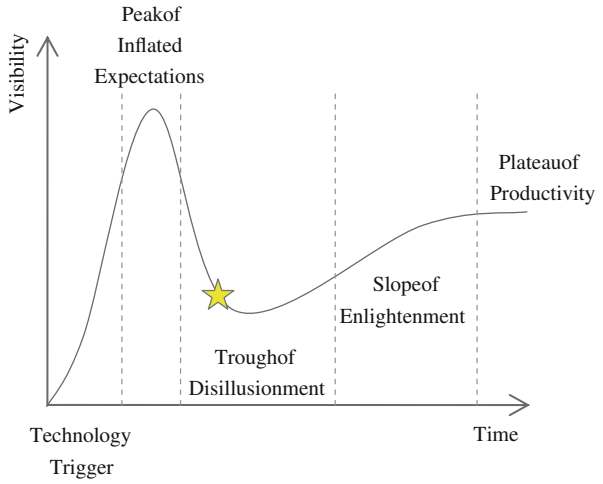


Fig. 5.4 Where the Gartner Group sees the “Project-Oriented Agile Development Methodology” now

5.2 The Dark Side of Agile

Agile Methods have generated a notable interest in the Software Engineering community as they address typical shortcomings of traditional software development. Today it is almost impossible to find a practitioner in software production that has not heard about Agile methods.

The mission of Agility is to flatten the "cost of change curve" (see Fig. 4.5), i.e., to increase the ability to change things also late in the development process to be able to produce value throughout the process.

The need to be more flexible is beyond any doubt overwhelming. This explains why Agility became immediately so popular: it was seen as the answer to such need. The metaphor used was luring: Agility is the opposite of large and heavy—so if software systems and software processes become too large and heavy, let us be Agile! It was so fashionable to be "Agile" that always more people claimed to be so, even if they were not. In particular, a number of consultants and managers adopted quite "liberally" the term Agility to promote their services [7]. Agility was then interpreted just as the cut in complexity, a rather simplistic and unjustified cut in complexity whatever the term complexity was—documentation, good development practices as design, etc. And this was against the ideas of the proposers especially the most remarkable—we recommend to read the seminal paper of Martin Fowler on Agility in design [13].

A "dark side" of Agility emerged.

In addition, most likely also against the wills of the proposers, enthusiastic early subscribers to the Agile Manifesto became zealot. They wanted to do more, to do better. So they read the Agile manifesto as in Fig. 5.5.

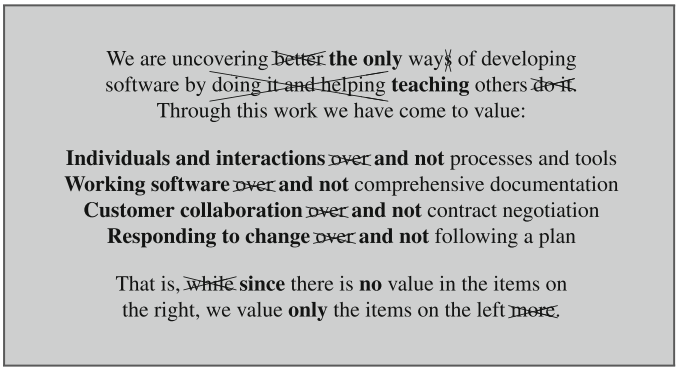


Fig. 5.5 The Dark Agile Manifesto [16–18]

These two phenomena together caused a set of misconceptions about Agile software development—still common today—such as that it is forbidden to document or to plan within Agile methods. The two phenomena also evidence what is still today missing in the application of the Agile Manifesto, and it is highlighted by the

last prescription of the Dark Agile Manifesto: teams need to understand how much to value the items on the left (how much working software, how much customer collaboration, how much responding to change) or the items on the right (how much planning, how much contract negotiation, etc.).

In fact, the misconceptions about Agile can be linked to the four statements of the "Dark Agile Manifesto." Rakitin et al. [25] claim that the right way to "translate" the Agile Manifesto is:

- **Individuals and interactions** over processes and tools: "Talking to people instead of using a process gives us the freedom to do whatever we want."
- **Working software** over comprehensive documentation: "We want to spend all our time coding. Remember, real programmers don't write documentation."
- **Customer collaboration** over contract negotiation: "Haggling over the details is merely a distraction from the real work of coding. We'll work out the details once we deliver something."
- **Responding to change** over following a plan: "Following a plan implies we have to think about the problem and how we might actually solve it. Why would we want to do that when we could be coding?"

Such "translations" show that to some, Agile Methods appear as approach that advocates coding without discipline, planning, and documenting: "this is nothing more than an attempt to legitimize hacker behavior [25]."

In programming, "a hack is something we do to an existing tool that gives it some new aptitude that was not part of its original feature set." [32]. So by "hacker behavior" Rakitin et al. claim that the adopters of Agile methods do not develop software following standards, processes, and policies, i.e., do not follow state-of-the-art practices that have been proven to be necessary or to contribute positively to the successful outcome of the project.

The Agile community developed another term to describe such behavior and to dissociate itself from it: the "cowboy coder." The "cowboy way" advises not to follow defined processes, standards, etc. to solve the required tasks faster (see Fig. 5.6).

A traditional software engineer, a cowboy coder and an Agile software engineer…

…chat with each other. Here is what they say:

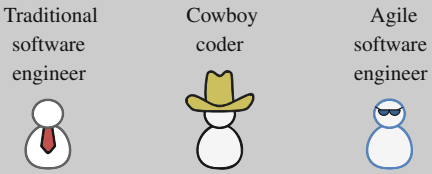


Fig. 5.6 How a traditional software engineer, a cowboy coder, and an Agile software engineer could look like

(continued)

TRADITIONAL SOFTWARE DEVELOPER: I need to have completed analysis and design before proceeding to code.
COWBOY CODER: I do not need any analysis and design.
AGILE SOFTWARE DEVELOPER: I do not need to have completed analysis and design before proceeding to code.

TRADITIONAL SOFTWARE DEVELOPER: I need to write all the documentation in a complete and pervasive way so that people in the future will be able to understand what is in here.
COWBOY CODER: I do not need any documentation.
AGILE SOFTWARE DEVELOPER: I need to write the code so that people in the future will be able to understand what is in here. I need to write only the documentation that is needed by people.

TRADITIONAL SOFTWARE DEVELOPER: Especially close to the deadline, I need to work like crazy to get the project done. Programming is hard.
COWBOY CODER: Only close to the deadline, I need to work like crazy to get the project done. Programming is fun.
AGILE SOFTWARE DEVELOPER: Especially close to the deadline, I need to work no more than 40 h a week to get the project done, keeping a constant pace and a fresh mind. Programming is fun.

TRADITIONAL SOFTWARE DEVELOPER: The code is mine and none is allowed to touch it!
COWBOY CODER: The code is mine and none is allowed to touch it!
AGILE SOFTWARE DEVELOPER: The code is of the team and everyone is allowed to modify it also extensively, provided the tests keep running!

TRADITIONAL SOFTWARE DEVELOPER: At the end we will do integration. It is going to be hard, so we need to define precise interaction protocols and to document them with maximal details.
COWBOY CODER: At the end we will do integration! No problem, it is easy: it will take 5 min.
AGILE SOFTWARE DEVELOPER: We need to integrate our system at least daily, so that at the end we will not have any problem.

TRADITIONAL SOFTWARE DEVELOPER: The customer should only see working and cleaned-up versions of the product. It is important to balance the contact with the customer so time is not wasted.
COWBOY CODER: If possible, the customer should only see the final versions of the product. It is important to minimize the contact with the customer so time is not wasted.

(continued)

AGILE SOFTWARE DEVELOPER: The customer should (a) be constantly exposed to the product being build and to the development team, and, whenever possible, (b) have a representative on site.

TRADITIONAL SOFTWARE DEVELOPER: If it is not broken, do not touch it!
COWBOY CODER: Even if it is broken, do not touch it! Try to hide it!
AGILE SOFTWARE DEVELOPER: Even if it is not broken, constantly refactor it! Use the test cases to ensure you do not introduce an undesired bug.

TRADITIONAL SOFTWARE DEVELOPER: Plan everything well in advance so that there will be no need to change! A change is a clear symptom of bad and/or not sufficient planning.
COWBOY CODER: Do not plan anything and try not to change! A change is a clear symptom of an annoying customer or manager.
AGILE SOFTWARE DEVELOPER: Plan everything that you can reasonably foresee and get ready to change! Changes occur naturally in software projects.

TRADITIONAL SOFTWARE DEVELOPER: Change is evil! Fight it!
COWBOY CODER: Change is evil! Fight it!
AGILE SOFTWARE DEVELOPER: Change is human! Get ready to cope with it!

We have seen in the previous chapter that Agile Methods, e.g., Extreme Programming, actually are based on Best Practices and follow state-of-the-art methods. Agile Methods acknowledge their importance and combine them in a way to maximize Agility, i.e., the ability of the development team to respond to changes of the requirements.

We partly attribute to "Agile extremists" that Agile methods acquired a reputation in the management community as a sloppy, undisciplined method to "hack," not develop software.

The questions are now how we can avoid Agile extremism, how to understand how much planning is needed to maximize value, when planning begins to destroy value, etc. The Agile Manifesto does not answer this question.

In fact, Agile developers pose themselves the question of how much planning is necessary to maximize the value, or in Lean terms, which is the minimal combination of activities that maximize value.

This way of thinking is quite new. It was introduced by Value-based Software Engineering [3, 4], a school of thought that criticizes that much of current software engineering practice and research is done in a value-neutral setting. Many treat every activity, requirement, use case, object, test case, defect, etc. as equally important. Value-Based Software Engineering promotes the idea to integrate value

considerations into the full range of existing and emerging software engineering principles and practices [3].

Value-based Software Engineering promotes the creation of value-based requirements engineering, architecting, design and development, verification and validation, planning and control, risk management, quality management, and people management, which consider the value they deliver to the stakeholders.

In Chap. 2 we analyzed how Lean Thinking aims to divide activities into:

- activities that **add value** and
- activities that **do not add value**; these activities can be further divided into:
  - activities that provide no value but **are needed**, e.g., because of current technologies, law requirements, or production methods;
  - activities that provide no value and **are not needed**, i.e., that can be removed.

We would like to apply this approach to Agile Methods by selecting the right amount of planning, contract negotiation, comprehensive documentation, and the use of processes and tools looking at the value they provide. One way to accomplish this, proposed by Value-based Software Engineering, is to consider the risk.

Risk is measured using risk exposure. Let us recall its definition:

$$\text{risk exposure} = \text{probability of an unsatisfactory outcome} \times \text{loss to the parties affected if the outcome is unsatisfactory}$$

The consideration of risk helps in this case to establish the optimal amount of the activity ("planning" in the following example); Fig. 5.7 shows how.

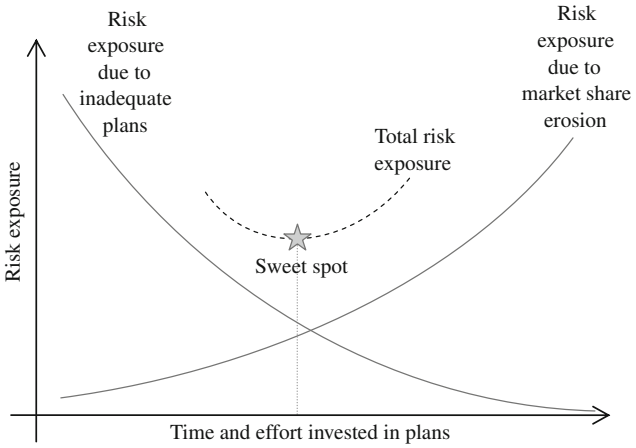


Fig. 5.7 Determination of the sweet spot [5]

On the horizontal axis we consider the time and effort invested in planning, on the vertical the risk exposure.

The first curve from the top-left to the bottom-right shows the risk exposure due to inadequate plans. If no planning is done, there is a high risk exposure due to a high probability that the project will fail if not enough planning is done and because this will result in a high loss. The other extreme is that a lot of time and effort is invested in planning; in this case the risk exposure can be reduced to a minimum.

The second curve from bottom-left to the top-right shows the risk exposure due to market share erosion (what we described with the falling value in Fig. 5.7). If we do not spend time and effort in plans and deliver quickly, the risk exposure is low. On the other hand, if we plan too much and we spend too much time, the risk exposure of the risk that our product will become obsolete will rise.

These two curves represent also the risk to destroy value: inadequate plans mean that we might invest effort in something that does not create value, and market share erosion means that we invested effort in something that is not needed anymore because the market now requires something else.

The optimal amount of time and effort to spend in planning is determined by the total risk exposure and its minimum, the "sweet spot" [5]. Risk exposure gives us the possibility to spend that amount of time and effort that maximizes value creation.

It is not possible to calculate a generic sweet spot for all companies in all markets. Its position depends on the environment.

For example, Fig. 5.8 shows the situation for a company in a rapidly changing market: the risk of losing customers because the product is on the market too late is higher than in the market of Fig. 5.7. We show this with a shifted curve for the risk exposure due to market share erosion. As a result, the total risk exposure also shifts to the left, resulting in a sweet spot that implies a lower amount of time and effort to invest in plans than before.

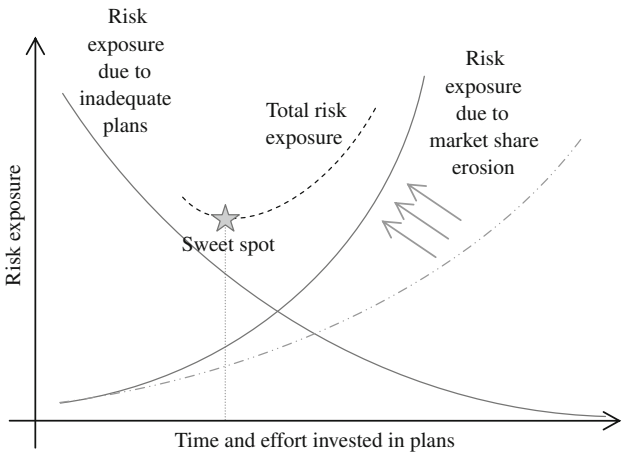


Fig. 5.8 Sweet spot in companies in turbulent markets [5]

Figure 5.9 shows the opposite example: this market faces a high risk if the product on the market is not working as planned, e.g., medical devices. As a result, the risk exposure due to adequate plans is higher, resulting in a right shift of the total risk exposure. This means that the optimal time and effort to invest in plans is higher than in Fig. 5.7.

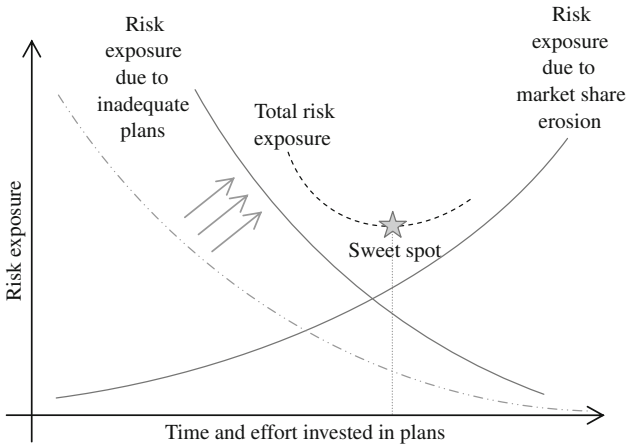


Fig. 5.9 Sweet spot in companies in markets requiring thorough planning [5]

This example has shown that to determine the value of activities in software development is not always an obvious exercise. The Agile manifesto claims: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." As an answer to our question of how much planning is needed to "deliver valuable software," Lean Thinking tells us to analyze planning and to get rid of all activities that are not adding value.

We approached the example of planning and analyzing the risk exposure as suggested by Value-Based Software Engineering. Still it is not clear how to obtain the risk exposure curves and how to determine our current location on these curves (how much time and effort are we currently investing in planning?).

We need a way to constantly measure the produced value of the performed activities. We will approach this later in the chapter about measurement.

5.3 The Skepticism Towards Agile Methods

The adaptive nature of Agile methods was conceived with a premise in mind: it is not possible to plan every detail in advance. This goes against conventional wisdom, which teaches us to "first think, then do." Software development is often compared with an idealized view of construction works: since it is possible to plan a house

completely and then to construct it according to the plan, the same must be possible for software.

Apart from the fact that such a view on construction words is simply wrong—also when building a house it is not feasible to plan everything (e.g., what kind of view you will have on the 3rd floor from the bedroom)—such expectations might lead to the conclusion that Agile software development does not follow a rational sequence of activities.

Another reason why Agile software development is not yet widely accepted is because it goes against the current best practices of project management. Software development projects are seen as any other project, and therefore, project managers assume they should be managed as any other project.

The Guide to the Project Management Body of Knowledge is a recognized standard for the project management profession [24].

Reading this standard brings one back to the times of the waterfall model: the generic project life cycle is shown as in Fig. 5.10.

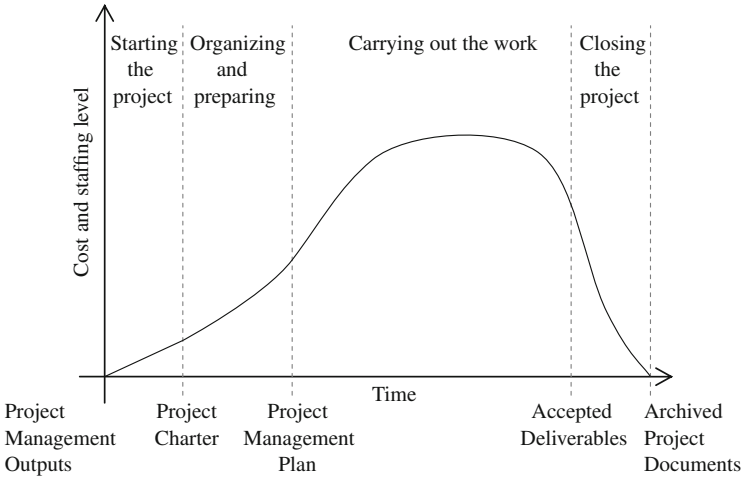


Fig. 5.10 Typical cost and staffing levels across the project life cycle [23]

The graph in Fig. 5.11 shows that what project managers trained using the PMBOK standard expect is a waterfall-like progression of the different activities of the project. The stakeholder influence, risk, and uncertainty at the beginning of the project are considered high but diminishing over time. The cost of changes are considered low at the beginning but increasing over time. Again, this resembles what we know from the waterfall approach.

The Guide to the Project Management Body of Knowledge just briefly (half a page) mentions the possibility to divide the project in iterations to reduce the complexity of a project or (also just half a page) that requirements and scope are difficult to define in advance.

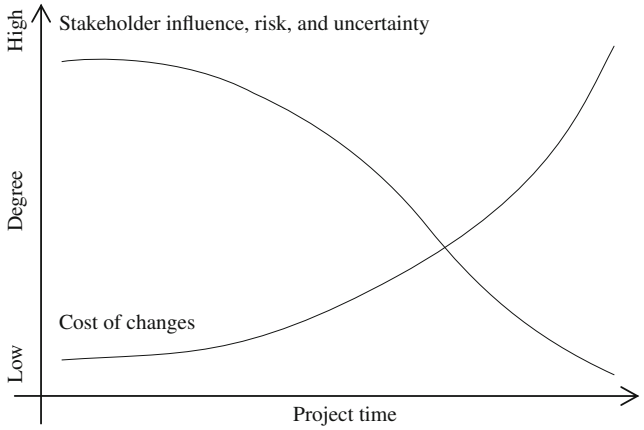


Fig. 5.11 Stakeholder influence, risk, uncertainty, and the cost of changes during the project [5]

Hence, we assume that managers without IT background expect to work like the standard teaches and will oppose an approach that appears to:

- **promote inadequate preparation**: Agile methods avoid big upfront designs as it is expected that requirements should be analyzed and studied in depth only a short time before their implementation;
- **accept exploding costs of changes**: the project manager assumes a traditional cost of change as shown in Fig. 5.11, while Agile methods assume a cost of change curve as shown in Fig. 4.5, i.e., that is flatter because of the adoption of tools and techniques that allow to change things also later in the project life cycle;
- **accept a high risk**: while in a traditional project the stakeholder influence, risk, and uncertainty are considered to diminish (see Fig. 5.11), Agile methods underline the importance of the possibility of the stakeholder to change the priorities of the project throughout the project.

To put it in other words, if one looks at Agile methods from the perspective of the Guide to the Project Management Body of Knowledge, the conclusions may be wrong because they are based on assumptions that differ from the assumptions of Agile methods.

The mismatch in the expectations originates from different approaches how project management, engineering, development, etc. is carried out in the different disciplines.

This mismatch can be seen also with other stakeholders, not only managers: according to our experience, when customers are not familiar with software engineering at all, they imagine it as something they know and behave accordingly. Some imagine it as building a house: assume that first you have to plan and then you construct according to the plan. Such customers react annoyed when you constantly

ask for feedback or requirements throughout the project since they planned to invest time in the project only at the beginning.

Another thinking model is to see software development as to cook—you just need to assemble different components in the right combination. When developers are confronted with problems they have never seen before and need time to develop a solution, this is seen as the fault of the developer that does not know all the available components. A similar way is to see software developers like configuring a video recorder: there are a predefined number of switches and buttons, and the skills of the developer decide whether he is able to set up the system in the right way or not.

In addition, only few customers are aware of the difficulty of developers to guarantee a correct functioning of the software since they are not domain experts.

In summary, different types of stakeholders have expectations that for them are obvious and therefore are not communicated. Nevertheless, it is necessary to deal with them: managers might come to the conclusion like [25], that what Agile methods are proposing does not lead to the aimed objectives, i.e., that Agile methods are not aligned to the business goals. Customers might come to the conclusion that these company is incompetent because its developers constantly ask for feedback.

To overcome these differences it is necessary to share the underlying assumptions. As recommended by meeting facilitators, in a project it is important to agree on:

1. "Where we are": an assessment of the current situation;
2. "Where we want to go": a description of the goals; and
3. "How to get there": a description of the method to achieve the goal.

The mismatch we described above originates from a different understanding of the three points above. Moreover, the mismatch in "how to get there" also related to a mismatch in the first two points. Is the company facing an unseen problem without detailed requirement but only some vague ideas or is the problem a standard problem that can be carried out using proven solutions? "Where we are" wants to agree on such questions to decide about "how to get there." The same counts for "where we want to go": do we give priority to end users or to software written according to a predefined specification?

We do not want to say that one option is always better than the other; what we want to point out is that these three points have to be communicated and agreed among all stakeholders [19]. We will see later how the Goal Question Metrics approach can help in doing this.

What is knowledge?

The term knowledge is part of the "Data Information Knowledge Wisdom Hierarchy" developed by Zeleny [40]. His knowledge taxonomy consists of four terms (see Fig. 5.12): data, information, knowledge, and wisdom. Each term represents a higher level of understanding [2].

(continued)

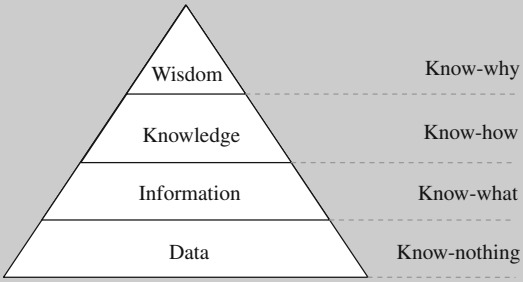


Fig. 5.12 The Data, Information, Knowledge, Wisdom pyramid [40]

Data are represented through symbols or signs. Examples of data are "h_2o," "yeast bacteria," and "starch molecules." Data alone are of no use; therefore, Zeleny calls data "know-nothing."

Information adds context to data so that they become useful. If we add context to the examples of data above and, e.g., know that we want to bake bread, we understand that we are looking at a list of ingredients: water, flour, salt, spices, etc. This level of understanding is also called "know-what." It also includes the recipe for making bread that describes a number of steps to do with the ingredients.

Knowledge is described as "know-how;" it refers "to observer's distinction of 'objects' (wholes, unities) through which he brings forth from the background of experience a coherent and self-consistent set of coordinated actions [40]." Knowledge connects data and information into a network of relations. In the bread baking example, knowledge describes the state that one knows several recipes, understands the functions that the different ingredients have in the recipe, and understands what happens if he forgets to add one ingredient.

Wisdom goes again one step further and links knowledge with a goal. In our taxonomy, wisdom is described as "know-why": we know why the different ingredients work and why we are using them. In our example this would mean that we know why we are using water, flour, salt, and spices to make bread and not croissants.

According to Ackoff, data, information, and knowledge are past oriented, while wisdom is future oriented [2].

5.4 The Zen of Agile

Agile methods have been conceived and refined by "gurus," e.g., Extreme Programming by Kent Beck, Crystal Clear by Alistair Cockburn, and Scrum by Ken Schwaber. This might be one of the intrinsic reasons for the problems we mentioned before: the language of the gurus must be persuasive and often elusive.

The guru is the person with wisdom—he knows what, when, and how things should be done to achieve the desired goal. It is the interest of the guru to hide the assumptions on which the rules are based and on which previous works he or she based his findings and how he verified that what is claimed really works. For example, it is not clear—reading the Agile Manifesto—why it is good to "focus on individuals and interactions over processes and tools": it is not stated why and how software development should benefit from it.

The guru has no advantage in making her or his followers independent adults; otherwise his role as a guru would vanish. In this way, the adopters remain dependent on the guru: the adopters need the guru to continue to use the method in cases not described by the guru upfront, e.g., how it can be extended, in which order the different practices should be adopted, etc.

The described strategy of the gurus works because a precondition is met: the followers of the guru—programmers, managers, requirement engineers, project managers—are looking for a silver bullet: they want a simple, safe method that solves their problems. And gurus are willing to give it to them.

This situation leads to a number of issues because we have to treat the method as a black box: it is not possible to use its nuts and bolts for an evaluation; they have to be reengineered (or ignored) if anything else than the entire package has to be evaluated.

Having to treat an Agile Method as a black box, we do not know the rationale behind the different constituent parts. Whenever we have to assess something complex, conventional wisdom tells us to verify its constituent parts. Let us say we want to evaluate the quality of a car. One step would be to check the quality of the tires, the engine, etc. If we want to do that for an Agile Method, we have to conduct a study on our own (as in [36]), since "Agile Method practices sometimes lack clarity and the underpinning rationale is often unclear [9]."

Some authors define Agile methods, Extreme Programming in particular, as irrational and vague [22, 33]. Other authors claim that the Agile Method practices are on a too high level of abstraction which causes an inconsistent interpretation and implementation [1, 5, 21].

Not having a clearly stated rationale behind methods and practices makes it also difficult to tailor a method to specific needs [6, 14, 31, 35]. Without this knowledge we do not know what we are risking if we omit one method or if we extend another.

Stephens and Rosenberg [33] compare Extreme Programming with a ring of poisonous snakes, daisy-chained together (see Fig. 5.13). Each snake represents a practice that has issues that are compensated by the use of another practice. "All it takes is for one of the snakes to wriggle loose, and you've got a very angry, poisonous serpent heading your way. What happens next depends on the snake and to an extent on how good you are at damage control [33]."

The authors show that the different practices depend on each other such as:

1. Requirements are elicited gradually, which—since developers do not have a complete picture—can result in wasted effort. This problem is alleviated by incremental design, i.e., to be ready to change the design incrementally.

2. Incremental design avoids designing everything upfront to minimize the risk of being wrong. This practice is considered safe because the code is constantly refactored.
3. Constant refactoring involves rewriting existing code that was thought to be finished: it went through discussions and thinking to get the design right. It potentially introduces new defects, but it is considered safe because of the presence of unit tests.
4. Unit tests act as a safety net for incremental design and constant refactoring. With unit tests it is difficult to verify the quality of the design of an architecture. Pair programming alleviates this problem.

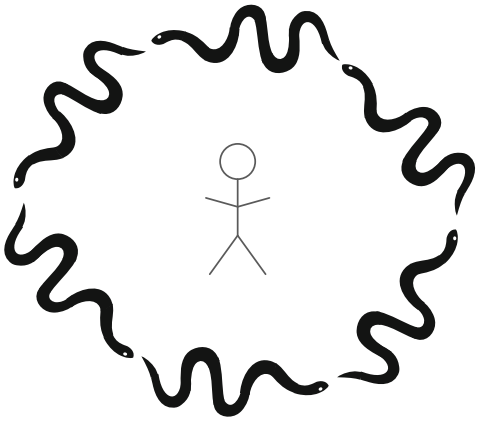


Fig. 5.13 Extreme Programming from the point of view of Stephens and Rosenberg [33]

This shows that if we want to reach the plateau of productivity, i.e., if we want to be able to use this method efficiently and effectively, we must understand when Agile Methods pay off, why they work, how they work, how they can be adapted, the understanding of the value of the single activities, etc. [15].

Taking the statements of the Agile manifesto, we should be interested on how to manage knowledge on how to value individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.

To gain and share knowledge is one of the central points of Lean Thinking. In fact methods such as just-in-time production bring problems quickly to the surface. It has—among other effects—the effect to uncover problems and forces everybody to solve the problem right away.

As described in Chap. 2, every worker has the right to stop the production line if a problem occurs. This already tells to everybody that there is a problem. A signal ("Andon" in Japanese) notifies all other workers where the problem took place so that workers that have the skills to help can jump in (see Fig. 5.14).

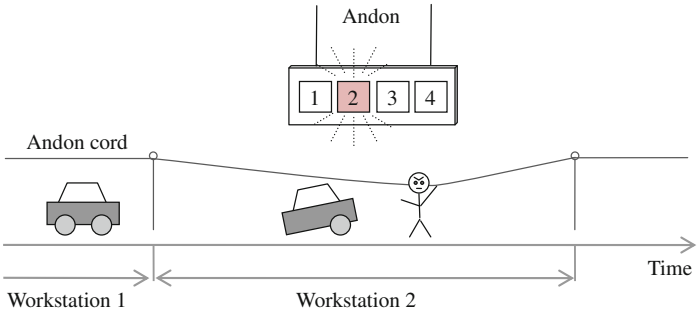


Fig. 5.14 The Andon notifies everybody of the state on the production line

Extreme Programming, e.g., promotes the "Informative workspace" practice, and uses a task board to visualize the current status of the project. Usually divided into different sections that describe the status of each work station, user stories are placed on the task board (see Fig. 4.12) depending on their status. So it is possible to get an overview of the entire project on one sight.

It is important to understand that if we take all these tasks and put them into a system where the location of the task does not represent the phase in which the task is currently, but where, for example, we have a text field, in which we store the "phase," many advantages of the task board are lost. In fact, the task board embodies the knowledge that:

- a task can only pass through a predefined set of phases;
- the first phase is the leftmost;
- the last phase is the rightmost;
- the sequence of phases is defined as the phases from left to right;
- many tasks stuck in the same phase signal a problem (the space for one phase is limited).

The task board of Fig. 4.12 embodies what we described in Chap. 2 with "visual control." The task list of Table 5.2 does not show that phase 9 of the 4th task does not exist—the task board makes it impossible to add a task to a phase that does not exist.

Table 5.2 Example of a task list

ID	Description	Phase
1	Set export filename to "export.xml"	1
2	Add feature to print in draft mode	4
3	Fix app crash when opening jpg file	2
4	Remove "save as pcx" option	9

There are other examples of showing progress visually: Scrum [30] or Crystal Clear [8] use "burn-down charts," charts that show the development progress against the predictions. An example is given in Fig. 5.15: on the vertical axis, the remaining story points are shown and on the horizontal axis, the iteration. As development progresses, developers implement user stories and the remaining story points diminish. Jumps show when the original scope was modified: a jump upwards shows that after one iteration, the team faced a new total amount of story points that had to be done to finish the project, and a jump downwards means that the scope was decreased.

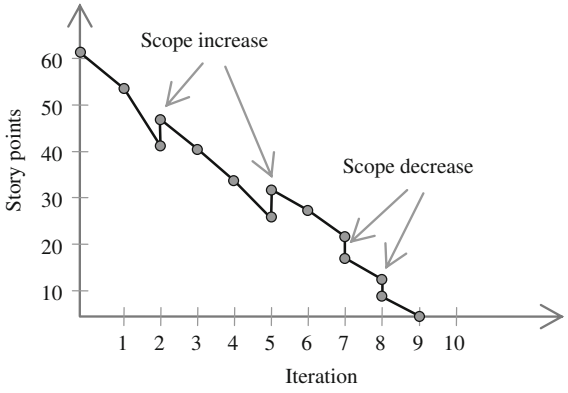


Fig. 5.15 Burn-down chart

The burn-down chart can be used to predict when the project will finish according to the velocity of the team (i.e., how many story points the team is able to finish in one iteration). Figure 5.16 shows how: let us assume we are in the 6th iteration. The burn-down chart is shown on the top of Fig. 5.16; the velocity of the team after each iteration is shown on the bottom.

Using the information about the past velocities, it is possible to estimate the end of the project. For example, in Fig. 5.16, taking the average of the last three velocities, we predict that this project (without further scope changes) will finish after the 10th iteration.

In Chap. 1 we emphasized one of the main problems of software: it is invisible. For this reason it is very easy for the project to continue for a considerable time before problems become apparent [28].

Coming back to the Agile manifesto, we need to find ways to systematically collect, organize, and share knowledge about software development and to rise our understanding of the effects of our actions, similar to the burn-down charts that visualize progress in value creation and the effects of scope changes to all stakeholders.

This will be the content of the following chapter about experience management.

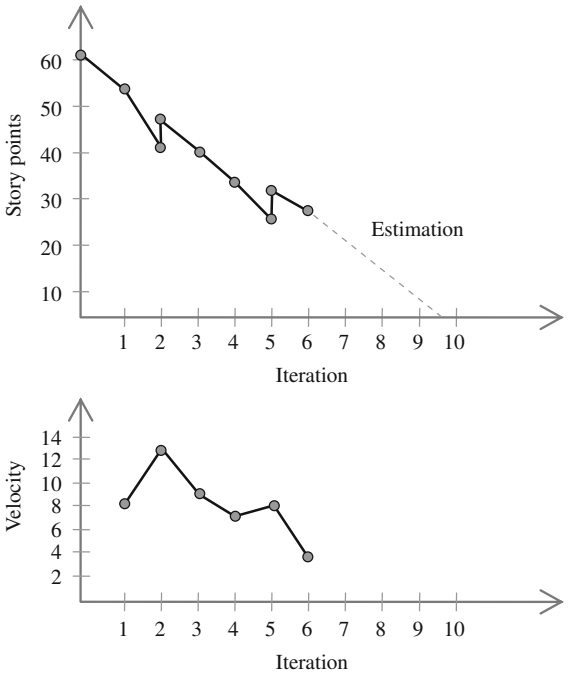


Fig. 5.16 Estimation with burn-down charts

5.5 Summary or "What Stops us from Moving from Agile Towards Lean Software Engineering?"

"Agile thinking" and Lean Thinking are related [10,38]: if we understand Agility as the ability to change quickly and leanness as the ability to be effective, then leanness includes agility: if I'm Lean, I can change quickly. This does not hold the other way around: if I can change quickly, I am not necessarily Lean. I could change quickly in an ineffective way, wasting resources.

Lean Thinking tells us to focus on value, i.e., to clarify where we want to go and our understanding of how to get there. The misconceptions, the inherent guru approach, and therefore unknown background of Agile Methods stop companies from using Agile Methods effectively and also to become Lean.

The knowledge—understood as "know-how"—that has to be internalized by all stakeholders has to be gathered, documented, and shared. The "guru approach" packages knowledge into simple, clear practices which are easy to explain and to follow. Nevertheless, it is not the right approach since it makes it difficult to understand when and which practice is appropriate, how to collect knowledge about the production process, and how to improve.

In Chap. 3 we identified three aspects that Lean Thinking emphasizes: the focus on value, knowledge, and improvement. We now want to explore possibilities to answer the following questions:

1. What is the value of every step? How much is enough? When is the point reached where value is not increased anymore but destroyed?
2. How can I obtain knowledge about my production process? How can I create visibility of the ongoing activities or problems? How can I store knowledge to create experience? How can I design the production process so that the team uses the gained experience?
3. How can we systematically improve? How can I build on the experience to anticipate problems (create wisdom)? How can I make sure that some mistakes never happen again?

We will look in the next chapters how three technologies—the Goal Question Metric Approach, the Experience Factory, and Non-invasive Measurement—can be used to overcome the described impediments.

Problems

5.1. How would you correct Fig. 5.10, the typical cost and staffing levels across the project life cycle, and Fig. 5.11, the stakeholder influence, risk, uncertainty, and the cost of changes during the project for an Agile project?

5.2. Imagine you are the boss of a small software development company. Which actions would you do or which practices would you introduce to prevent that your programmers fall into the trap of following a software guru?

References

1. Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile Software Development Methods—Review and Analysis, vol. 478. VTT Publications, Espoo. Online: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf (2002). Accessed 4 Dec 2013
2. Ackoff, R.L.: From data to wisdom. J. Appl. Syst. Anal. **16**, 3–9 (1989)
3. Boehm, B.W.: Value-based software engineering. ACM SIGSOFT Softw. Eng. Notes **28**(2), 3 (2003)
4. Boehm, B.W.: Value-based software engineering: overview and agenda. In: Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., Grüunbacher, P. (eds.) Value-Based Software Engineering. Springer, Berlin/Heidelberg (2006)
5. Boehm, B.W., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Longman, Boston (2003)
6. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Inf. Softw. Technol. **38**(4), 275–280 (1996)
7. Ceschi, M., Sillitti, A., Succi, G., Panfilis, S.D.: Project management in plan-based and agile companies. IEEE Softw. **22**(3), 21–27 (2005)

8. Cockburn, A.: Crystal Clear a Human-Powered Methodology for Small Teams. Addison-Wesley, Boston (2004)
9. Conboy, K., Fitzgerald, B.: The views of experts on the current state of agile method tailoring. In: McMaster, T., Wastell, D., Ferneley, E., DeGross, J. (eds.) Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda. IFIP International Federation for Information Processing, vol. 235. Springer, New York (2007)
10. Dall'Agnol, M., Janes, A., Succi, G., Zaninotto, E.: Lean management — a metaphor for extreme programming? In: Marchesi, M., Succi, G. (eds.) Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering (XP). Lecture Notes in Computer Science, vol. 2675. Springer, Genova (2003)
11. Fenn, J., Raskino, M.: Mastering the Hype Cycle: How to Choose the Right Innovation at the Right Time. Gartner Series. Harvard Business Review Press, Boston (2008)
12. Finley, I., Wilson, N., Huizen, G.V.: Hype cycle for application development, 2012. Online: http://www.gartner.com/id=2098316 (2012). Accessed 4 Dec 2013
13. Fowler, M.: Is design dead? In: Succi, G., Marchesi, M. (eds.) Extreme Programming Examined. Addison-Wesley, Boston (2001)
14. Grundy, J.C., Venable, J.R.: Towards an integrated environment for method engineering. In: Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering on Method Engineering: Principles of Method Construction and Tool Support: Principles of Method Construction and Tool Support. Chapman & Hall, Atlanta (1996)
15. Janes, A.: Measuring the effectiveness of agile methodologies using data mining, knowledge discovery and information visualization. In: Marchesi, M., Succi, G. (eds.) Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering (XP). Lecture Notes in Computer Science, vol. 2675. Springer, Genova (2003)
16. Janes, A., Succi, G.: The dark side of agile software development. In: Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH), Onward! ACM, Tucson (2012)
17. Janes, A., Succi, G.: The dark agile manifesto. Online: http://www.darkagilemanifesto.org (2013). Accessed 4 Dec 2013
18. Janes, A., Succi, G.: The dark side of agile software development, first results. In: Wuksch, B. D., Peischl, B., Kop, Ch. (eds.) Proceedings of the 11th User Conference on Software Quality, Test, and Innovation (ASQT). Österreichische Computer Gesellschaft, Graz (2014)
19. Janes, A., Šarūnas Marciuška, Sarcià, A., Succi, G.: Domain analysis in combination with extreme programming to address requirements volatility problems. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute, Boston (2013)
20. Järvenpää, H., Mäkinen, S.J.: Empirically detecting the hype cycle with the life cycle indicators: an exploratory analysis of three technologies. In: Proceedings on the International Conference on Industrial Engineering and Engineering Management (IEEM). IEEE, Singapore (2008)
21. Koch, A.S.: Agile Software Development: Evaluating The Methods For Your Organization. Artech House Computing Library. Artech House, Boston (2004)
22. McBreen, P.: Questioning Extreme Programming. Addison-Wesley Longman, Boston, MA, USA (2002)
23. Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 4th edn. Project Management Institute, Newtown Square (2008)
24. Project Management Institute: A Guide to the Project management Body of Knowledge (PMBOK® Guide), 5th edn. Project Management Institute, Newtown Square (2013)
25. Rakitin, S.R., Donné, W., Holmes, N., de Boer, B.: Letters: Manifesto elicits cynicism; more markup remarks. IEEE Comput. **34**(12), 4 (2001)
26. Regis, B.N.N.: Evaluation of the Most Used Agile Methods (XP, LEAN, SCRUM): With the Definition of Quality Developed by Toyota. LAP Lambert Academic Publishing, Germany (2012)

27. Rodríguez, P., Markkula, J., Oivo, M., Turula, K.: Survey on agile and lean usage in finnish software industry. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, Lund (2012)
28. Royal Academy of Engineering and British Computer Society: The Challenges of Complex IT Projects: The Report of a Working Group from the Royal Academy of Engineering and the British Computer Society. The Royal Academy of Engineering. Online: http://www.bcs.org/upload/pdf/complexity.pdf (2004). Accessed 4 Dec 2013
29. Rubin, K.S.: Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley Signature Series (Cohn). Pearson Education, Upper Saddle River. http://books.google.at/books?id=3vGEcOfCkdwC (2012)
30. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press, Redmond (2004)
31. Smolander, K., Tahvanainen, V.P., Lyytinen, K.: How to combine tools and methods in practice — a field study. In: Steinholtz, B., Sølvberg, A., Bergman, L. (eds.) Advanced Information Systems Engineering. Lecture Notes in Computer Science, vol. 436. Springer, Berlin/Heidelberg (1990)
32. Stafford, T., Webb, M.: Mind Hacks: Tips & Tricks for Using Your Brain. Hacks Series. O'Reilly Media, Sebastopol (2004)
33. Stephens, M., Rosenberg, D.: Extreme Programming Refactored: The Case Against XP. APress, Berkeley (2003)
34. Szőke, A.: Optimized feature distribution in distributed agile environments. In: Proceedings of the 11th International Conference on Product-Focused Software Process Improvement (PROFES). Springer, Limerick (2010)
35. Tolvanen, J.P., Lyytinen, K.: Flexible method adaptation in case: the metamodeling approach. Scand. J. Inf. Syst. **5**, 51–77 (1993)
36. Vanderburg, G.: A simple model of agile software processes—or—extreme programming annealed. In: Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, San Diego (2005)
37. VersionOne.com: 8th annual state of agile survey. Online: http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf (2014). Accessed 4 May 2014
38. Wang, X., Conboy, K.: Comparing apples with oranges? the perceived differences between agile and lean software development processes. In: Galletta, D.F., Liang, T.P. (eds.) Proceedings of the International Conference on Information Systems (ICIS). Association for Information Systems, Shanghai (2011)
39. Wilson, N., Huizen, G.V., Prentice, B.: Hype cycle for application development, 2013. Online: https://www.gartner.com/doc/2560015, (2013). Accessed 4 May 2014
40. Zeleny, M.: Management support systems: towards integrated knowledge management. Hum. Syst. Manag. **7**(1), 59–70 (1987)

Chapter 6
Enabling Lean Software Development

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.

(Consider ye the seed from which ye sprang;
Ye were not made to live like unto brutes,
But for pursuit of virtue and of knowledge.)

Dante Alighieri, Divina Commedia, Inferno, Canto 26, 118–120

*Lazily, Uli went back to his office. He called Euril, Perim, Sinon, and Elp. "Knowledge, value, improvement—this is what they asked me to clarify... Any suggestion guys?" None spoke; they were all still a bit shocked of all what happened the day before. "Well, I had a meeting with our top managers and actually there were not so excited of keeping the project running. It appears that for some of them not altering the consolidated way of doing business is more valuable than keeping a project and trying something new. Can you believe this?" "Indeed, said Perim, indeed. The good-old-way does not jeopardize any position while trying something new may alter the current status of businesses. But you should also consider the point they raised. I do think that there is something to think. Yesterday I asked something similar and you answered with a typical consulting term... what was it? Situational awareness—the usual consultant b***s**t!"*

Uli stood, joined his hands "Ladies, gentlemen" he started "can you consider why you started to work in software? What were your aims, your desires? All of you knew at the time that this was the land of unknown, the land of discovery, the land of opportunity. You selected software engineering because you aimed to increase your knowledge and the knowledge of the human being, still being able to do something concrete; you wanted to pursue your virtues. If you wanted to stay in a more comfortable, secure, quite discipline, you would have built houses and not software systems. If you wanted to be creative but not attached to the reality, you would have studied management. But now, you are a software engineering—someone who combines quest for the unknown and the ability to build solid elements, someone who put knowledge, and also value and improvement at the top of his priorities."

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_6

None said a word. Uli sat. Everyone was silent and it was a very loud silence. The walls were speaking: two posters dominated the scene. On one side a big poster contained an artistic painting of stars in the Austral hemisphere on the 9^{th} of April. It was a bit surreal but beautiful. In the spare time Uli used to joke with his senior architects telling them that he would have liked to be the first western man crossing the ocean to see such scene, or that he should have rented the space shuttle to take them all for a space tour as a prize for completing the project on time. On the other side there was the picture of an ancient Greek ship sinking during a horrendous nightly hurricane, with a mountain in the background and under a pale light of the moon; this picture had always been there, Uli disliked it but never took it away, as, he said, reminded always the risk of a failure.

After this heavy silence and after staring at the two pictures Perim said: "I want a ticket for the space shuttle!" and left the room. All the other also left.

At that point Uli knew what to do.

6.1 Introduction

The concept of Lean Software Development refers to the several attempts to transfer and adapt the principle of Lean Management into Software Engineering.

Remember, Lean Management was conceived in Japan by Taiichi Ono [21] for the automobile industry. Posting it to the software industry is not straightforward.

At the time of writing this book, a detailed methodology that fully applies the principles of Lean Management to Software Engineering does not exist yet. We have the impression that existing approaches do as if writing software would be similar to producing a car and ignore that software is invisible. A comprehensive measurement approach is needed that is aligned with the organizational goals as evidenced by the founders of Lean.

Current approaches emphasize different values of Lean but neglect the need of instilling a measurement and an experience management culture to overcome the invisibility of software and to improve its development constantly.

6.2 Existing Proposals to Create "Lean Software Development"

Practitioners and academics are exploring the terrain to adopt Lean ideas within software development. Pioneers in this exploration are Mary and Tom Poppendieck. In their book "Implementing Lean Software Development From Concept to Cash" [23], they characterize Lean software development with the following seven principles:

1. Eliminate waste;
2. Build quality—we used the terms "autonomation" and "standardization";
3. Create knowledge;
4. Defer commitment—we used the term "just-in-time";
5. Deliver fast—get frequent feedback from the customer and increase learning through frequent deployments;
6. Respect people—we used the term "worker involvement";
7. Optimize the whole—we used the term "constant improvement."

Furthermore, to respect people means that the knowledge people accumulate during their work is acknowledged and that they are given the possibility to change the working processes. According to them, this has consequences on different levels:

- **Entrepreneurial Leadership**: the leader promotes committed and thinking people and concentrates their efforts on creating a product that provides maximum value to the customer.
- **Expert Technical Workforce**: the company makes sure that the technical expertise is nurtured and that teams have the expertise needed to accomplish their goals.
- **Responsibility-Based Planning and Control**: teams are organized using "Management by Objectives" (see Chap. 4) and people are trusted to self-organize to achieve their goals.

Curt Hibbs and his colleagues have developed a different proposal to adapt the principle of Lean Management to Software Engineering. Their approach is more oriented to the code. In their book "The Art of Lean Software Development, A Practical and Incremental Approach," [13] propose the following practices:

1. Source Code Management and Scripted Builds;
2. Automated Testing;
3. Continuous Integration;
4. Less Code;
5. Short Iterations; and
6. Customer Participation.

These principles are an implementation of the concept of autonomation to coding.

The use of source code management and scripted builds together with automated testing is one way to instantiate autonomation for software engineering.

Automated, scripted builds and automated testing autonomate coding because they detect if the produced source code does not conform to the expectations and stop the production process. Therefore, they contribute to avoid committing defective code to the production code.

Continuous integration is also a form of autonomation. By continuously integrating the different parts of code, all the problems related to integrating different parts of the software system become immediately evident and the production process does not proceed forward until the issues that break the integrations are solved.

Short iterations and customer participation enable the team to obtain frequent feedback and to improve the understanding of what creates value for the customer.

In summary, the proposal of Mary and Tom Poppendieck aims to "leanify" the overall process while Hibbs and his colleagues specify how coding can be made Lean.

Alan Shalloway et al. [25] take a more comprehensive perspective and propose a "Lean-Agile software development." Their approach is more generic than the two previously presented and organizes the transition of Lean Thinking to Lean Software Development into the following layered model (see Fig. 6.1):

1. **Foundational Thinking**. The underlying belief system of Lean Thinking, based on the work of Deming.
2. **Perspective and Principles**. The Perspective is the choice of what is considered important to observe in the process. The Principles are the rules of behavior that adhere to the Foundational Thinking and are taken from the work of Mary and Tom Poppendieck.
3. **Attitudes**. The choice of what is considered important and what is not.
4. **Knowledge**. "Know-how" based on experience or, in other words, "lessons learned."
5. **Practices**. Recommendations on what to do, based on the knowledge acquired.

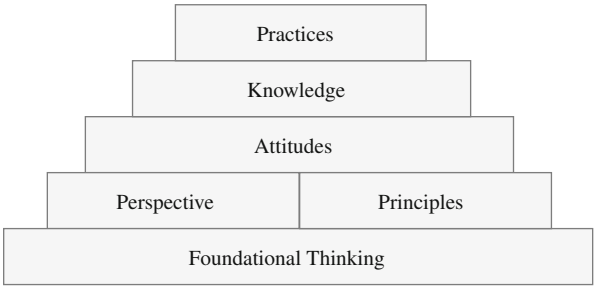


Fig. 6.1 The layered structure of Lean-Agile software development [25]

What is continuous integration?

Martin Fowler and Kent Beck were the first to write about continuous integration. We use ten practices proposed by Fowler [10] to explain this concept:

(continued)

1. **Maintain a single source repository**: maintain all resources of one software project in one place;
2. **Automate the build**: automate all steps to transform source code into a running system.
3. **Make your build self-testing**: include automated tests in the build process and execute them after building a new release.
4. **Everyone commits to the mainline every day**: the more often everybody commits to the mainline, the lower the effort of resolving conflicting changes by different developers becomes.
5. **Every commit should build the mainline on an integration machine**: because of different reasons (e.g., undisciplined developers, environmental differences between the developer machine and the integration machine, etc.), tests can still fail on the integration machine. Therefore, every commit should start an automatic build and test on the integration machine.
6. **Keep the build fast**: the faster the build, the faster the feedback that is given to the developer, and the lower is the risk that other developers are making their modifications based on the defective code, increasing the caused damage.
7. **Test in a clone of the production environment**: test your build in an environment that is similar to the production environment.
8. **Make it easy for anyone to get the latest executable**: put the latest executable on a well-known place to allow demonstrations and exploratory testing, find out about changes, etc.
9. **Everyone can see what's happening**: communicate to everybody the state of the build.
10. **Automate deployment**: to test the developed code in multiple environments, it is important to automate the necessary deployment steps.

Table 6.1 contains some of the tools currently available to support the different phases of continuous integration.

Table 6.1 Tools to support continuous integration

Tool	Useful for step
Subversion [2], GIT [12]	1
Make [11], Apache Ant [1]	2
Unit testing frameworks (known as xUnit frameworks) such as Junit for Java [16] or CppUnit for C++ [7], GUI testing frameworks such as Sikuli [26]	3
CruiseControl [8], Jenkins [15]	5
VMWare [31], VirtualBox [22]	7, 10

The three examples presented above (the proposal of Mary and Tom Poppendieck and of Curt Hibbs and his colleagues and the approach of Alan Shalloway et al.) show that Lean Thinking can be translated in different ways into software engineering. However, all these three approaches lack an essential component of Lean Management, its concrete use of real measurements supporting the process [19]. They are more faith-based, while Lean advocates a constant and concrete analysis of the process to produce value and eliminate waste.

Concretely, our approach is to develop a Lean software development process that avoids the three issues we identified in the previous chapter:

1. the problem of communicating the goals and methods of Agile methods to stakeholders, which generates skepticism since Agile methods seem to ignore "well-known" best practices;
2. the guru approach that has dominated the way Agile ideas became known among practitioners; and
3. Agile extremists that promote the dark side of Agile.

Now we describe how we want to tackle these issues.

6.3 Share a Common Vision

Lean Thinking advocates new, unconventional methods for producing goods. It is essential that these methods can be explained solidly to our customers. They should not think that we are "original." They should understand such methods and, at least, understand that they are grounded in solid theories. Otherwise, we would not have customers, or, worse, we might get customers who want to adopt our proposals simply because they are cool, and when their coolness will go away, we will not have anymore a job. It is interesting to note that several Agile projects had this fate, despite being successful.

Therefore, we need to communicate to our customers how we work, what we do, what outcomes we expect from it, and which support we need from them.

Agile methods heavily rely on the collaboration with the customers. Extreme programming, for example, has a practice called "customer on-site," requiring customers to sit with the project team throughout the project and to supply the essential knowledge of the applicative domain whenever needed and to help the team to stay focused on the common goal.

However, customers have their own priority. In most cases, their ideal relationship with the developers is that they communicate shortly their desires, and, after a certain amount of time (the sooner, the better), they get what they dreamed at. Van Deursen [30] has identified three major causes why it is hard to have customers on-site.

Actually, it turns out to be difficult to convince the customer that it is worth to collaborate personally and continuously [30]:

- customers have to do their regular work and be on-site, which is not always possible;
- the customer usually wants to buy a "whole solution," and not to run a customization project requiring his involvement; and
- the best customers from a programmer's perspective are also often best in other aspects, which makes them busy, and it is unlikely to allocate to the project all the required time.

Some customers expect software development to be like building a house; they want the "whole solution," the "turn-key project." They want to get the solution in a ready-to-use condition. Such customers think: "Why do you ask me? You are the expert, you should know. Why am I paying you?"

Having the customer on-site, we are only halfway through: establishing a fruitful communication with the customer is also challenging. Some reasons for this are [30]:

- Technologists and end users have a high "semantic gap," which makes communication complicated. Both sides base their communication on assumptions. If some information is based on an assumption that the other side does not know about, this information might be not interpreted as intended by the speaker. Making these gaps explicit, i.e., talking about the hidden underlying assumptions, is perceived as an annoying, boring activity.
- Neither developers nor customers consider talking to each other a useful task, but rather a waste of time.
- End users may resist changes in their way of working, making it very hard to involve them in a constructive way in the customization of the product.
- Developers might be against an on-site customer. Beck and Andres [4] call it the "sausage factory" effect when the developers think: "if the customers knew how messed up software development was, they would never trust us."

From a financial point of view, a trade-off exists between having a customer on or off site. Let us analyze the trade-off looking at a client organization "BusyClient" that hires the software organization "AgileCoders" to produce a tool called "SuperTool." BusyClient has to decide if its best sales representative, Mr. Seller, should act as a customer on-site.

If Mr. Seller works as an on-site customer at AgileCoders, he is not working as a sales representative, but as a requirement analyst. He can work a bit while sitting at AgileCoders, but he cannot leave to visit customers. This causes considerable costs for BusyClient and increases the total development costs of SuperTool.

The alternative is that Mr. Seller helps only off-site, which makes it harder for AgileCoders to obtain a clear understanding of the requirements since Mr. Seller is sometimes busy and not reachable when he is talking with customers.

AgileCoders might waste time because of misunderstandings, wrong assumptions because Mr. Seller is not available, and so on. This might then delay the shipment of SuperTool, which increases again the costs for BusyClient.

Additionally, because the shipment of SuperTool is delayed, AgileCoders is also facing higher costs compared to a scenario in which Mr. Seller acts as a customer on-site. AgileCoders, which wants to survive on the long run, has to charge BusyClient with this additional development costs.

In summary, BusyClient has to take decision: is it more costly to have Mr. Seller work for a while at AgileCoders or to pay more for SuperTool?

BusyClient can decide using the concept of risk exposure explained previously. The risk exposure of losing business opportunities increases the more Mr. Seller is absent from BusyClient. On the other side, the risk exposure of an expensive development of SuperTool decreases with the amount of time Mr. Seller invests to manage requirements at AgileCoders (see Fig. 6.2). BusyClient should consider the total risk exposure and choose the sweet spot that minimizes it.

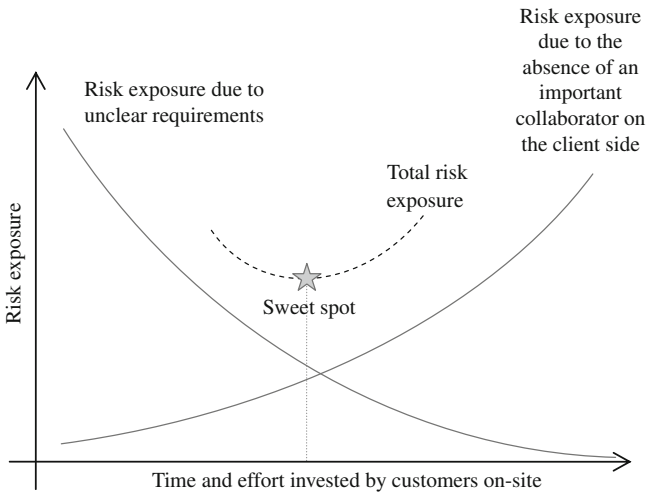


Fig. 6.2 Trade-off between an on-site and off-site customer

The current (2nd) edition of Extreme Programming sees on-site customers (the practice is now called "Real Customer Involvement") as a corollary practice, i.e., as "difficult or dangerous to implement before completing the preliminary work of the primary practices [4]."

In any case, with or without a customer on-site, AgileCoders and BusyClient have to have a shared view on how the project is carried out. If they do not, BusyClient might expect something different from what AgileCoders is delivering. We observed such a situation with a company that was doing frequent releases. After some time we noticed that the client was quite nervous because he interpreted the

frequent releases as an indicator for low quality: in his eyes, the developers could not get things right and had to fix things continuously.

There are several approaches to communicating a strategy to a client. One possibility is a "mission statement": it states the vision and describes the chosen means to achieve it.

The mission statement of St. Michael's Hospital in Ontario states that its vision is "Creating a healthier world, through our culture of caring and discovery" and states the following means to achieve it [29]:

1. providing exemplary physical, emotional, and spiritual care for each of our patients and their families;
2. balancing the continued commitment to the care of the poor and those most in need with the provision of highly specialized services to a broader community;
3. building a work environment where each person is valued and respected and has an opportunity for personal and professional growth;
4. advancing excellence in health services education;
5. fostering a culture of discovery in all of our activities and supporting exemplary health sciences research;
6. strengthening our relationships with universities, colleges, other hospitals, agencies, and our community; and
7. demonstrating social responsibility through the just use of our resources.

The mission statement is easy to understand. Its aim is to be a general guidance for the day-to-day decisions within the organization.

The problem arises if we do not know if the mission is being achieved or not. It is like not knowing where we are on the map, then we do not know where to go to reach our destination.

To understand how good we are in achieving the mission, we need to find ways to measure it. For example, for the point five of the mission statement above, we could look at the "Percent of time dedicated to research." This measurement would tell us how much time employees are able to dedicate to research.

Only through measurements can we objectively (see box below) assess the current situation and compare our performance with the performance of others or with our performance of the past.

Using the "Percent of time dedicated to research" to measure point 5 of the mission statement defines what we specifically mean by it. It shows what we consider important but also what we do not. For example, by not measuring tangible results like patents or papers, we tell that we do not consider them essential.

This example shows that to find the right set of measurements, we need to have a clear understanding of what is causing success and what is preventing it. If we have a wrong perception of the reality, we will measure the wrong thing.

In the city, to measure how much time it will take us to reach some place, it is fine to use the distance in km. On the mountain this is not enough. We have to consider the height difference too; otherwise our estimation will be very imprecise.

This example shows that it can be necessary to collect a set of measurements to have a precise understanding of the situation. On the other hand, we prefer having few measurements to explain a situation than to have many. This preference (in statistics called "parsimony") aims to keep the measurement easy to understand (and easy to extend if needed).

What does "objectively" mean?

The term "objectively" is a word used in everyday's language. Objective is the opposite of subjective. It means that we try to observe some object excluding the influence of us looking at it. This is sometimes difficult or even impossible.

For example, if we take an experienced skier and ask whether some skiing slope is steep, he will probably say: "no." If we ask a beginner, he might be frightened just to think about it.

The two answers are subjective: they depend on who gave the answer. We cannot compare the answers of many skiers, since they are based on evaluations of the terrain that are influenced by their own experience.

To get an objective answer, we need to find a way to measure the steepness of the slope, independently from who is measuring. We need to (a) define a measure of steepness and (b) define how the measurement is obtained, i.e., define a measurement procedure.

The second aspect—to define the measurement procedure—is crucial: only if the measurement can be performed by anybody obtaining the same result can we then speak about an objective measurement.

We could define the steepness in percent as the relationship between the vertical climb and the horizontal distance. We measure how much height a slope gains in relationship to how much horizontal distance it gains. Figure 6.3 shows a slope where (at the point where we measured) the vertical climb is 0.88 m and the horizontal distance is of 2 m.

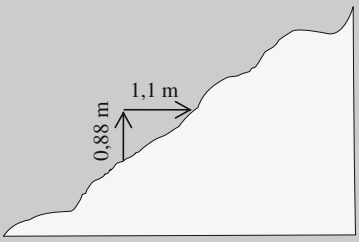


Fig. 6.3 Measuring the steepness of a slope

(continued)

According to our definition, the steepness in percent is calculated as

$$\frac{0.88\,\text{m (vertical gain)}}{1.1\,\text{m (horizontal distance)}} \times 100 = 80\,\%$$

As a measurement procedure we choose to use two yardsticks: one is positioned perpendicular and is used to measure the vertical gain, and one leveled yardstick is used to measure the horizontal distance from the end of the first yardstick back to the slope.

Using the steepness in percent and the agreed measurement procedure, we can objectively say how steep a slope is, whether it is 10 % or around 100 % as the couloir of Fig. 6.4.

Fig. 6.4 Joel couloir, Sella group, Dolomites, Italy: Is it steep or not?

An example of a "mission statement" with measurements is the Balanced Scorecard (already mentioned in Chap. 3). The goal of the Balanced Scorecard is to provide a balanced (all aspects of the company should be considered) view of the performance of the company. The Balanced Scorecard itself, in its entirety, can act as a mission statement since it defines what is important (what is measured, what is considered relevant in the organization) and what is not.

The Balanced Scorecard is structured in perspectives, which are the different views of the organization. The initial set of views proposed by the authors are [17]:

1. **Customer perspective**: measures the ability of the company to provide value to the customers. This perspective includes performance, quality, and service measurements.
2. **Internal business perspective**: measures the ability of the company to adapt the internal processes to satisfy customer needs.
3. **Innovation and learning perspective**: the customer and internal business perspective define what the company considers important for competitive success. For example, the ability of the company to innovate, improve, and learn.
4. **Financial perspective**: measures if the company's strategy, implementation, and execution are contributing to bottom-line improvement.

The Balanced Scorecard helps to get an overall picture of the company.

A problem affects different parts of the company at different times. For example, a customer service that is not able to satisfy customers will be visible looking at the internal business perspective. If customers complain, it will appear in the customer perspective. Finally, if customers switch to the competition, we will see it in the financial perspective.

This means that we can map cause and effect relationships within the Balanced Scorecard [20]. Figure 6.5 shows some of them as arrows between the perspectives.

We now present two ways to communicate a strategy to stakeholders: the mission statement as well as the Balanced Scorecard. These two examples differ in the approach: the Balanced Scorecard is a quantitative approach that collects quantitative evidence to interpret the reality; the mission statement follows a descriptive, qualitative approach (see Chap. 11) to give an overall picture of the elements that characterize the strategy and how they interact.

The way a strategy is described depends also on the type of control we want to exert. In Chap. 4 we discussed behavior controls (e.g., to ensure that employees dedicate a certain amount of time to research) and outcome controls (e.g., to ensure that developers produce a certain amount of code per year). In the same way, a strategy can define the desired behavior and/or the desired results qualitatively or quantitatively.

There is decadelong debate whether the qualitative or quantitative approach is preferable [3]; both have their advantages. Some authors combine qualitative and quantitative approaches, for example, as "exploratory designs" or "explanatory designs" [6]. An exploratory design begins with a primary qualitative phase, then the findings are validated by quantitative results. An explanatory design is characterized by an initial quantitative phase that is followed by a qualitative phase. Usually, the qualitative results serve to explain the quantitative results.

In Lean management we find qualitative and quantitative approaches. The standard worksheet (see Chap. 2) uses a qualitative, descriptive approach. Autonomation uses a quantitative approach to detect a problem; it requires some measurable property to verify its correct value. Because of the importance of autonomation in Lean management, we focus on quantitative ways to define and evaluate the achievement of strategies.

We will use the GQM approach described in the next chapter (similar to the Balanced Scorecard approach, but more general) to quantitatively define the common vision, i.e., what Lean really means for the company. This will alleviate the problem of communicating the goals and methods of Lean to stakeholders and build trust towards those that claim the advantages of Lean.

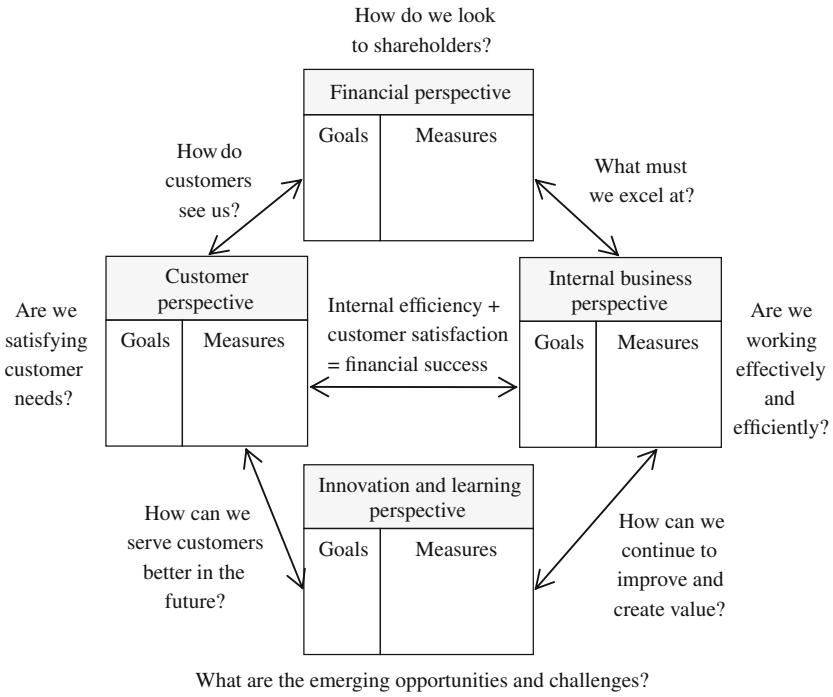


Fig. 6.5 The Balanced Scorecard [17]

6.4 Deprive Gurus of Their Power

We previously stated that Agile methods have been conceived and refined by "gurus." What we criticize is that gurus tell us the "know-how," but not the "know-why." This critique is not completely fair, since gurus not always actually know the "know-why," i.e., the reason why what they preach is working.

Frequently, the gurus were those people that discovered the new method (e.g., Ken Schwaber, Kent Beck). They made the experience that something works and something does not. It is this experience that they are describing in their books. This does not mean that they were able to develop the wisdom why their method works.

If we need to find out how good a certain technology can work for us and cannot find anyone that can tell us, we have to develop the experience ourselves. We need to use the so-called scientific method[1] to systematically find the knowledge we seek. We need to [3] (see also [32]):

1. formulate a problem in form of hypotheses, i.e., tentative explanations;
2. identify what we want to study;
3. apply research methods to obtain data (e.g., observation, survey, experiment);
4. analyze the data; and
5. use the results to confirm or falsify the hypotheses;

Usually the scientific method begins with idea that pops up or somebody promoting a new technology or method to us. "You have to do testing, then you will have software without defects!" might be a claim. If you are a risk-seeking person, you will immediately introduce testing throughout the company. You risk that the advice is wrong and that the defects increase or that other aspects (such as development speed) suffer. If you are a risk-averse person, you follow the scientific method.

According to the scientific method, we need to formulate the research problem first. An initial formulation could be: "Does testing reduce the number of defects?" We will begin to investigate this question, develop test cases for classes, and document the defects that we find for both types of classes: tested and untested.

We then will formulate the hypothesis: "Testing a method reduces the number of defects in that method." Counting the defects that we find and classifying them whether they were found in tested or untested classes is the measurement with which we test our hypothesis.

There is a difference between confirming hypotheses and falsifying them. If we confirm a hypothesis, we do not know if there is some situation in the future that falsifies it. If we falsify a hypothesis, we know it is false. If we are not able to falsify it, we can consider the hypothesis provisionally valid.

[1] The here described scientific method should not be confused with the scientific method promoted by Frederick Winslow Taylor.

In our case the result of our experiment can have three results:

1. we proof that the hypothesis is wrong: we now know that—in our environment and in our experimental setting—testing does not reduce the number of defects;
2. we cannot proof that the hypothesis is wrong: we now know that—in our environment and in our experimental setting—testing can reduce the number of defects; and
3. we cannot proof that the hypothesis is wrong or right (e.g., if the results are random): we cannot say anything; we have to continue investigating. Using the words of the English writer William Cowper (1731–1800): "Absence of proof is not proof of absence."

The example shows that a hypothesis that is formulated vaguely is hard to falsify. It is difficult to proof that testing never reduces the number of defects present in a method. Moreover, the usefulness of a hypothesis that could not be falsified is not high: the statement that testing can help is not that useful. An example of a more specific hypothesis is: "Does the % of code that represents testing code correlate with the number of defects?" Failing to falsify this hypothesis would mean that the more we test, the less defects we have. We could then start to look at the optimal amount of testing, and so on.

The experience that we gain from our experiments should be used to improve our work. Only then the time and effort we invested will pay off. We have to document our findings and refine them as we get more knowledge. To be able to apply it and to get the support from others, we need to communicate our findings in way that others understand it.

If we want to share our experience, we have to package it in a reusable form. Reusable means that the know-how and know-why become evident: others can understand how and why it works. Experience reuse wants to make use of previously gained experience in similar problems to help to solve an actual one.

Being able to use a previously packaged experienced has several advantages [5]:

- **Shorter problem-solving time**: the cumbersome task of designing experiments, formulating hypotheses, collecting data, etc. can be avoided.
- **Improved solution quality**: building on previous experience can reduce the probability of wrong decisions.
- **Less skills are required**: the problem solver needs to have less skills if he can rely on previous experience.

The application of the scientific method in software development—the continuous experimenting, evaluating, and adapting—is the way how software companies innovate and gain competitive advantage. This finding convinced scientists and practitioners to develop process models that embed the steps advised by the scientific method.

Lean Thinking also advocates this method to constantly improve.

An example of such a study would be to compare Kanban and Scrum in a specific context and find out their different effects. Such a study was conducted by Sjøberg et al. [28], and they discovered that in their context, after replacing Scrum with their implementation of the Kanban concept, they were able to reduce the number of bugs and improve productivity.

In Chap. 8 we will look at the Experience Factory, which is one way to perform continuous improvement using the scientific method and builds on the Plan-Do-Study-Act method presented previously.

The relationship between Lean and Agile

We can find Agile ideas within Lean Thinking: "Everyone knows that things do not always go according to plan. But there are people in the world who recklessly try to force a schedule even though they know it may be impossible. They will say 'it is good to follow the schedule' or 'it is a shame to change the plan,' and will do anything to make it work. But as long as we cannot accurately predict the future, our actions should change to suit changing situations [21]."

Here Ono clearly describes Agile practices within the Toyota context. But in the Toyota Production System, Agility is a means to an end, not an end in itself. Also inside Agile Methods there are Lean principles, for example, the tenth principle of the Agile Manifesto: "Simplicity—the art of maximizing the amount of work not done—is essential."

Altogether, these methods use each other to achieve their respective goals:

- Agile Methods aim to achieve Agility, i.e., the ability to adapt to the needs of the stakeholders.
- Lean production aims to achieve efficiency, i.e., the ability to produce what the stakeholders need with the least amount of resources possible.

Both methods, Lean production and Agile methods, focus on being effective: to maximize the value for their stakeholders. However, they have different perspectives. While Agile Methods focus on software development, Lean production is an approach that aims to optimize the entire organization.

To illustrate the different perspectives of Lean and Agile, we look at the entire "socio-technical system" (see Fig. 6.6).

(continued)

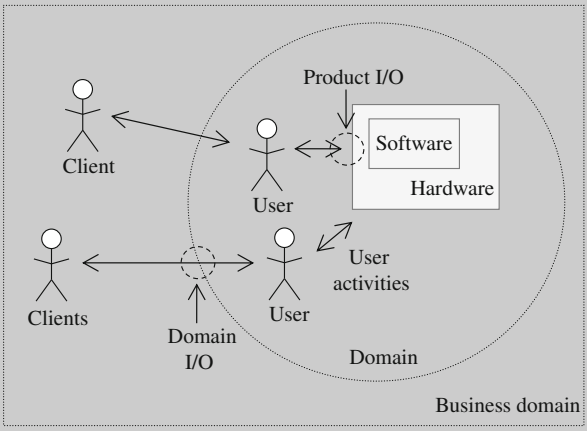


Fig. 6.6 Socio-technical system [18]

This concept looks at IT organizations from three perspectives:

- the product perspective is about the hardware and software of the product;
- the domain perspective is about how users use the product; and
- the business domain perspective is about the business value the users are able to add using the product.

For example, in most restaurants nowadays the waiter uses a device to register the orders. The device is the product. The waiter is the user of the product. The business domain perspective analyzes how well the waiter is able to satisfy the wishes of the guests using the product.

Software is embedded into a socio-technical system, and users interact with hardware and software to solve issues, which help to fulfill their business goals. Therefore, software cannot be seen as a purely technical issue [27].

The difference between Agile and Lean is that they were conceived to work in different perspectives of the socio-technical system.

Agile methods concentrate on the delivery of a product that provides value to the user. The point of view is the one of the developer creating a product for the user. The user knows what is best for him and provides the requirements.

Lean Thinking looks at the entire business domain and seeks the most efficient way to create value for the client of the organization, not the user of the product. This allows to optimize over the entire organization, not only within the activity of software development as Agile methods do.

6.5 Disarm Extremists

In contrast to the gurus of the previous chapter, Agile extremists are the followers of the guru. The extremists we are talking about are risk neutral, optimistic, and idealistic people.

They are willing to accept risk to introduce radical changes. Because of their optimism, risk is not managed, i.e., anticipated, estimated, and minimized using countermeasures, but it is ignored. Problems are addressed as they arise. Their idealism makes them see the whole world as Agile, Lean, etc. Every problem is framed in their "believe system," in their view of the world.

To rise the awareness that a given technology does not always work, we need objective data; otherwise, we and the extremists discuss based on faith.

Unfortunately, the collection of objective data is expensive. The costs to introduce a measurement program (for the first year) can account for 1–2 % of the total engineering or IT effort [9]. In a study Rico and Pressman made in 2004, the complete cost to use a manual measurement program like the Personal Software Process [14] to help produce 10,000 lines was $145,600 [24].

To disarm extremists and confront them with hard data, in Chap. 8 we introduce non-invasive measurement. This term—borrowed from medicine—indicates that the measured object is not altered because of the measurement. In the case of measurement, the term indicates that we adopt an approach in which no time has to be spent for the measurement itself, just for the data analysis and interpretation. This kind of measurement is non-invasive because it does not disturb, i.e., distract those involved in the measurement process.

6.6 Summary

This chapter is an anticipation of what will follow in the following chapters: we will introduce the different components we propose to create what we describe in the preface: a practical implementation of Lean software development, gluing together well-proven tools to provide a way to develop Lean. We want to achieve this through the utilization of goal-oriented, automated measurement for the creation of a Lean organization and the facilitation of Lean software development.

The components we foresee are:

- Agile Software Development, described in Chap. 4,
- Non-invasive measurement, described in Chap. 9,
- GQM$^+$Strategies, described in Chap. 7,
- the Experience Factory, described in Chap. 8, and
- Lean Thinking (together with the practices proposed by Taiichi Ono in his book "The Toyota Production System"), described in Chap. 2.

In Chap. 10 we will see how the different components work together.

Problems

6.1. Tag each software development practice of Mary and Tom Poppendieck's proposal of Lean software development as:

- **value**: if its primary goal is to identify what has value and what has not;
- **knowledge**: if its primary goal is to increase the understanding of what happened, what is happening, and what will happen; and
- **improvement**: if its primary goal is to improve the status quo.

6.2. Imagine you have to develop a Balanced Scorecard for a software development team. Which perspectives would you use? Which goals would you use for each perspective?

References

1. Apache Software Foundation: Apache ant (2013). Online: http://ant.apache.org. Accessed 4 Dec 2013
2. Apache Software Foundation: Apache subversion (2013). Online: http://subversion.apache.org. Accessed 4 Dec 2013
3. Atteslander, P.: Methoden der empirischen Sozialforschung. Studienbuch Series, 10th edn. Walter de Gruyter, Berlin (2003)
4. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Reading (2004)
5. Bergmann, R.: Experience Management: Foundations, Development Methodology, and Internet-Based Applications. Lecture Notes in Computer Science. Lecture Notes in Artificial Intelligence, vol. 2432. Springer, Berlin (2002)
6. Borrego, M., Douglas, E.P., Amelink, C.T.: Quantitative, qualitative, and mixed research methods in engineering education. J. Eng. Educ. **98**(1), 53–66 (2009)
7. CPPUnit Contributors: Cppunit—c++ port of junit (2013). Online: http://sourceforge.net/projects/cppunit. Accessed 4 Dec 2013
8. CruiseControl contributors: Cruisecontrol (2013). Online: http://cruisecontrol.sourceforge.net. Accessed 4 Dec 2013
9. Ebert, C., Dumke, R.: Software Measurement: Establish, Extract, Evaluate, Execute. Springer, Berlin (2007)
10. Fowler, M.: Continuous integration (2006). Online: http://martinfowler.com/articles/continuousIntegration.html. Accessed 4 Dec 2013
11. Free Software Foundation: Gnu make (2013). Online: http://www.gnu.org/software/make. Accessed 4 Dec 2013
12. GIT Contributors: Git (2013). Online: http://git-scm.com. Accessed 4 Dec 2013
13. Hibbs, C., Jewett, S.P., Sullivan, M.: The Art of Lean Software Development: A Practical and Incremental Approach. Theory in Practice. O'Reilly Media, Sebastopol (2009)
14. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Professional, Reading (1996)
15. Jenkins CI Contributors: Jenkins ci (2013). Online: http://jenkins-ci.org. Accessed 4 Dec 2013
16. JUnit Contributors: Junit (2013). Online: http://sourceforge.net/projects/junit. Accessed 4 Dec 2013
17. Kaplan, R.S., Norton, D.: The balanced scorecard: measures that drive performance. Harv. Bus. Rev. **70**(1), 71–79 (1992)

18. Lauesen, S.: Software Requirements: Styles and Techniques. Addison-Wesley, Harlow (2002)
19. Maglyas, A., Nikula, U., Smolander, K.: Lean solutions to software product management problems. IEEE Softw. **29**(5), 40–46 (2012)
20. Martinsons, M., Davison, R., Tse, D.: The balanced scorecard: a foundation for the strategic management of information systems. Decis. Support Syst. **25**(1), 71–88 (1999)
21. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)
22. Oracle: Virtualbox (2013). Online: http://www.virtualbox.org. Accessed 4 Dec 2013
23. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, Upper Saddle River (2006)
24. Rico, D.F.: ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers. J Ross Publishing Series. J. Ross Publishing, Boca Raton (2004)
25. Shalloway, A., Beaver, G., Trott, J.R.: Lean-Agile Software Development: Achieving Enterprise Agility. Lean-Agile Series. Addison-Wesley Professional, Upper Saddle River (2009)
26. Sikuli Contributors: Sikuli script (2013). Online: http://sikuli.org. Accessed 4 Dec 2013
27. Sitter, L.U.D., Hertog, J.F.D., Dankbaar, B.: From complex organizations with simple jobs to simple organizations with complex jobs. Hum. Relations **50**, 497–534 (1997)
28. Sjøberg, D., Johnsen, A., Solberg, J.: Quantifying the effect of using kanban versus scrum: a case study. IEEE Softw. **29**(5), 47–53 (2012)
29. St. Michael's Hospital: St. Michael's Hospital, Mission & Values (2013). Online: http://www.stmichaelshospital.com/about/mission.php. Accessed 4 Dec 2013
30. van Deursen, A.: Customer involvement in extreme programming: Xp2001 workshop report. ACM SIGSOFT Softw. Eng. Notes **26**(6), 70–73 (2001)
31. VMWare: vmware (2013). Online: http://www.vmware.com. Accessed 4 Dec 2013
32. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Computer Science. Springer, Berlin (2012)

Part II
The Pillars of Lean Software Development

This second part illustrates the basic concepts of our implementation of Lean Software Development, besides Agile software development and Lean Thinking: Non-invasive Measurement, the Goal Question Metric and the GQM$^+$Strategies approach, and the Experience Factory.

The reader who knows already how the following methods work can skip these chapters:

- the Goal Question Metric and the GQM$^+$Strategies approach (see Chap. 7),
- the Experience Factory (see Chap. 8), and
- Non-invasive Measurement (see Chap. 9).

These three chapters can also be used independently from this book to learn about the technologies they describe.

Chapter 7
The GQM$^+$Strategies Approach

. . . pergis pugnantia secum
frontibus adversis conponere: non ego avarum
cum veto te, fieri vappam iubeo ac nebulonem:
est inter Tanain quiddam socerumque Viselli:
est modus in rebus, sunt certi denique fines,
quos ultra citraque nequit consistere rectum!

You keep on comparing issues that
are irreconcilable: when I advise you
not to be greedy I do not encourage you to throw away your money!
There is something between Tania and the father in law of Visellius:
there is a always a balanced way to do things, there are limits beyond which nothing can be right!

Horace, Satires, 1, 1, 102–107

Est Meeting Room, 11AM. "Good morning everyone," Uli started without even waiting for J to open the meeting. But J was there and did not object: he understood the level of excitement of the guy, so he simply nodded. "I had two hours to prepare for this talk, and I have to tell you in true honesty what I did. I took a nap and then a good espresso at Phaeacians, so I prepared myself with this device." Uli showed a napkin with a few lines on it. "I knew it was a waste of time!" Ari commented loud, and Helios agreed with his expression. However, a nod from J was enough to let Uli go ahead without any further comment.

"I felt asleep on my desk and I was dreaming at my code, my perfectly reusable classes, my organization. I was revising a package for tour operator, aeaea.com, and I was trying to make it really astonishing, factoring out all possible reusable elements. Suddenly, the code started moving, shaking, vibrating."

"Did you have some cocaine this morning?" Ari asked. A few people laughed, but J nodded again and Uli went ahead.

"From the code a beautiful woman emerged. Not only she was incredibly pretty, but she was really enchanting and had something magic in herself. She started:

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_7

'Stay with me Uli, keep making me better, work on me, do not let me alone, do not deliver me unless you are really 100% sure you could stay forever away from me.' She was luring me, and I could not resist. Euril then came into the room where I was working, and he also stared at her, but... in a few seconds he was transformed into a droid, well a programming droid, but just a droid. I was a bit frightened, but the attraction for the woman was so strong that I kept dealing with her. The time passed, I cannot quantify whether they were seconds, minutes, hours, or even years. Also Sinon and Elp came into the room and they had the same fate as Euril. Still I was devoting all my time to this sorceress, regardless of the resources spent.

Abruptly, Perim entered the room and screamed at me: 'What the heck are you doing Uli? Everyone in the office is becoming a programming droid, enjoys programming, making the code perfect, and ignores that we indeed we need to deliver value to our customer. We do not write code for a aesthetic desire of being better, we do it because we have another, superior goal, to be successful!' Well, only a beautiful woman could have taken me away from the sorceress... "

Hera smiled silently—she knew this; J stroke her with his eyes... He was jealous of any woman working with him.

"Perim went ahead asking me how I was making sure that I was going in the right direction, whether I was really achieving my goals. I had to admit that I was not remembering them properly, but writing good code was always good. 'Still, if you consume all your resources on a tiny portion of the code and you forget what is value for the customer, you are just lost! Now, please, do this, take just a short break from this sorceress.' Saying this Peril sat on my desk and put her body between me and the screen/sorceress. 'Now write down on a piece of paper what you want to achieve, your goals, and then put down some simple, jet objective, questions that you ask yourself to determine whether you are progressing toward them, and, eventually, ways to gather numbers that answer such questions.' I did it and it was immediately clear to me that I was simply lost. I turned my head to the sorceress and, crying, I told her: 'I have to go, sorry. This is all so beautiful, but I cannot afford it any more.' She hugged me for the last time.

'You have found your way, a solid way to keep your navigation progressing, so go. I give you my last advice, though. Go to the land of the dead, I mean, dead projects, and ask their advice. They will be useful.'

She moved her left hand touching softly my face, my eyes, my lips, and my hands; all my co-workers were not any more droids! She disappeared."

7.1 Introduction

We mentioned already that one of the tenets of Lean management is visual control. However, software is invisible. It would be hard to exercise visual control on something that is not visible! A way to make it more visible is to use metrics.

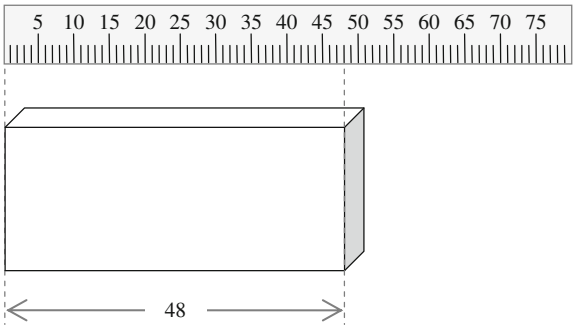


Fig. 7.1 To measure means to compare

Measuring an entity means assigning values to properties of such entity (see Fig. 7.1) [7]. When we say that a box has a weight of 300 g and a width of 48 cm, we mean that:

- There is a box.
- The box has at least two "measurable" properties, weight and width.
- The weight has a value of 300 g.
- The width has a value of 48 cm.

So we can now build our mental, "visual" image of the box even if we have not seen the box.

Likewise if we say that we have a bottle of 1 L of Barolo wine with 14.5 % of alcohol from 1964, we mean that we have a bottle of wine containing wine with the following "measurable" properties:

- The brand is Barolo.[1]
- The amount is 1 L.
- The year of production is 1964.
- The alcoholic content is 14.5 %.

So now we can build again our image of the bottle of Barolo.

Measurement in software development is essential for understanding, controlling, and improving the development process. Software is invisible; therefore, we need a way to make it visible. Through measurement we can do this.

The process of measuring is a process of comparing the interesting attribute of an entity with one or more reference entities. For instance, if I want to measure the width of a box, I need to compare it to the reference unit for width. Saying that the box has a width of 48 cm means that I compared it with the centimeter, the reference

[1]Notice that we consider also the brand as measurable. It is measurable in the sense that we can attach to it a clearly identified value belonging to a set of possible values. It is what is called "nominal" measure by metrologists.

unit for width, and I find that I can put 48 times the centimeter into the width I am measuring.

Therefore, to make measurement possible, it is necessary to define a reference unit. For example, to measure width, the reference unit can be the centimeter. In the year 1793, during the French Revolution, the French Academy of Sciences decided to introduce a more "rational" unit of measure for lengths—the meter. It defined the meter as what they believed to be the ten-millionth of the distance from the Equator to the North Pole through Paris. To make sure that everybody uses the same unit, an iron "prototype meter" was created and kept in a safe place in Paris.

7.2 What Can We Measure?

Generally speaking, we can measure everything that we can observe. Some things can be observed with our senses, and some things—such as the Higgs boson—require the use of expensive equipment like the Large Hadron Collider in Geneva.

We distinguish direct and indirect observations. A direct observation studies the properties of an event at the moment it occurs. Indirect observations study the traces and effects of events after their occurrence and infer their properties.

Software development in a production process—this means that it is an activity that consists of a series of steps taken to produce software. Typically a production process has inputs (resources) and outputs (artifacts).

We can now measure everything that we are able to observe, for example:

- Which activities are executed in a typical project?
- How much time do we spend for testing?
- How much time do we spend to write documentation?
- How much code is reused?
- How much code do we produce per week?

Some other things cannot be observed directly; we have to observe them indirectly, for example:

- How many mistakes do we make per week?
- What is the quality of our code?

These two examples are indirect measurements since we cannot observe the mistake or the quality directly. Mistake and quality are concepts that need to be further defined so that they can be inferred from other aspects that we can observe.

For example, to a mistake, we have to define how to identify a mistake once we have the source code in front of us. After defining the rule of what constitutes a mistake, we are able to interpret what we observe to identify interesting concepts such as mistakes, delays, copy/pasted code, etc.

The same applies to quality: we have to define what we mean by quality, then we can define what to observe and how to interpret the observation to infer the quality. For example, if the quality in our context is how fast an application is able to process some operations, we can measure quality in two steps:

1. we observe the number of operations that the application is able to process per minute and
2. we define which range of values we consider acceptable, average, and good.

In this way, we followed what we stated above: we observe values of properties (operations per minute) and we interpreted the results and inferred an observation.

Figure 7.2 gives some examples of the four types of measurement we looked at: direct or indirect observations and with or without tool support.

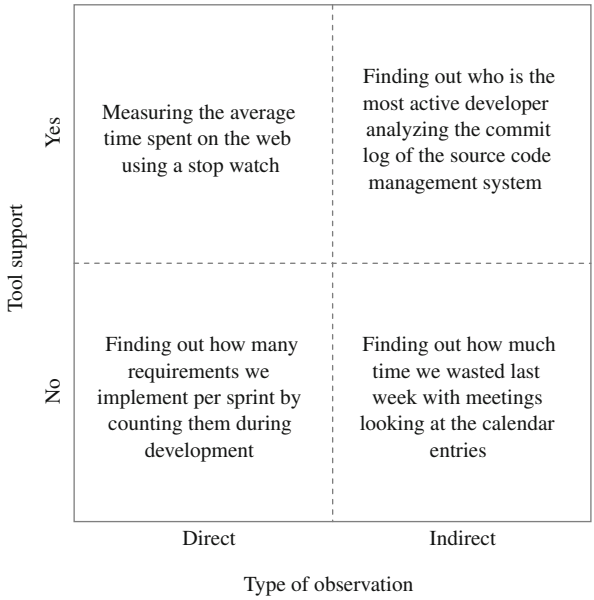


Fig. 7.2 Measurement examples

7.3 What Should We Measure?

Not everything that can be measured should be measured. Measurement costs, so the decision what to measure has to be based on the expected outcome of the measurement.

Altogether, the decision on what to measure depends on what we want to find out and on what can help us to understand the problem or to discover a new opportunity. We need a framework that helps us to define what and how we want to measure.

Let us assume we measured the quality of our code as in the example of the previous section and we obtained a value of 12. Let us further assume that we defined that if a value is bigger than 20 and smaller than 50, we consider it acceptable. Our value is definitely too low and we decide to increase that value.

Now, that we want to understand how to increase that result, we need to perform another measurement. We could measure the execution time of every component of our application to discover which component is responsible for the largest amount of time. That component is the one with the highest potential to improve speed, so we want to focus on that component first.

This example shows that we need to understand what we want before we decide what we want to measure and how we want to measure it. For this reason, the Goal Question Metric (GQM) model proposed by Basili and Weiss is so important.

The GQM is used to define (a) what data have to be collected and (b) how the data are interpreted.

A measurement framework is defined on three levels [2]:

1. **Conceptual** level (goal): defines what and why we study. What is studied is the "object of study," the specific products, processes, and resources. Why something is studied identifies the reason, the different aspect taken into consideration, the considered point of views, and the environment.
2. **Operational** level (question): here there are the questions that define (a) what parts of the object of study are relevant and (b) what properties of such parts are used to characterize the assessment or achievement of a related goal. Such properties are often called the focus of the study. Altogether, the questions specify which specific aspect of the object of study is observed to understand if the goal is achieved or not. Questions are measurable entities that establish a link between the object of study and the focus. For example, if the object of study is a car and the focus is its environmental impact, a question could be: "How high are the carbon dioxide emissions of the car?".
3. **Quantitative** level (metric): defines the set of software measurements needed to answer the questions in an objective (quantitative) way.

It is important not to confuse the focus with the point of view. The focus is the part of the object of study that is studied. It is an objective view on the object of study. The point of view describes who is measuring and represents a subjective view on the measurement goal.

An Analogy from the Software Testing Domain

Software testing aims to understand if a program fulfills the requirements or not. A specific type of testing is unit testing. Unit testing evaluates a piece of source code and assesses if it works as intended. To define a unit test, we need to define:

(continued)

1. the source code to test (line 6 in Listing 7.1);
2. the initial state, i.e., the values of all variables that the tested source code takes as an input (line 7 in Listing 7.1);
3. how to execute the tested source code (line 11 in Listing 7.1);
4. which part of the output we are interested in (line 15 in Listing 7.1); and
5. the value we expect to obtain as a result (line 15 in Listing 7.1).

The test consists in performing steps 1–5 and evaluating if the actual output corresponds to the expected output.

```
1 // Prepare the environment
2 Class.forName(``org.postgresql.Driver'');
3 Connection connection = DriverManager.getConnection(↘
  connectionString);
4
5 // Set the initial state
6 SomeSuperCoolClass s = new SomeSuperCoolClass(connection);
7 s.setParameter(4);
8
9 // SomeSuperCoolClass.performOperation() is the
10 // tested source code and is executed here
11 SomeResult r = s.performOperation();
12
13 // SomeResult.getLength() is the aspect of the result
14 // we are interested in. The expected value is 5.
15 Assert.assertEquals(5, r.getLength());
```

Listing 7.1 A JUnit test case

A unit test can be seen as an analogy to the questions defined within a GQM. A question refers to the object of study and inquires something about the focus. Likewise, a unit test case refers to the tested source code and inquires something about its execution.

The GQM defines the measurement model as a hierarchy of goals, questions, and measurements (see Fig. 7.3). This hierarchy details what is measured, how it is interpreted, and to which answers it leads.

The GQM in Fig. 7.4 evaluates the taste of a glass of Barolo, an Italian red wine. In this example, the taste is evaluated using three criteria: the sweetness, the aroma, and the flavor of the wine. To measure the sweetness of the wine, it is possible to use an electronic oscillating U-tube meter (an objective measurement); to measure the remaining two aspects, aroma and flavor, we use the personal evaluation of our waiter, an expert in the field. The opinion of the waiter is not that objective, but it is also not as subjective as asking a beginner.

The definition of measurement goals is critical to the successful application of the GQM approach. As mentioned, every measurement goal has to be described stating the purpose of the measurement (what and why it is measured), the perspective

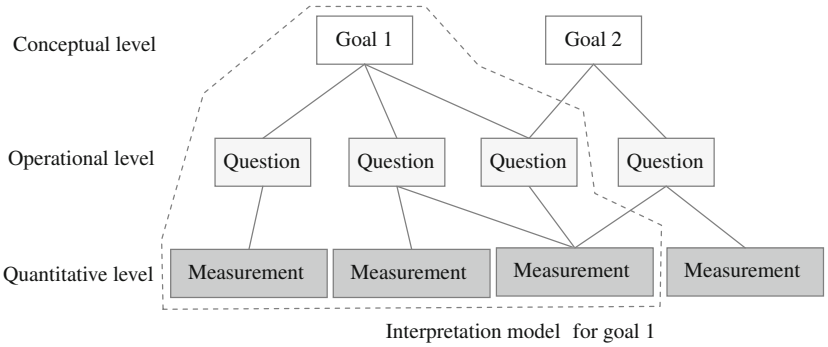


Fig. 7.3 GQM measurement model

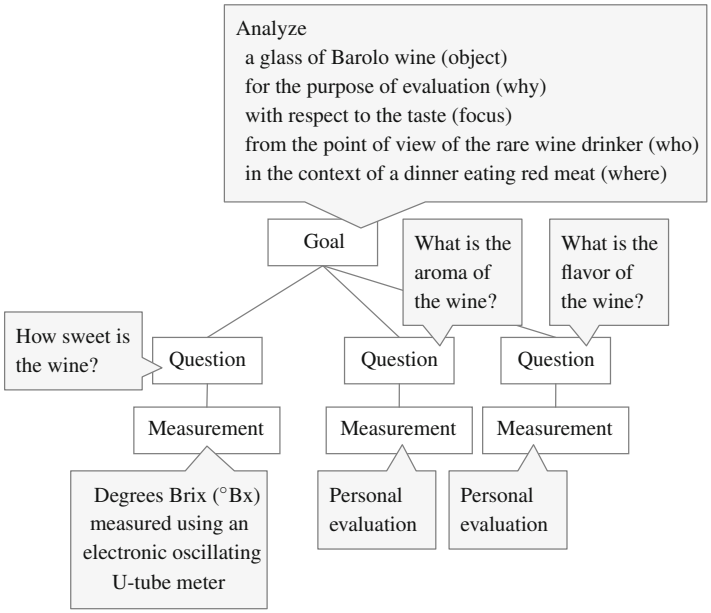


Fig. 7.4 GQM model to evaluate the taste of a glass of Barolo wine

(what specifically is observed, the focus, and from which point of view the observation is made), and the environment (in which context the measurement takes place).

To ease the definition of measurement goals, the GQM supplies goal templates (see Fig. 7.5).

The explicit formulation of the purpose, the perspective, and the environment is used to understand which data are needed to fulfill the measurement goal and to understand how to interpret the collected data.

Purpose

Analyze: *a glass of Barolo wine*
(objects: process, products, resources)

for the purpose of: *evaluation*
(why: to characterize, evaluate, predict, motivate, improve)

Perspective

with respect to: *the taste*
(focus: cost, correctness, changes, reliability, ...)

from the point of view of: *a rare wine drinker*
(stakeholder: user, customer, manager, developer, ...)

Environment

in the following context: *a dinner eating meat*
(context factors influencing the measurement)

Fig. 7.5 A GQM goal template [1]

Figure 7.6 depicts that, starting from the object of study (the big cube), we define the aspect of study (the small cube) and the point of view on it.

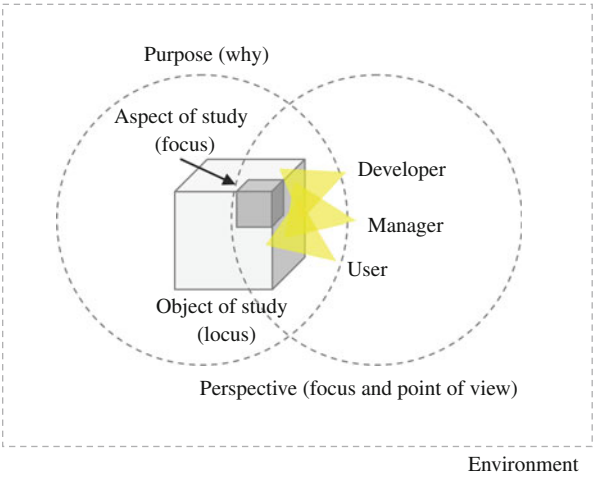


Fig. 7.6 The definition of a measurement goal

Defining the "right" questions is not always trivial. Basili helps us in this respect and classifies how such questions can be posed.

As said in the definition of the measurement goals, questions are used to characterize the goal in a quantifiable way and measurements to describe the data

that will be used to answer the questions. Basili et al. [2] classify GQM questions into three groups:

1. questions that characterize the object of study with respect to the overall goal, e.g.:
   - Is Barolo considered a superb, good, or miserable wine?
   - What is the average price of a bottle of Barolo?
2. questions that characterize relevant attributes of the object of study with respect to the focus, e.g.:
   - What is the aroma of the wine (e.g., spicy, smoky, oak, etc.)?
   - What is the characteristic of the wine (e.g., tannin, rich, complex, etc.)?
   - What is the sensation of the wine (e.g., sparkling, acid, crisp, etc.)?
   - What is the flavor of the wine (e.g., plum, lemon, berry, etc.)?
3. and questions that evaluate relevant characteristics of the object of study with respect to the focus, e.g.:
   - Is the taste satisfactory from the viewpoint of a rare wine drinker?
   - Does the taste match well with the meat?

Once we have the questions, we are not done. There could be multiple measurements that can be used to answer the same question. As we have seen before, the sweetness of wine can be measured using an electronic oscillating U-tube meter, but it could be also evaluated using a personal evaluation.

The selection of measurements to answer the developed questions depend on different factors, e.g., the amount and quality of data that is already available, the cost-benefit ratio of performing a specific measurement, the level of precision needed, etc.

We now consider an example; see Fig. 7.7. This figure shows a GQM model that consists of one goal, two questions, and five measurements. In this example we want to shorten the time needed to process change requests by a software development team.

7.4 Applying the GQM Step-By-Step

Developing a fully fledged GQM is not always easy. In this section we present the approach proposed by Park et al. [11] to simplify such task.

They propose the following sequence of steps:

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.

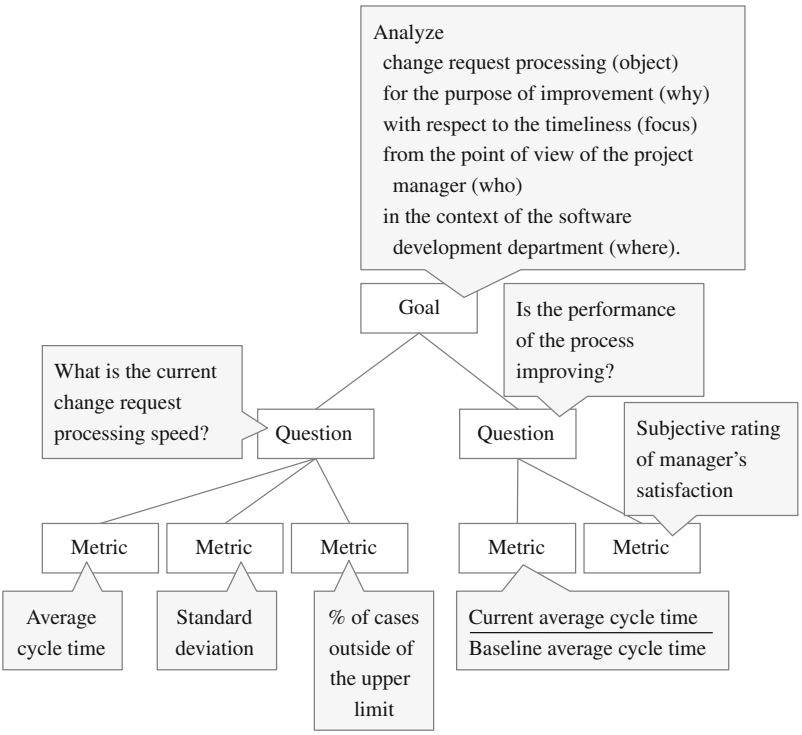


Fig. 7.7 A GQM example [1]

6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

The first step identifies the organizational goals that are used as a starting point.

The measurement goals are derived from the organizational goals to ensure that only relevant goals are studied. The organizational goals need to be operationalized; that means concrete measurable goals have to be derived from them to the level where specific measures can be formulated.

For example, the business goal "Improve customer satisfaction" needs to be operationalized repeating steps 1–4 above to obtain one or more measurement goals that are concrete enough to allow an analysis through the collection of measurements (see Figs. 7.8 and 7.9).

The second step identifies what is needed to understand, assess, predict, or improve to achieve the stated business goals. The business goal of before, "improve customer satisfaction," needs to be refined stating the factors on which customer

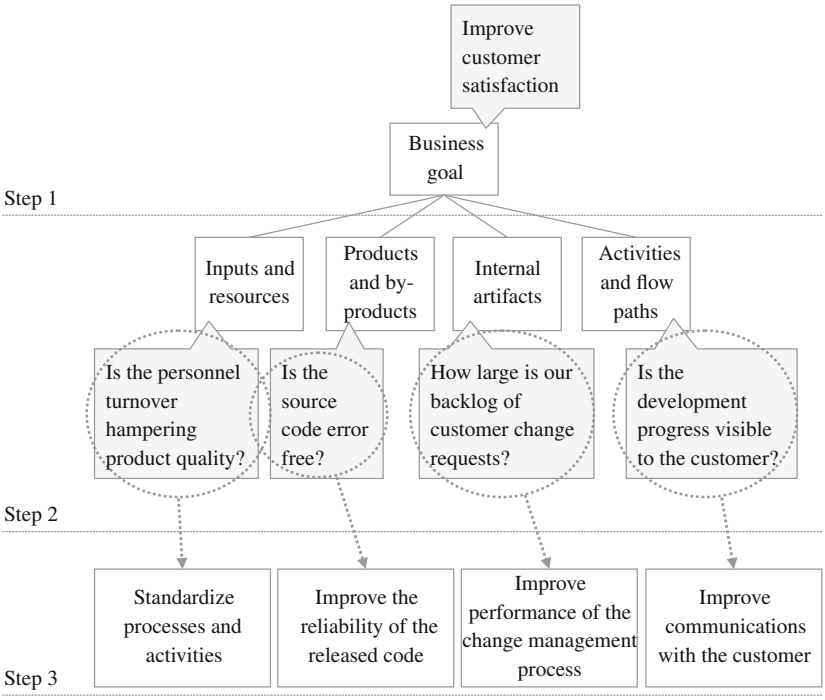


Fig. 7.8 Goal oriented measurement example, the steps 1–3 [11]

satisfaction depends, what the ideal states of these factors are, and how the current state of these factors can be evaluated.

Park et al. [11] propose these factors:

- inputs and resources,
- products and by-products,
- internal artifacts such as inventory and work in process, and
- activities and flow paths.

This step derives factors from the organizational goals that have to be observed and controlled. The third step groups the aspects identified in step two into subgoals that state goals for activities that support the business goals.

Starting from the subgoals obtained from step 3, step 4 focuses on how to reach them identifying the entities and attributes that influence their outcome. Once these entities and attributes are identified, step 5 creates GQM measurement goals as seen above.

We can distinguish active and passive measurement goals [11]:

- active measurement goals aim at controlling or causing changes and
- passive measurement goals aim at learning or understanding.

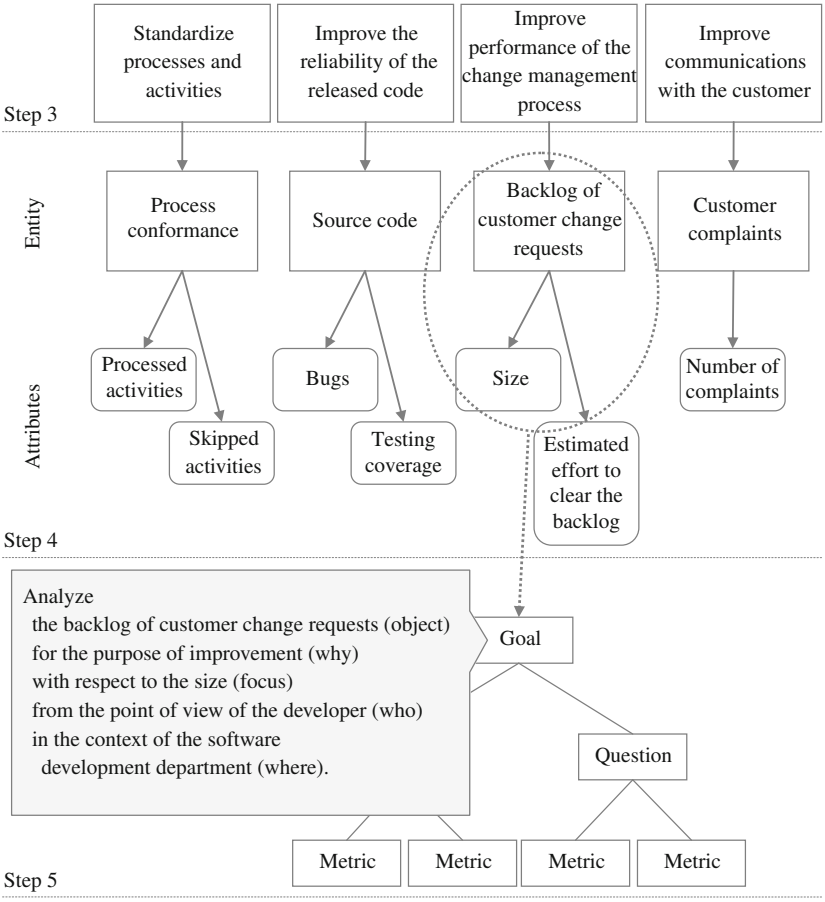


Fig. 7.9 Goal-oriented measurement example, steps 4–5 [11]

Examples for the terminology used within the formal goal formulation of active measurement goals are "to evaluate" or "to improve," and "to characterize" or "to predict" for passive goals.

Steps 6–8 correspond to the remaining definition of the GQM: step 6 identifies quantifiable questions and the related indicators that will be used to achieve the measurement goals, step 7 identifies the data elements that will be collected to construct the indicators that help answer the questions, and step 8 defines the measures to be used to make these definitions operational (see Figs. 7.8 and 7.9).

One novelty introduced by the approach of Park et al. is that they suggest to add an intermediate step between the question definition and the measurements definition phase, i.e., a phase in which indicators are explicitly identified. Moreover, Park et al. use a definition of an indicator that is a more narrow one than we used, i.e., "a picture or display of the kind one would like to have to help answer the question [11]."

Because of the use of (graphical) indicators as an intermediate step, they call this approach the GQ(I)M approach.

In Chap. 11 we provide a detailed example on how to develop a GQM model.

7.5 Alignment

We already mentioned in Sect. 7.3 that not everything should be measured, but only what brings value. The GQM approach brings one step further as it helps us to obtain valid measurements (that means that we really measure what we want to measure; see Chap. 9). The next step is to measure what brings value to the organization, that is, what helps to fulfill the organizational goals. The approach described in Sect. 7.4 already implicitly linked measurement with the organizational goals as it started identifying business goals.

Organizational goals are hierarchically structured [8]. Every activity within an organization is a means to an end, a part of the organizational strategy to achieve an organizational goal. In the same way a software engineering process that is part of a set of processes within a larger organization has to be aligned to the organizational goals to contribute to their achievement [3, 6, 12]. This results in a goal hierarchy such as the example in Fig. 7.10, where each upper goal (e.g., a certain revenue as financial goal) justifies the next goal (e.g., to increase sales), and so on.

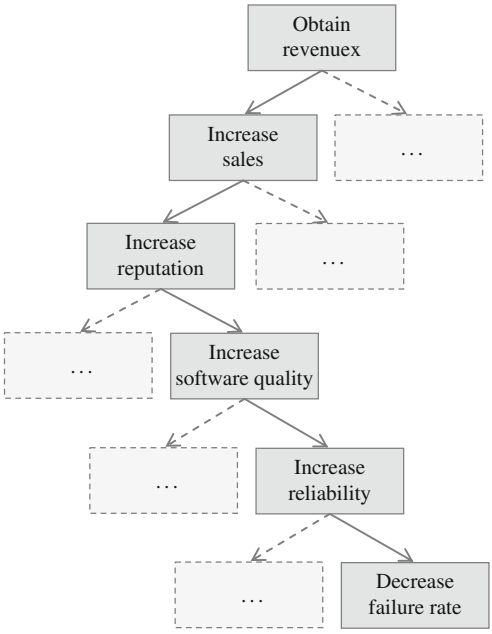


Fig. 7.10 A goal hierarchy

In the example of Fig. 7.10, the company decides to obtain a given financial goal through an increase of sales. There would be alternative options shown by the dashed box. The financial goal should be furthermore obtained through an increase of the reputation of the company. This increased reputation is thought to be obtained through an increase in the software quality. Moreover, in this example the software department decides that this will be achieved through an increase of the reliability, i.e., a decrease of the failure rate.

Such a goal hierarchy fulfills two functions: first, it describes abstract goals in more detail; for example, "increase reliability" better describes what is meant by "increase software quality." Second, it describes means-end relationships; for example, to increase the reputation of a company is the means to increase sales [13].

To support organizational decisions with data, we have to link measurement goals to organizational goals [5]. The balanced scorecard approach described in Chap. 3 does not define a methodology to perform this link. At this point we see how the GQM$^+$Strategies approach [5] extends the GQM model: it considers the goal hierarchy motivated by the organizational strategy and creates a measurement model that links business goals to measurement goals.

Figure 7.11 illustrates the concept: every element of the goal hierarchy is linked to a GQM model that measures the achievement of the business goal at that level.

The decision of **how** a given goal should be achieved, what means are used to obtain it, is defined by the strategy. A strategy is accepted by the organization as the right approach to achieve its goals, given the environmental constraints and

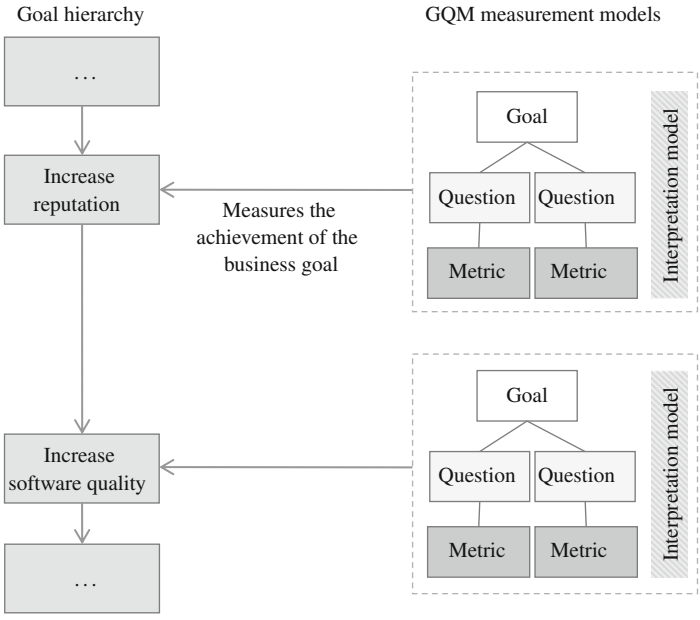


Fig. 7.11 Measurement along the goal hierarchy (adapted from [4])

risks [10]. The strategy explicitly states the steps that have to be achieved to obtain the desired goal.

To understand which software strategies are the most effective to achieve a given organizational goal, it is important to link the organizational goals to the derived goals of the underlying processes by explicitly stating which strategy was adopted. This will help to develop the necessary knowledge to select and tailor the right strategy in the future.

GQM+Strategies provides a goal template to support the measurement of organizational goals. It consists of the following elements [5]:

- **Object**: the object of study (see Chap. 7);
- **Focus**: the aspect of study (see Chap. 7);
- **Magnitude**: the desired magnitude of improvement;
- **Time frame**: the time frame for achieving the goal;
- **Organizational scope**: the scope of responsibility for achieving the goal;
- **Constraints**: constraints or conflicting goals; and
- **Relationships**: relationships to other goals.

Table 7.1 shows an example of an organizational goal.

Table 7.1 GQM+Strategies business goal example

Element	Example
Object	Software product line A
Focus	Net income
Magnitude	8 % per year
Time frame	Every year
Organizational scope	Development team 2
Constraints	Maintain current product price
Relations	Business goal “Increase market share”

The selection of a software development strategy that is aligned to the organizational goals is often difficult [3] since:

- there is a lack of understanding how software contributes to the organizational goals;
- it is difficult to understand which software strategies are the most effective to achieve a given business goal; and
- it is difficult to understand how established software strategies have to be tailored considering influencing factors (e.g., requirements, regulations).

As a consequence, the strategy adopted within the organization and the underlying software development processes often mismatch [6]. Additionally, software engineering projects are frequently faced with unrealistic goals [3]. If the contribu-

tion of software engineering activities to the achievement of the organizational goals is not clear, it is not possible to define an optimal business strategy that involves all activities within an organization.

In summary, a software development organization has to:

1. develop an initial strategy that addresses the organizational goals and keep track of the reasons why a certain strategy was thought to be effective or not (plan);
2. execute the actions foreseen in the strategy (do);
3. monitor the software development process to see if its outcome contributes to the fulfillment of the organizational goals (check);
4. tailor the strategy to improve its effectiveness and start over again (act).

Different methods exist to develop strategies; a common approach is the SWOT (Strength, Weaknesses, Opportunities, and Threats) analysis [9]. This analysis assesses a given goal considering separately:

- **Strengths**: organizational aspects that support the achievement of the objective;
- **Weaknesses**: organizational aspects that inhibit the achievement of the objective;
- **Opportunities**: external conditions that support the achievement of the objective;
- **Threats**: external conditions that inhibit the achievement of the objective.

These groups of organizational attributes generate four groups of questions which can be used to generate strategies and derive organizational goals from them by trying to use our strengths and overcome our weaknesses to exploit opportunities and to reduce our vulnerabilities against threats (see Fig. 7.12).

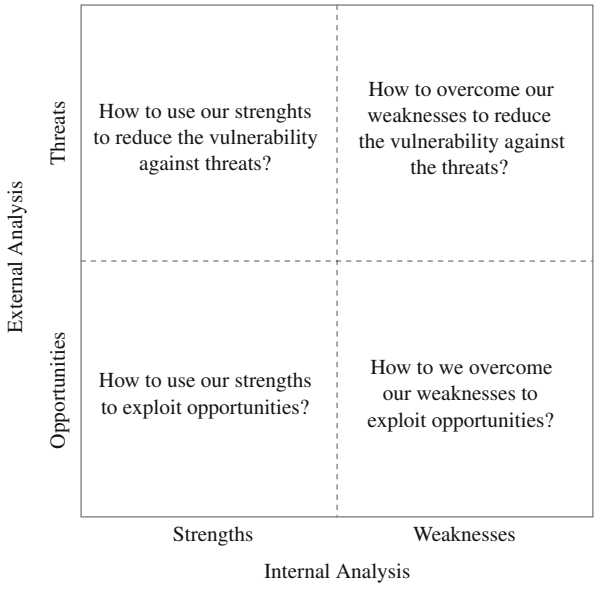


Fig. 7.12 SWOT analysis [9]

So far we described the GQM+Strategies approach in a simplified way. Figure 7.11 shows that GQM measurement models (which in GQM+Strategies are called "GQM Graph") directly measure the achievement of an organizational goal. In fact, for each organizational goal, GQM+Strategies also considers [5]:

- **Context factors**: characteristics of an organization or its environment that have an impact on measurement.
- **Assumptions**: Expected, uncertain characteristics of the organization or its environment that have an impact on measurement.
- **Strategy**: A plan to achieve the organizational goal.

A GQM+Strategies model (called GQM+Strategies Grid) including all key elements is depicted in Fig. 7.13.

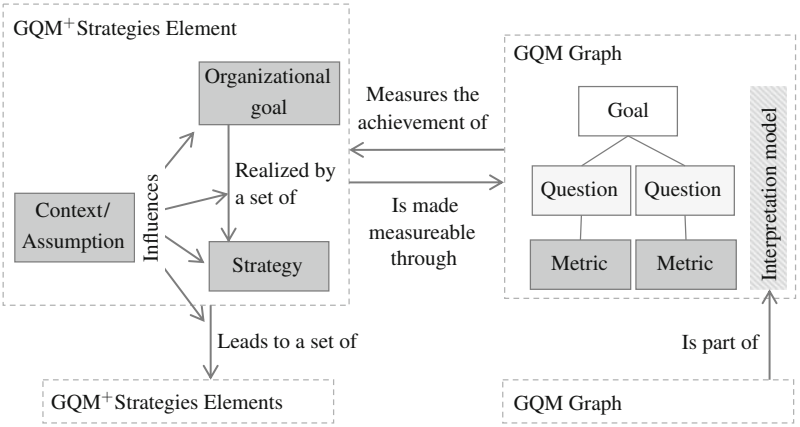


Fig. 7.13 GQM+Strategies Grid (adapted from [5])

In GQM+Strategies, the goal hierarchy is modeled using a hierarchy of GQM+-Strategies Elements. For each element we define an organizational goal, context factors, assumptions, and one or more strategies to achieve the goal.

7.6 Summary

The GQM+Strategies approach helps to define what we measure, why we measure it, and how the collected data are interpreted. It shows what is considered important—it can be used to communicate this to other collaborators. It also states which evidence is collected (measurements) to understand if the company goals were achieved—so it states what finally counts.

Therefore, the GQM+Strategies should be constantly updated so that it reflects the goals of the team, the department, and the organization [5]. It represents the

shared view of how value should be evaluated and therefore created. Basili et al. recommend to update a GQM$^+$Strategies grid using an approach based on the Quality Improvement Paradigm (see Chap. 8) [5].

This chapter gave an overview of the Goal Question Metrics and the GQM$^+$ Strategies approach. We needed this to understand how to put measurement into a learning context, which will be discussed in the next chapter.

Problems

7.1. Imagine you want to evaluate how readable the source code of some program is. Define a GQM model to describe what and why you would measure.

7.2. The development in company M occurs according to the following schema: when a new project is started, a developer takes an old project that is the most similar to the new requirements and makes a copy and starts implementing the required modifications. To improve this process and to help the company to adopt a component-based approach, we want to understand which pieces of code are the best candidates for future components and which variability points they have.

To understand which pieces of code are the best candidates for future components and which variability points they have, the company M defines the following goal:

- **Object of study**: the source code of two projects committed in our source code repository;
- **Purpose**: evaluate;
- **Focus**: repeated source code;
- **Stakeholder**: Programmer;
- **Context factors**: the Java programming language.

Some of the questions connected to this goal are: "How much source code is repeated from one project to another?" Which graphical indicator would you like to have to answer these questions?

References

1. Basili, V.R.: The experience factory and its relationship to other improvement paradigms. In: Sommerville, I., Paul, M. (eds.) Proceedings of the European Software Engineering Conference (ESEC). Lecture Notes in Computer Science, vol. 717. Springer, Berlin (1993)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1. Wiley, New York (1994)
3. Basili, V.R., Heidrich, J., Lindvall, M., Munch, J., Regardie, M., Trendowicz, A.: Gqm$^+$Strategies — aligning business strategies with software measurement. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Computer Society, Madrid (2007)

4. Basili, V.R., Lindvall, M., Regardie, M., Seaman, C., Heidrich, J., Münch, J., Rombach, H.D., Trendowicz, A.: Linking software development and business strategy through measurement. IEEE Comput. **43**(4), 57–65 (2010)
5. Basili, V.R., Trendowicz, A., Kowalczyk, M., Heidrich, J., Seaman, C., Münch, J., Rombach, D.: Aligning Organizations Through Measurement: The GQM+Strategies Approach. The Fraunhofer IESE Series on Software and Systems Engineering. Springer International Publishing, Berlin (2014)
6. Becker, S.A., Bostelman, M.L.: Aligning strategic and project measurement systems. IEEE Softw. **16**(3), 46–51 (1999)
7. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, London (1998)
8. Heinen, E.: Das Zielsystem der Unternehmung. Die Betriebswirtschaft in Forschung und Praxis. Gabler, Wiesbaden (1966)
9. Kotler, P.: Marketing Management. The Prentice Hall International Series in Marketing. Prentice Hall of India (2000)
10. Object Management Group: Business motivation model (bmm) (2007). Online: http://www.omg.org/spec/BMM,. Accessed 4 Dec 2013
11. Park, R.E., Goethert, W.G., Florac, W.A.: Goal-driven software measurement — a guidebook. Technical Report CMU/SEI-96-HB-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1996)
12. Porter, M.E.: Competitive Advantage: Creating and Sustaining Superior Performance. The Free Press, New York (1985)
13. Schwarz, R.: Controlling-Systeme: eine Einführung in Grundlagen, Komponenten und Methoden des Controlling. Die Wirtschaftswissenschaften. Gabler, Wiesbaden (2002)

Chapter 8
The Experience Factory

Dicebat Bernardus Carnotensis nos esse quasi nanos gigantium humeris insidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea.

(Bernard of Chartes used to say, that it is as if we are dwarfs who seat on the shoulders of giants, so that we can see more things and farer away than them; but this is not caused by a better eye-vision or a taller body but because we are taken up and raised by their magnitude.)

John of Salisbury, Metalogicon 3, 4.

Needless to say, the people in the room started being more interested.

"I woke up because the phone rang. It was the usual, well-known consultant offering a new tool called Cimmerians to do post-mortem analysis—what a coincidence! I felt asleep again. Now I was exploring a tool with really a poor user interface, but the tool had a nice chatting module. Through this tool I was immediately contacted by Agam.

The story was sad. Agam had just been fired by his contractor, who is also his ex-wife, Clitem. Nothing personal, just that they needed to restructure the business, and Agam is a great guy, but he is an external. They thought that the knowledge of an external is not so valuable, because it is likely that he will move from contract to contract to whoever pays him better—and this is understandable and obvious. But the net result for the company is that the company then does not grow, because the experience is lost. Agam tried to convince Clitem that he had only one contract left open and that it is for his brother, so he could not reasonably give it up. She did not change her mind and he was out.

Then Tires contacted me. He had just heard of Agam. Tires is an interesting guy. He was never particularly successful, on the contrary, he made much less money than he deserved. But he had brilliant ideas. He anticipated most of the future trends of software development: object orientation, patterns, etc. However, he was not trusted, perhaps because he was never able to market himself properly: instead of presenting himself as the innovator or the troubleshooter, he looked most of the

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_8

time as an arrogant and very opinionated jerk with unconventional and ungrounded ideas on how to develop software.

He told me: 'You know, your capital is your mind, and your mind is crafted by all the experience you have. So, capitalize your experience! I know that someone already suggested you to define clearly your goal, your questions, your measurements. This is good. But you can do more! Trace down your experience, save it in a way that also other people will be able to take advantage of it and try also to take advantage of other people experience. But do not do it in an ad-hoc way. Do it systematically, and ask other to comment on it. Do not be afraid to share successes and failures, to make wrong comments, to say something that is not right or perfect, share!' Then, he started cursing at Clitem and he left."

8.1 Introduction

The single most valuable asset of a software company is the knowledge and the experience of its people: knowledge and experience in developing the code, writing documents, managing projects, and so on. Therefore, it is of paramount importance for a software company to capitalize on such knowledge and experience and to create an infrastructure where this knowledge and experience are stored, preserved, and made available for use even when new projects come, new collaborators are hired, or present collaborators move away.

The model of the Experience Factory proposed by Basili [4, 5] addresses such need. However, before proceeding with the exploration of the concepts of the Experience Factory, we focus our attention again to the so-called PDSA.

8.2 Why Plan-Do-Study-Act Does Not Work in Software Engineering

The Experience Factory is an approach to collect and reuse past experiences, based on the Plan-Do-Study-Act paradigm, PDSA [10]. As mentioned in Chap. 2, the goal of the PDSA is to create within an organization a flow of constant quality improvement based on four steps (see Fig. 8.1):

- **plan** the activities we need to perform so that we achieve the desired improvement and their expected outcome;
- **do** execute the plan;
- **study** the outcome, measure it, and compare it with the expected outcome; understand the reasons for the difference between reality and expectations;
- **act** according to the results, that is, institutionalize the planned activities or adjust them.

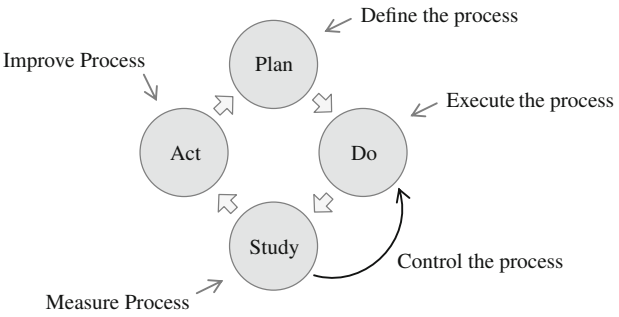


Fig. 8.1 The Plan-Do-Study-Act cycle

A consequence of the PDSA is that the process in a company becomes more controlled, that is, its results are more predictable. Predictable means that it is possible to state, at least approximately, the probability that the observed results will fall within given boundaries [20].

A more predictable process results in:

- a more reliable forecast of the effort required to complete the associated tasks,
- a more efficient allocation of resources, e.g., less inventories are needed since one can trust that a certain output will be available for the following processes, and
- an easier and earlier identification of the sources of errors: since the output and the characteristics of the output are known, it will be—compared to a process with unpredictable output—easier to detect anomalies and their sources.

A method to find out that a process result is unexpected is statistical process control using control charts. Control charts (see Fig. 8.2) visualize the collected data and its expected variability and support spotting anomalies, where it is likely that something happened that should be further investigated.

The control chart in Fig. 8.2 shows the control chart of a "quality characteristic," i.e., an aspect the development team wants to observe and to "keep under control" over time.

The center line marks the expected value of the visualized characteristic, the upper and lower control limits mark the limits beyond those a point will be considered abnormal. The upper and lower warning limits mark limits after which countermeasures are suggested.

Unfortunately, it turned out to be difficult to apply statistical process control in software development, even though it is a desirable goal. In fact, the task of the "study" step of PDSA is to find out how to adapt the process so that it becomes controlled.

Control charts as well as the PDSA were developed for replicable manufacturing processes; therefore, they aim to optimize production processes that have limited degrees of variability and can expose solid sets of standards. Within such production

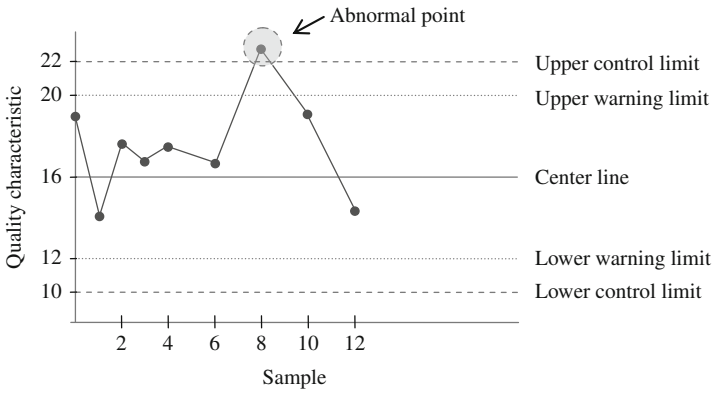


Fig. 8.2 Statistical control chart [12]

processes, it is possible to collect statistically representative datasets based on the continuous repetition of the same process and to develop quantitative models of the process that lead to diagrams like the one in Fig. 8.2.

Software development is different [3]: its production has a very high degree of variability. Altogether, the software discipline is evolutionary and experimental [15], not as repetitive as a production process. It is often simply not possible to find the same activity executed several times, so it is hard to collect data to analyze it statistically [3].

Moreover, it is largely human based so that software development is less predictable than production, model building is more difficult, the models are less accurate, and we have to be cautious in the application of the models [3]. These problems caused the development of solutions that are adapted to software engineering such as the Experience Factory.

8.3 The Experience Factory

The Experience Factory is an instrument to systematically collect, analyze, generate, and reuse experience in software development [4]. Remember, experience is meant as "valuable, stored, specific knowledge that was acquired in previous problem solving situations [6]."

Experience management is a special kind of knowledge management. It is restricted to managing experience, which is a special kind of knowledge. Experience is a valuable, stored, specific knowledge that was acquired in a problem-solving situation. Thereby, experience is in contrast to general knowledge, which has a broader scope and which is discovered inductively from large bodies of experience, e.g., through scientific research. This restriction has several important implications for the knowledge management activities [6]:

- The Experience Factory approach defines a framework to continuously improve the quality of the software development process. This is accomplished through the systematic collection, creation, and reuse of experience.
- Experience management can be considered a variant of knowledge management [8, 18] that manages data and information and transforms it into knowledge and wisdom to increase the understanding of the underlying principles of the analyzed phenomena [1, 9].

We will use the concept of the Experience Factory as a way to implement continuous learning within our approach. This experience can, once collected, be used in two ways:

- as a controlling instrument: to compare expected outcomes (those coming from the accumulated experience) with realized outcomes; this will allow to understand the status and the progress of development, to detect variations in the process that are not caused by common reasons, and to understand which countermeasures will bring the process back into a controlled state [12] and
- as a process improvement instrument; by reusing accumulated experience to improve (e.g., avoid repeating the same errors, standardize work, distribute knowledge, etc.) the software development process.

The Experience Factory uses the Plan-Do-Study-Act approach, adapted to the software development domain, called the Quality Improvement Paradigm (QIP) [2].

The QIP—as the PDSA approach—is a process model. It prescribes a set of activities that have to be executed in the prescribed order to achieve the expected results. It is an iterative model, i.e., it is based on the idea that a set of steps is executed repeatedly until all the work is done. The steps are organized using two cycles: the first is for long-term learning, and the second is executed during the project phase and provides early feedback while the project is carried out (see Fig. 8.3).

The main cycle consists of the following six steps [3, 5]:

1. characterize,
2. set goals,
3. choose processes,
4. execute,
5. analyze, and
6. package.

8.3.1 Work Distribution

Within the PDSA cycle (see Fig. 8.1), the goals of the first two steps ("planning and doing") and the last two steps ("studying and acting") have different objectives: "planning and doing" aim to solve a problem; "studying and acting" aim to capture and reapply the gained knowledge to do better next time.

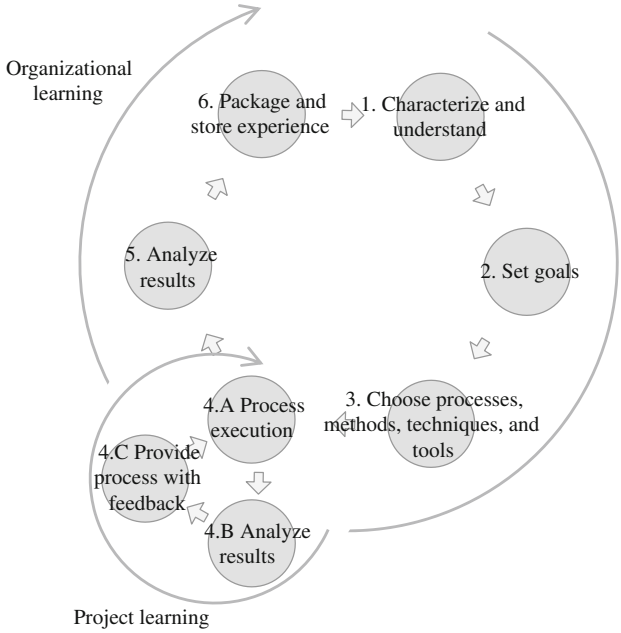


Fig. 8.3 The Quality Improvement Paradigm cycle [5]

The required skills for "planning and doing" are to develop ways to analyze requirements, and design and implement a solution, and to validate and verify the adopted solution with the requirements of the client. On the other hand, "checking and acting" require the ability to study current working practices; to measure how software is currently developed; to analyze, abstract, and generalize current development methods; to develop new methods; to improve current methods; to tailor existing methods to new problems; to formalize findings; etc (see Fig. 8.4).

For this reason, Basili et al. [4] recommend to introduce the Experience Factory as a separate organizational unit that specializes in the "checking and acting" tasks to support the other organizational unit responsible for "planning and doing."

The Experience Factory is the organizational unit responsible for the "experience packaging," i.e., to use past experiences to develop improved ways to produce software and to package this new knowledge in a way that it will be used in the next Plan-Do-Study-Act cycle. In fact, the steps "study" and "act" deserve a particular attention within software engineering since they have to be carried out differently due to the particular nature of software (see Fig. 8.5).

The packaging and storing of experience are not and end by itself; it is the responsibility of the entire organization to facilitate the reuse in all the activities of the software development.

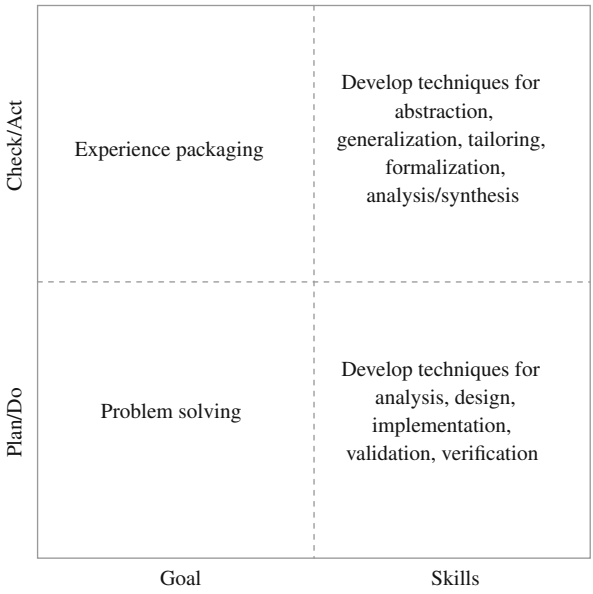


Fig. 8.4 Goals and skills needed within the project organization and within the Experience Factory [4]

8.4 The QIP Step-by-Step

We now describe every step proposed by the QIP in detail.

The characterize and understand step (**step 1** in Fig. 8.3) analyzes the current project with respect to different characteristics to find a similar set of projects. This activity helps to identify a context in which it will be possible to reuse experiences.

The goal is to evaluate and compare the new project in respect to similar past ones and use past projects for prediction. Different aspects can be used to categorize a project, for example, the classification proposed by [11] (Fig. 8.5):

- people factors (e.g., expertise, organization, problem experience, etc.) [14],
- process factors (e.g., life cycle model, methods, tools, etc.),
- product factors (e.g., deliverables, required quality, etc.), and
- resource factors (e.g., target and development machines, time, budget, legacy software, etc.).

"Set goals" (**step 2** in Fig. 8.3) identifies the goals of the process execution. It formalizes all aspects that are important and therefore should be observed during the project development.

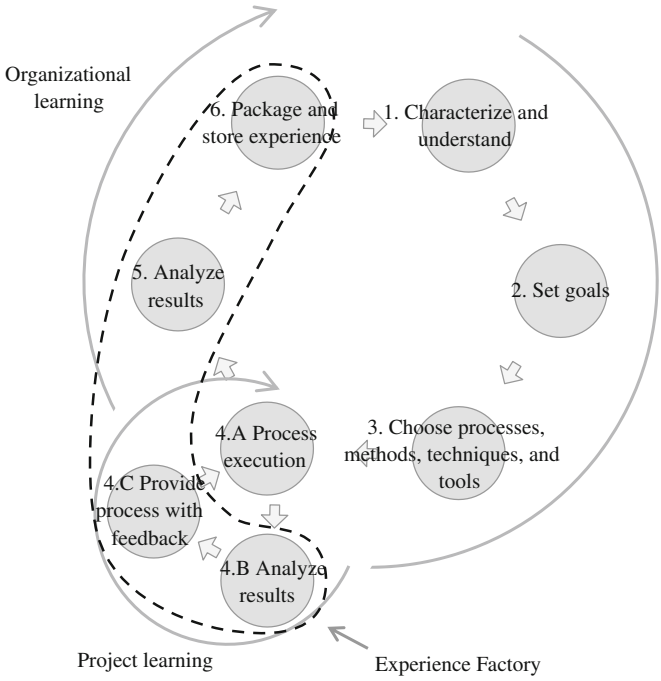


Fig. 8.5 Responsibilities of the Experience Factory [4]

The goals reflect what on organizational level is important. If they are measurable, their outcome can be compared to the actual results. A goal such as "we want to have more beautiful user interfaces" without specifying what "more beautiful" means is not measurable and therefore useless. After the development no one could objectively tell how much "more beautiful" the user interface now is. Moreover, we cannot assess how good we were in achieving our goal or how much we are distant from our goal.

To ensure measurable goals, the QIP adopts the Goal Question Metric Method (see Chap. 7).

After settings the goals, the appropriate means to achieve them have to be selected (**step 3** in Fig. 8.3). With means we intend to choose process models, methods, tools, etc. The word "appropriate" expresses that we aim to select and adjust means that are effective to obtain the stated goals.

We also want to improve our ability in choosing the appropriate means to achieve our goals. It is necessary that what we choose is measurable so that we can observe how much an action contributes to the achievement of a goal.

The "process execution" step (**step 4.A** in Fig. 8.3) represents the actual software development activity prepared in the three steps before. Ideally, during this step we collect data that can be used as input for the next step (analyze results).

Based on the stated goals, the collected data during software development, and the reflections about ongoing and past projects, we revise the plan made in step 3 and compare the expected outcomes with the actual ones (**step 4.B** in Fig. 8.3) and provide the process with feedback (**step 4.C** in Fig. 8.3). This phase aims to improve our understanding of the results of our choices, to answer questions like "what is the impact of the environmental factor A on the effectiveness of technique B?" or "how does the development time change when technique X is used?"

To profit from the gained experience of the previous step, future projects should be able to access and use it. The best way how this is done depends on the type of experience we are dealing with; it could be process models, lessons learned, checklists, source code, components, patterns, etc. The gained experience (**step 5** in Fig. 8.3) has to be packaged and stored so that it can be used for future projects (**step 6** in Fig. 8.3).

The six steps of the QIP can be related to the four steps of the Plan-Do-Study-Act paradigm (see Fig. 8.6).

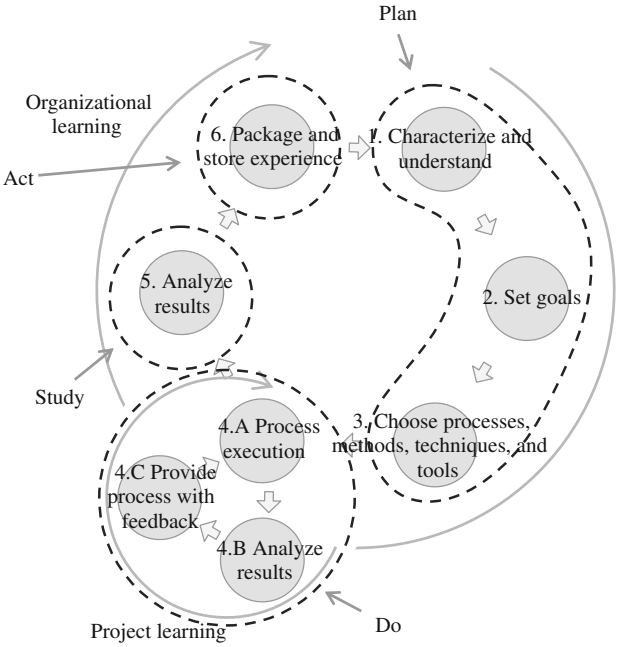


Fig. 8.6 The Quality Improvement Paradigm cycle [4]

Reflection, Retrospective, and Post-Mortem Analysis

One of the principles of the Agile Manifesto says: "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

This principle to reflect on the past and to learn from it can be implemented on different organizational levels. On the project level, the project team meets regularly during the project to discuss how to become more effective in that project. On the organizational level, all involved collaborators meet after a project and discuss about the positive and negative aspects of past projects and look on what to improve in future projects and how to improve the professional development of every project participant [7].

The terms used for this learning activity vary. "Reflection" and "retrospectives" are used for both organizational and project learning; "post-mortem analysis" is used for organizational learning. Unfortunately, much of the knowledge obtained in a project remains unnoticed and is never shared between individuals or teams since reflection does not occur frequently in practice [21].

The type of collected, analyzed, packaged, and reused experience depends on the organizational goals. A possible taxonomy is [5]:

- product packages (programs, architectures, designs),
- tool packages (constructive and analytic tools),
- process packages (process models, methods),
- relationship packages (cost and defect models, resource models, etc.),
- management packages (guidelines, decision support models), and
- data packages (defined and validated data, standardized data, etc.).

Table 8.1 gives some examples of previous implementations of the Experience Factory together with the experience and the format of the experience that was managed.

The feedback given within the project learning loop (**step 4.C** in Fig. 8.3) compares the actual project results with past experiences and provides immediate feedback such as "your error density is now 10 percent higher than usual" or "when you inspect more than five pages of code at a time, your performance goes down [18]."

Table 8.1 Examples of experiences and experience packages of previous implementations of the Experience Factory.

Source	Experience	Format of the experience packages
[4]	The impact that available technologies have on the software process, which technologies are beneficial to the environment and, how the technologies must be refined to best match the process with the environment.	Training Programs, standards, policies, guidebooks, pre-configured environments
[5]	The selection and the tailoring of quality-focused software engineering technology for specific processes.	Lessons-learned, "process packages": process definitions (stating the performed activities, the adopted technologies, the involved people, used resources) together with training and consulting material.
[13]	The accuracy of effort estimation practices, improvements in quality due to formal reviews on requirements documents, the impact of a clear acceptance process.	Intermediate results of previous projects, checklists, templates
[18]	Competence in software development and acquisition.	Descriptions of software development processes (e.g., software inspections, software risk management, and requirements engineering), adapted for the intended users, with links to training materials, experiences, checklists, frequently asked questions, and expert emails. The content and the format of the packaged experience is tailored to a concrete anticipated usage situation.
[19]	Experience on training, assessment, and guidance of the following technical areas: programming languages, Experience Factory, inspection, design, testing, risk analysis, process, and other.	Document packages linked to the project where they have been produced and to the consultant that created it.

8.5 The Role of Measurement

The suggested way to study a problem in the Toyota Production System is to go directly to the workers and see what they do [16]. This activity was also popularized as "Management by wandering around" [17]. The idea is to observe firsthand which problems collaborators face and how they cope with them, help them to improve, get rid of obstacles, etc.

In fact, the last step, the step "act," is crucial for organizational learning: only if the obtained knowledge on how to improve has an impact on the current working practices, the costs of studying the problem can be justified. Following the Toyota philosophy, also organizational learning should be driven by the provided value for the customer.

The step "study" should be performed with a clear goal, and this goal should be of value for the customer so that the outcome, the input for the step "act," improves the production process in the right direction, driven by the provided value for the customer.

The above described way to study a problem contained the words "go" and "see." These activities might be trivial for physical goods, e.g., it is immediately visible if a door of a car is missing, but because of the invisible nature of software and its complexity, to "go and see" if the software is correct is not possible. We need some indirect way to "go and see" if the software development is currently producing value and also to see if modifications to the development process are achieving the desired effects.

Such an indirect way is software measurement. With software measurement, we collect data about the employed resources to develop software, about the ongoing development activity, and about the output of the project, i.e., the software itself.

It might be tempting to start to collect every aspect of the software development process with the idea to analyze it later if something can be found to improve it. This would follow the push philosophy described in Chap. 2: it would mean to create a "data inventory," to analyze the data, to generate improvement ideas, and to "push" them to the software development teams after.

While this approach is useful in research or for an exploratory study, within a Lean environment, the "pull" philosophy advises to drive the data collection by the expected increase in the value for the customer and to avoid unnecessary costs.

To follow a pull philosophy means to start from an expected improvement of the value for the customer, for example, avoiding to waste certain resources, or the shortening of a necessary, but costly process. The knowledge needed to implement an improvement constitutes the measurement goal, i.e., it justifies why data should be collected at all. This immediately confronts the foreseen data collection costs with the expected benefits. If the expected benefits outweigh the expected data collection costs, the measurement goal "pulls" the required data from the data collection processes. In other words, the expected improvement drives the data collection needs, i.e., which data are collected and how data are collected.

The Goal Question Metric approach discussed in Chap. 7 is used within the Experience Factory to implement the "pull" paradigm, i.e., the collected data are justified by the goals and the value of achieving these goals for the customer.

8.6 Summary

This chapter presented the Experience Factory, an approach to organize the collection, organization, and reuse of experience to support organizational learning. It builds on the QIP and on the Goal Question Metric Approach. Since software is invisible, we need to use measurement to collect experience. The next chapter presents an efficient way to collect measurements: non-invasive measurement.

Problems

8.1. What type of wisdom (in the sense of "know-why," see Chap. 5) would you manage in an Experience Factory to support Lean Thinking? Distinguish between organizational learning and project learning.

8.2. Assume you are responsible to introduce Lean software development in a company. You start doing an SWOT analysis of the team, thinking about the objective of introducing Lean software development. This is the result:

- Strengths:
  - The team consists of many young members, willing to learn something new.
  - The level of creativity is high.
  - The average technical skills of the team is very high.
- Weaknesses:
  - To follow a defined software development process is perceived as limiting. For some, the end justifies the means; that means they sometimes use hacks to implement a requirement instead of maintaining the readability and structure of the code.
  - It is hard to plan the work for a project; the planning accuracy is low.
- Opportunities:
  - The market for tailored software is growing.
  - The complexity of the problems customers have is growing; therefore, their requirements are unclear at the beginning. An approach that enhances flexibility helps to strengthen the position on the market.
- Threats:
  - There is a high pressure from the competition to innovate.
  - There is a high pressure from the competition to lower prices.
  - Some team members think that there is no time for trying out another new software development hype.

Answer the following questions:

1. How would you use the strengths of the team to reduce the vulnerability against threats?
2. How would you use the strengths of the team to exploit opportunities?
3. How would you overcome the weaknesses of the team to reduce the vulnerability against the threats?
4. How would you overcome the weaknesses of the team to exploit opportunities?

References

1. Ackoff, R.L.: From data to wisdom. J. Appl. Syst. Anal. **16**, 3–9 (1989)
2. Basili, V.R.: Quantitative evaluation of software methodology. Technical Report TR-1519, Department of Computer Science, University of Maryland, College Park (1985)
3. Basili, V.R.: The experience factory and its relationship to other improvement paradigms. In: Sommerville, I., Paul, M. (eds.) Proceedings of the European Software Engineering Conference (ESEC). Lecture Notes in Computer Science, vol. 717. Springer, Berlin (1993)
4. Basili, V.R., Caldiera, G., McGarry, F., Pajerski, R., Page, G., Waligora, S.: The software engineering laboratory: an operational software experience factory. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, Melbourne (1992)
5. Basili, V.R., Caldiera, G., Rombach, H.D.: The experience factory. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1. Wiley, New York (1994)
6. Bergmann, R.: Experience Management: Foundations, Development Methodology, and Internet-Based Applications. Lecture Notes in Computer Science, Lecture Notes in Artificial Intelligence, vol. 2432. Springer, Berlin (2002)
7. Birk, A., Dingsøyr, T., Stålhane, T.: Postmortem: Never leave a project without it. IEEE Softw. **19**(3), 43–45 (2002)
8. Davenport, T.H., Probst, G.J. (eds.): Knowledge Management Case Book: Siemens Best Practices, 2nd edn. Wiley, New York (2002)
9. Davenport, T.H., Prusak, L.: Working Knowledge: How Organizations Manage What They Know. Harvard Business Review Press, Boston (1997)
10. Deming, W.E.: Quality, productivity, and competitive position. Massachusetts Institute of Technology, Centre for Advanced Engineering Study (MIT-CAES), Cambridge (1982)
11. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, London (1998)
12. Florac, W.A., Carleton, A.D.: Measuring the Software Process: Statistical Process Control for Software Process Improvement. Addison-Wesley Professional, Reading (1999)
13. Houdek, F., Schneider, K., Wieser, E.: Establishing experience factories at daimler-benz: an experience report. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society, Kyoto (1998)
14. Janes, A., Sillitti, A., Succi, G.: Non-invasive software process data collection for expert identification. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE). Knowledge Systems Institute, San Francisco (2008)
15. Naur, P., Randell, B. (eds.): Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO (1969)
16. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)
17. Peters, T.J., Jr., Waterman, R.H.: In Search of Excellence. Harper & Row, New York (1982)

18. Schneider, K., von Hunnius, J.P., Basili, V.R.: Experience in implementing a learning software organization. IEEE Softw. **19**(3), 46–49 (2002)
19. Seaman, C.B., Mendonça, M.G., Basili, V.R., Kim, Y.M.: User interface evaluation and empirically-based evolution of a prototype experience management tool. IEEE Trans. Softw. Eng. **29**(9), 838–850 (2003)
20. Shewhart, W.A.: Economic control of quality of manufactures product. D. Van Nostrand Company (1931). Online: https://ia601607.us.archive.org/8/items/bstj9-2-364/bstj9-2-364.pdf. Accessed 4 Dec 2013
21. Williams, T.: Identifying the hard lessons from projects—easily. Int. J. Proj. Manag. **22**(4), 273–279 (2004)

Chapter 9
Non-invasive Measurement

Se vuoi che una cosa sia fatta, fai come il podestà di Buie, che ordinava e poi faceva da solo.
(If you want that something gets done, do as the podestà of Buie, who used to give orders and then to do the tasks himself.)

The grandma of one of the authors.

Ahead went Uli: "While I was still looking at the back of Tires, the phone rang, it was Poli.

Poli is the son of Posis, a very wealthy investor in water drilling endeavors, and because of such a father, he became the CIO of a software company for which we do contracted work, as some of you might already know. I had to supply software for him, and, believe me, it was hard. He was really a concrete head—the messages passed to him very slowly and most of the time distorted.

Apparently, in my dream I was (still?) supplying him software. He cursed at me: 'I am in charge of the company and I need your software. You know, I know how to make things working. I promise you, if you do not complete the software on time I will let the whole world know about your incompetence. I will send a mail to the whole world containing my frank opinion on you. I know you can sue me later, but I will let the lawyers of my father to handle this. And, I do not want to give up with the contract, I need your software now and it is too late to replace you, I cannot replace you!'

The work on this software was a pure nightmare (well a nightmare inside a nightmare), since he could never get firm on any single idea and he could not even acknowledge this; every time we tried to sit down with him asking him to check the requirements we wrote, he told us that he paid us so he did not want to spend any additional time or resources for us: 'guys, when I ask for an espresso, I do not explain the waiter how to roast the coffee beans, ok? So, do your job, the one for which you are paid!' There was no point in trying to explain him that his involvement in setting the requirements that were used for him was essential. No way, still he could not do it!

I knew he could keep his threaten. Elp informed me that a friend of him working for another supplied of Poli heard that he was already getting ready for the massive spam. He also told me the message to put in the spam: '<package name> package

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_9

was screwed up by Uli, of FSS—he works so well!' He was also paying some illegal hackers to create a virus to disseminate this message in billion machines.

What could I do? Breaking the contract was not an option—he told me that in such a case, no matter what would have been the consequences for him, the message would have been divulged.

It was not possible to fight him directly, nor to force him to do something that he did not want. He was very stubborn. So, the only way to cope with him was to accept that he was going to send the message around and to try to embody in his own desires what I wanted him to do. So, I had to get him to say what I wanted.

In word I needed to find a way so that, while Poli was doing what he wanted, what his nature lead him to do, he was also doing what I wanted—that is to neutralize any possible bad publicity against FSS and me.

I had an idea. 'Poli, I called him on the phone, I want to ship you right now a new version of the software with some of the improvements you wanted and one addition!' 'Finally Uli, finally you could do it. Good. And what is the addition?' 'I changed the name. You are right, what we do is something really trivial, so I decided to call the package no. I wanted to acknowledge your help and your guidance, and make sure that everyone in the world will have very clear in her or his mind your role. So our work will be on nothing and you will get all the glory.' 'Thanks you Uli, I think I deserve this, after spending so much time in coaching you!' I closed the phone, I changed the name of the package into "no" and I shipped it to Poli.

In some twenty minutes my phone rang. 'Uli, it was Poli, do you think I am so dumb not to understand that you just lured me with the change of name of package and you did not do anything else? I am not stupid! I am a smart manager! Now you will pay the consequences of your actions!'

Poli implemented his threat: the world was flooded of emails against me and FSS saying: 'no package was screwed up by Uli, of FSS—he works so well!"'

9.1 Introduction

Lean Thinking is based on the idea to collect data to find out how to control and improve the process. Following the same line of thought, to improve software development processes, we need to rely on measurement [14]. The gained insight can be used by the management and by collaborators to understand the value delivered to stakeholders, to deliver what is really needed, and to lower the risk of software projects.

Understanding cause and effect relationships allows to improve the alignment of inputs to outputs and thereby create a clear "line of sight" to desired results. This helps to increase the productivity of software production [14]. Moreover, coding quality, defect prevention, and effort prediction accuracy improve when developers measure their personal development process and in this way better understand how they work [25].

What Is Good Data?

This box is useful for people who do not have (yet) a knowledge of the base concepts of (software) measurement, such as measurement scales and the representational condition, and can be skipped by the other.

There is an evergreen principle that lasted more than the law of Newton and the principle of relativity of Einstein: "garbage in, garbage out." It is necessary to obtain good data to be able to obtain useful results. Now, what does "good" data mean?

Collecting data, we collect information about the real world. In other words, we describe interesting aspects of the real world using numbers (or more general, symbols). These symbols should have the following desirable properties [20]: the mapping from real-world objects to symbols follows the representation condition of measurement, and the collected data are correct, accurate, precise, and objective.

These desirable properties will be explained below starting from two concepts: well definition and the scales.

Well defined means that our data originates from a measurement system that maps real-world attributes **unambiguously** to the numbers or symbols our data consists of. As in Fig. 9.1, the height of a building is mapped to a number—50 and 110 in this example.

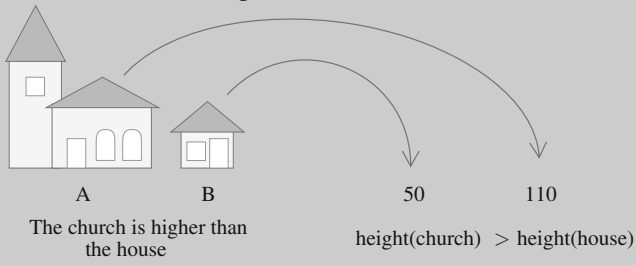


Fig. 9.1 Mapping the height of a building to a number

In many cases this mapping will be from real-world aspects to numbers, but also other symbols are possible, as in Fig. 9.2, the hands show whether a house fits our requirements or not.

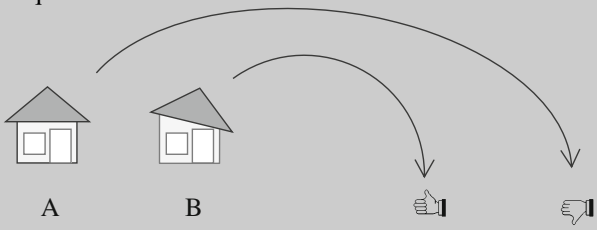


Fig. 9.2 Assigning non-numerical symbols

(continued)

In fact, measurement distinguishes different classes of how to assign symbols to real-world aspects: nominal, ordinal, interval, and ratio scales [53, 54].

Which possibilities do we have to "organize" the assigned symbols? Do we use symbols like the hands above or do we use numbers? Which advantages do we have in choosing between them? These questions will be answered below.

We say that symbols are assigned using a **nominal** scale if we create categories (such as the two hands of Fig. 9.2) and assign our real-world entities to these categories. Examples of nominal scales are gender (male/female), country of residence (Italy, Germany, Austria, etc.), and so on. This assignment has to be homomorph, i.e., that if objects are the same in the real world, they are also assigned to the same category (and also the opposite, that they are in different categories if they are not the same in the real world).

Symbols that are part of a nominal scale can be only distinguished from one another—such as the hands of Fig. 9.2: they represent different outcomes, but the upward pointing thumb is not necessarily "greater" or "better" than the downward pointing thumb (it even sounds strange to say this).

In a nominal scale, the concept of a "middle" element, a most representative element, or average is implemented by the mode (i.e., the value on the abscissa of the maximum value in the probability density function [57]).

If we can rank the categories of symbols so that we can say that something is higher, larger, smaller, etc. we say that we have an **ordinal** scale. An ordinal scale has all the properties of a nominal scale, and it allows ranking. An example of an ordinal scale is the grades given in school: "excellent" > "very good" > "good" > "satisfactory" > "sufficient" > "insufficient."

Figure 9.3 shows an example where the architecture of buildings was evaluated using a Likert scale. The Likert scale, invented by the psychologist Rensis Likert, consists of a set of options that indicate the level of agreement to a given statement, for example:

- strongly disagree,
- disagree,
- neither agree nor disagree (undecided),
- agree, or
- strongly agree.

(continued)

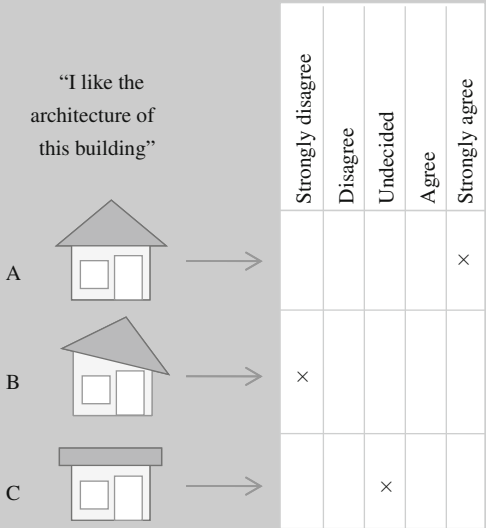


Fig. 9.3 Measuring the level of agreement to a statement

This scale is an example of an ordinal scale. What ordinal scales do not tell is the amount of difference between the categories. Is the difference from "insufficient" to "sufficient" the same as from "good" to "very good"? Ordinal scales does not represent this information, but this is the case for the so-called **interval** scales.

In an ordinal scale, the average is also represented by the median (i.e., the value at which the cumulative distribution function equals 0.5 [57]).

Interval scales define the differences between categories. An example is a set of dates. Dates are on an interval scale—we can tell that the difference between January 20th and January 30th is 10 days. We can also (as the ordinal scale) rank dates and (as the nominal scale) distinguish different entities of dates. So adding and subtracting works within an interval scale, for example, April 22nd + 5 days = April 27nd. What does not work is multiplication and division. It does not make sense to divide April 1st through 3rd (also this sounds quite strange), such as to multiply August 15th by 3.

When we say "it does not make sense," we mean that the operation of multiplying and dividing is not defined for the interval scale. Clearly, we can perform the division, but the result of such division does not make any sense.

We define a **ratio** scale, a scale where the mutual proportion between the measurements makes sense. A ratio scale must have a "natural" zero element, that is, an element representing the absence of the property being measured. Table 9.1 contains some examples of ratio scales.

(continued)

Table 9.1 Natural zero element for different measurements

Property	Natural zero element
Temperature in kelvin	Absolute zero
Duration in seconds	No duration (instantly)
Weight in kilograms	No weight
Price in euros	Free
Speed in kilometers per second	No speed, standstill
Distance in kilometers	No distance

As said, ratio scales are like interval scales, just that they have a natural zero element and in this way define how to interpret multiplication and division: dividing €100 by 10 means to interpret its result as (the distance of €100 to its natural zero element) divided 10.

In a ratio scale the average is represented also by the mean.

Figure 9.4 summarizes what we said so far about scales: nominal scales allow to count the occurrences of mapped entities to our categories (A, B, and C in the figure). Ordinal scales add the possibility of ranking the categories, interval scales allow to distinguish the distance between the categories, and finally ratio scales add a natural zero element.

Let us go back to the list of desirable properties that data should have above.

Validity is given when the measurements really represent what we want to measure. Assume we want to measure the intelligence of our cat, so we pick a standard IQ test and ask our cat to fill it out. Probably Kitty, our cat, will score quite badly on the test. Nevertheless, the conclusion that it has no intelligence at all is incorrect (see Fig. 9.5).

As said before, measurement is about assigning numbers or symbols to things we want to describe. The conclusion of Fig. 9.5 was wrong because the rules applied to do the assignment did not maintain the empirical relationships of the real world when translating them to numbers or symbols.

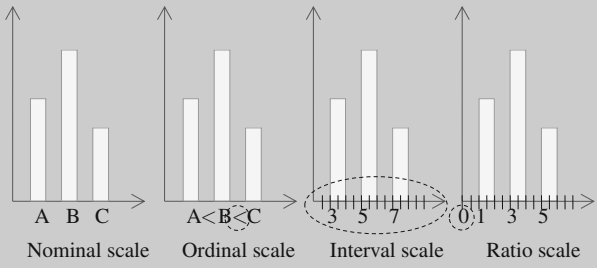


Fig. 9.4 From the nominal to the ratio scale

(continued)

This violated the **representation condition of measurement**. This condition is fulfilled if the measurement maps aspects of the real world into symbols in such a way that the empirical relations are preserved in this mapping. This means that the symbols or numbers follow the empirical relations that the real-world object has.

In this context, to "validate a software measure" means that the representation condition is satisfied. A mapping that follows the representation condition of measurement is well defined and valid.

Our cat scored zero on the IQ test → Our cat is stupid!

Fig. 9.5 Example of a wrong conclusion because of invalid measures [20]

For example, if we analyze the produced source code of a team and count the lines of code, this is a way to measure physical program size. In this case it fulfills the representation condition of measurement but it does not for functional program size: it is possible to have a poorly written program which is larger in terms of lines but has less functionality. In the case of functional program size, the representation condition is not fulfilled: a higher functional program size in reality does not correspond to a higher number (i. e., the mapping).

Once we have defined a correct mapping, we can focus on the data collection process: it should collect correct, accurate, and precise data:

- **Correctness** means that the data was collected according to the mapping rules that we defined. So if we say that the data are incorrect, we mean that the rules to collect it were not followed, which means that our conclusions (based on the assumption that the data was collected according to the rules) could be wrong.
- **Accuracy** describes how close our data reflects the "true" value. If the measurement is inaccurate, it could be that we are measuring using the wrong means (remember the example of the cat above).
- **Precision** tells us to which degree of detail our measurement process reliably measures. Accurate but imprecise measurement obtains data which reflects the true value, but only to a certain degree of detail.

Assume the weather on a specific day will be partly sunny with a passing shower in the morning. In the afternoon, the sunshine will even increase. The following four forecasts illustrate the differences between accuracy and

(continued)

precision (giving that the real weather was sunny with a bit of rain in the morning):

- Inaccurate, imprecise: "It will rain all day."
- Inaccurate, precise: "Mostly cloudy and windy in the morning. Mostly cloudy; windy with a few showers in the afternoon. Windy with rain at times in the evening. Windy with rain at times overnight."
- Accurate, imprecise: "It will be sunny all day."
- Accurate, precise: "Partly sunny with a passing shower in the morning. Abundant sunshine in the afternoon. Breezy in the evening. Temperatures will be from 24 to 30°C."

Precision is important in several situations: if we are performing a race on a sailing boat, a precise wind forecast that tells us that starting from 3 p.m. on a specific area of the lake it will be windless is crucial to win the race. In other situations it is less useful: when hiking on a mountain, it is not of extraordinary importance to know the height of the mountain to the millimeter.

The required precision will tell us which scale we need: is it, e.g., useful to know the exact amount of lines of code per class of some code base or would it be better to define groups (like "small," "medium," "large") and assign the classes to these groups? It depends on the specific circumstances and needs, but less information can be more valuable since it can convey the message faster.

In emergencies, or in general, whenever timeliness is more valuable than to know the exact measure, imprecise (but accurate) data are accepted. John Maynard Keynes stated once that "it is better to be vaguely right than precisely wrong."

The last desirable property of data that we want to point out is objectivity. Objectivity (as the opposite of subjectivity) in our context means that independently from who collected the data, the collected data will be the same. Nevertheless, sometimes subjective data are the only data that is available, for example, if we interview people about their opinions. The result of the interview depends on the way how the interview was conducted. Particularly in this case, it is important to document the methodology that was used to obtain the data so that everybody can understand what the obtained data represents.

This is also why we want that the data are correct: we want that the data was collected according to (documented) mapping rules.

This idea is related to the concept of consistency, which means that independently from who measured using which measuring tool, we want that the collected data are the same. So if two developers count the lines of code of a class, we want to obtain the same, consistent result. Moreover, if the data

(continued)

are correct, the collection process can be replicated. This means that others can, using the same mapping rules, collect data in the same way.

Usually we do not just want to collect data but also to analyze it. If we want that the analysis of the data are also objective, we have to document which data we use, how we analyze it, and which conclusions we obtain in such a way that the analysis can also be replicated. This does not guarantee that our conclusions are valid but that their validity can be verified by others repeating the analysis.

9.2 Does Measurements Collection Pay Off?

The aim of measuring is clear: we want to understand the development process better to improve it. Is it worth it? Do benefits outweigh the costs? Collecting and analyzing measurements takes time and therefore costs money. Moreover, "software development is inherently different from a natural science such as physics, and its measurements are accordingly much less precise in capturing the things they set out to describe [15]." Therefore, we must be able to prove if, or better when and under which conditions, it is worth it.

Ideally measurements collection should [20]:

- keep procedures simple;
- avoid unnecessary recording;
- train staff in the need to record data and in the procedures to be used;
- provide the results of data capture and analysis to the original providers promptly and in a useful form that will assist them in their work; and
- validate all data collected at a central collection point.

This list considers the costs of data collection and the provided value to the team.

A standard way to calculate "if it is worth it" is to consider the return on investment (ROI).

Return on Investment

The return on investment can be calculated dividing a financial representation of the benefits through a financial representation of the cost [55]:

$$\text{ROI} = \frac{\text{financial representation of the benefits}}{\text{financial representation of the cost}}$$

(continued)

For example, let us assume that measurement costs about €1,000.00 per month (resulting from increased salaries because of additional time effort). If through the availability of measures we can reduce development costs by €1,000.00 (because we spend less time in unnecessary activities), the resulting ROI would be 0: the benefit (i.e., the advantage that we have, the gain) equals to the outcome (€1,000.00) minus the costs (€1,000.00). This (€0), divided by the costs (€1,000.00), results in 0.

If the benefits would rise to €2,000.00, the resulting ROI would be 1, which means that for every invested Euro, the investment returns one additional Euro.

If the benefits are about €500.00, the ROI becomes negative:

$$\frac{500.00 - 1{,}000.00}{1{,}000.00} = -0.5.$$

ROI is easy to understand: what matters is if it is positive, negative, or 0 (breakeven). In this example we did not consider two aspects:

1. It can be that the benefits and cost occur during longer time periods so that we want to value benefits and costs that occur today in a stronger way than those that will occur in 10 years.
2. We did not consider how to quantify benefits and costs (which measurements we use and how precise the measurement has to be).

The first aspect can be solved by using the net present value (NPV) to analyze profitability [55]. NPV takes into consideration the time span between an investment and the occurrence of returns. It can be calculated discounting every benefit or cost using the following formula:

$$\frac{R_t}{(1+i)^t},$$

where t is the point in time the benefit or cost occurs (i.e., the number of the year), i is the discount rate (usually a rate of return that could be earned investing the resources in other projects with similar risk), and R_t is the cash flow at time t (i.e., a positive number representing benefits, a negative number representing costs).

Through the discounting of returns, those returns that occur tomorrow are considered to be worth less than those that occur today. This means that this calculation favors profits today than profits tomorrow.

The second aspect, the question of how to quantify benefits and costs, has to consider which part of the reality we want to cover (i.e., how many measurements we want to collect) and the degree of accuracy that is needed.

To decide how to measure benefits, we have to understand what our clients value. Do our clients value quality? Then it is necessary to further investigate what quality means for our customers and to measure that (e.g., ease of use, as few defects as possible, etc.). Do they value a fast delivery more than quality? Then we should keep measurements about quality to a minimum, and so on.

What is a benefit or not should depend on what the customers value. In certain markets, it is time-to-market, and in others it is costs, prestige (such as swiss watches), a specific functionality (such as wireless charging), a sustainable production process etc. In some cases it is both. To collect measurements that are not needed or not used means to waste time and money. This is exactly what Lean Thinking wants to avoid: if a measurement is not used, the costs to collect and analyze it should be removed.

The GQM approach described in Chap. 7 helps to link measurements, to questions and those to goals. A GQM model should be created also for this reason: the usefulness of a measurement becomes clear because of its link to a business goal.

After defining a minimum set of measurements, we have to decide about the precision that is required to understand the delivered value to our customers. The costs of collecting measurements, evaluating the results, and using the obtained knowledge depend on the necessary effort to perform these tasks, which is influenced by the chosen degree of precision.

In fact, to have a reliable computation of ROI in software development, one needs to have in place a measurements collection program. We know that Lean cannot be implemented without a suitable collection of measurements. Therefore, measurements are a main part of our approach, not an add-on. Our goal is to implement an effective measurement program, that is, we need to determine what are the measurements needed by our customers and with what level of precision.

How often did we see claims about an increase of ROI that are pure science fiction? Such computation of ROI is a waste and extremely dangerous as managers might believe in it, once. Then, after the first disillusion, one might consider the whole issue of data collection a pure fad, rejecting it forever.

In summary, software measurement is needed. However, we need to organize it so that it can be effectively used; otherwise, it might end up being just a waste. We need to constantly verify if the gained insight outweighs the caused costs since this depends on the concrete case, for example, as in the example of profitability in Table 9.2.

9.3 Non-Invasive Measurement

The term “non-invasiveness” comes from the medical field, in which it means that the diagnosis does not require to actually look into the body [7]. In software measurement, we mean the use of methods to collect data about the software development process, about the product, and about the employed resources that do not require the personal involvement of the participants of the process [48, 49].

Table 9.2 Example of a ROI calculation [55]

Item	Value
Costs:	
Costs to collect measurements by the software developers (80 h of effort in measurement-program-related tasks, measured from the hour-registration system)	$5,000
Costs to elaborate and analyze the collected data (240 h of effort for the measurement program, measured from hour-registration system)	$15,000
Total costs	**$20,000**
Benefits:	
Direct benefits:	
260 h of effort saving during the measurement program due to a measured reduction of interrupts	$16,000
60 h of effort saving due to the possible reuse of the collected data in other investigations	$9,000
Total direct benefits	**$25,000**
Indirect benefits:	
1-week-early product delivery, measured from value the marketing manager indicated	$100,000
Effort saving during remainder of the year due to the reduction of interrupts	$50,000
Increased focus on quality and time expenditure, both in the project as in other groups, measured from value for group manager (combination of buy-in and personal value)	$100,000
Update of engineering documentation due to a measurable number of interrupts on these documents, documentation measured from value for engineers	$5,000
Total indirect benefits	**$255,000**
Total benefits	**$280,000**
ROI (calculated as (280,000–20,000)÷ 20,000)	**13**

To compare non-invasive measurement with invasive measurement, we will relate the ROI of both approaches. Therefore, we will relate the costs and the expected benefits of implementing both types of measurement.

The introduction costs of a measurement program (for the first year) can account for 1–2 % of the total engineering or IT effort [16]. To make an example, Rico and Pressman [44] studied the complete cost to use a manual measurement program like the Personal Software Process [25] to help produce 10,000 lines of code and obtained as a result $145,600. To reduce such costs is one significant motivator for using non-invasive measurement tools [31, 47].

The costs (and so the return on investment) depend on the scope of the measurement program. Some costs arise for both approaches, and some only for

one of the two. To compare the type of costs that arise in manual and non-invasive measurement, we assume that they are carried out by the two processes shown in Fig. 9.6 (manual data collection) and Fig. 9.7 (non-invasive data collection).

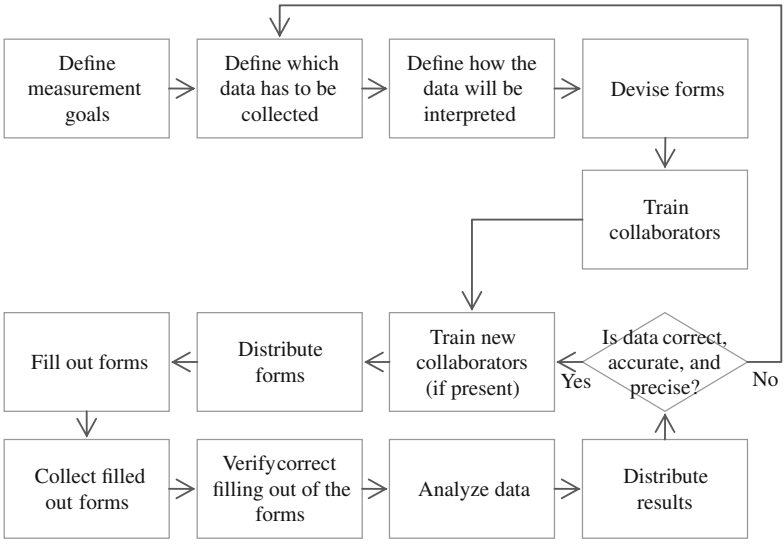


Fig. 9.6 Example process to introduce and maintain a manual measurement program

For the manual measurement program of Fig. 9.6, we assume that forms are used to collect data about the performed activities. Employees are supposed to fill out these forms regularly and to hand them in for data analysis. The measurement team has to verify the quality of the data regularly, analyze the data, and distribute the results of the analysis.

We assume a similar process for non-invasive measurement in which non-invasive tools are developed or acquired after deciding which data has to be collected. The main differences in the cost structure are that non-invasive measurement does not require to train collaborators on how to precisely and accurately collect data, that forms have to be filled out at all, and that employees do not have to be checked whether they filled out the form correctly or not [52].

These two scenarios represent the data collection activities on the short run. On the long run, the entire process will be repeated since new measurement goals will arise and this will require that all steps have to be performed from the beginning.

Table 9.3 lists the costs we consider in the two scenarios.

The costs reported so far represent costs caused by the activities required by a given approach. The choice for one or the other approach causes also indirect costs, i.e., costs that are not caused by the activities of the measurement process but that are risks caused by the adopted measurement method. Some of such risks are:

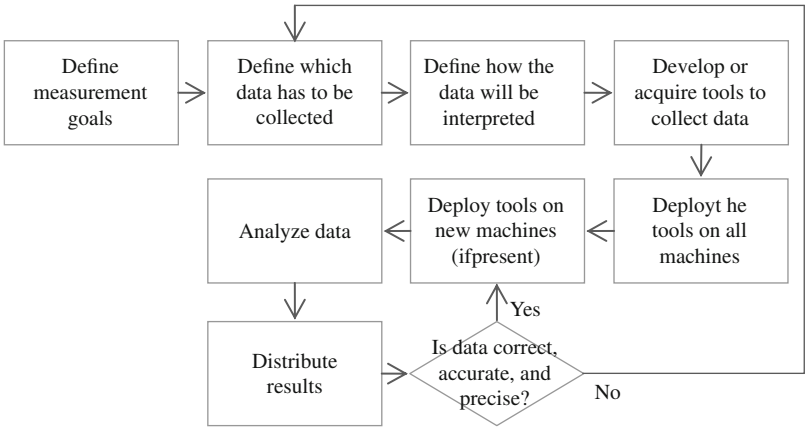


Fig. 9.7 Example process to introduce and maintain a non-invasive measurement program

- **Risk of distraction**: manual data collection forces employees to switch between work and data collection on a regular basis since it is recommended to collect the data while the memory about the reported fact is still fresh. Intuitively, the longer the time that passes, the less precise will the memory about the fact be. This means that an employee will be encouraged to regularly fill out data collection forms to ensure a high accuracy of the data.

  Task switching requires time and leads to a performance cost [32, 37, 41, 45, 46]. It creates costs because workers need time to "reorient" [9, 26, 56]. Steven Jenkins [30] defines "deliberately planned, chronic interruptions" even as "worst-case scenario" and advises to "never let people work on more than one thing at once."
- **Risk of inaccurate data**: the quality of manually collected data depends on the care and attention that the participant is willing to invest in data collection. This means that the data quality might be influenced by the context in which it is collected such as workload, project, time, date, location, etc.
- **Risk of incomplete data**: the non-invasive approach cannot collect data about nonobservable events. If data about nonobservable aspects (like opinions) is needed, we have two possibilities:

  – we use manual data collection (that means we measure directly), or
  – we find an indicator [2], i.e., some observable aspect that helps us to understand what we want to find out. (This means we measure indirectly.) To evaluate an opinion about a certain product, we can use the sales figures to estimate how good or how bad customers find a certain product. On the other hand, we have to accept that the selected indicator might be incorrect, inaccurate, or imprecise.

- **Risk of incorrect data**: non-invasive data collection records data without human participation. A consequence of this is that incorrect data cannot be rectified

Table 9.3 Cost factors of non-invasive and manual measurement

Cost description	Manual	Non-invasive
Setup costs:		
Define measurement goals, i.e., what has to be achieved	×	×
Define which data has to be collected	×	×
Define how the collected data will be interpreted	×	×
Devise forms	×	
Train collaborators on how to correctly fill in the forms	×	
Develop or acquire tools to collect the data non-invasively		×
Deploy the tools on all machines		×
Recurring costs:		
Train new collaborators on how to correctly fill out the forms	×	
Develop or acquire new tools to collect the data non-invasively		×
Deploy tools on new machines		×
Distribute forms	×	
Fill out forms with measurements	×	
Collect filled out forms	×	
Verify that the forms have been filled out correctly	×	
Analyze the results	×	×
Distribute the results	×	×
Verify that the collected data are correct, accurate, and precise	×	×

immediately (on the time of recording). It is necessary to find the wrong record after the recording and to update it. Everybody that tried to use speech recognition software to author a document made this experience: it might take more time to fix all the errors than to type it by hand. We face the same trade-off in this situation. If the non-invasive data is not accurate or precise enough, it can cost more to fix the data than to record it by hand. If there is no time or possibility to rectify the data and we accept incorrect data, the risk of wrong decisions increases.

Table 9.4 summarizes these risks again together with the data collection method for which they apply.

Table 9.4 Risks connected to the data collection method

Risk description	Manual data collection	Non-invasive data collection
Risk of inaccurate data	×	
Risk of distraction	×	
Risk of incomplete data		×
Risk of incorrect data		×

Once the decision is made to measure (either manually or non-invasively), we need to understand the cost development in relation to the measurement scope, i.e., the quantity and quality of the data we need to collect. Measurement scope is determined by the number of measures we need to collect, the precision we require, etc. There is a positive relationship between cost and scope. In general, the more scope is needed, the more costs are generated. We assume that the more data we need to collect, the more sophisticated the involved technologies have to be. This includes databases, data transfer mechanisms, analysis methods, visualization methods, etc. Sophisticated technologies increase the learning and maintenance effort for the development team and therefore also the costs. This means that it is not cheap to collect a large amount of low-quality data: it would still require the setup and maintenance of a system that is capable of processing such large amount of data. On the other hand, collecting few high-quality data might also require sophisticated technologies or algorithms. In general we assume that an increase of either quality or quantity (or both) causes an increase in costs; this is why we aggregate those two dimensions into "measurement scope."

We cannot calculate the measurement costs without knowing the specific environment in which a measurement program is going to be introduced, but we can present factors that should be taken into consideration.

Figure 9.8 shows a possible development of costs in relation to the time dedicated to measurement.[1] It is based on the following assumptions:

1. The costs of measurement increase with the quantity and quality of data that is needed.
2. Switching from product development to measurement and back requires time and leads to a performance cost [32, 37, 41, 45, 46]. As the amount of task switches and the measurement effort increases, the higher the costs of task switching. Humphrey [25] recommends that the programmer has to collect the

[1] Figure 9.8 is not based on empirical data, but on our experience. The costs in a particular context can be different than the ones depicted here.

measurements himself to maximize the learning effect. We assume that even in the case that the programmer is not collecting the data himself, he would still need to switch the task to answer the questions of the person that measures.

3. Decisions based on inaccurate data can cause costs. We assume that initially, as more and more data are collected, also the possible cost of wrong decisions due to inaccurate data rises. After a certain point, we collect enough data so that we can cross-check the data for its accuracy and the costs can be reduced again.[2]

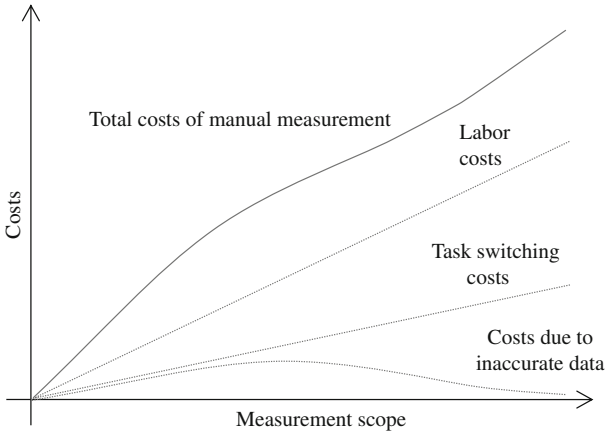


Fig. 9.8 Possible total measurement costs of an organization in relation to manual measurement effort.

The curves of Fig. 9.8 change as the context changes. The labor costs rise or fall depending on the salary of the person that performs the measurement and based on the activities (e.g., cross-checking data) that the measurement includes. The distraction costs fall or rise if the developer has a high or low task switching ability [3] and if the collected measures require a low or high degree of concentration. Finally, the costs due to inaccurate data fall or rise depending on the type of data and on the type of decisions that are based on that data.

As said before, the type of costs and the development of costs change as the context changes. Nevertheless, the instrument of depicting the foreseen development of costs in relation to the measurement scope can help to decide how much measurement is optimal.

[2]The costs of wrong decisions are represented as smaller than task switching costs. In fact, wrong decisions can cost much more than task switching costs. In this picture we look at the cost development, not at the absolute values that the curves depict.

In the case of non-invasive measurement, we expect a cost development as in the Fig. 9.9.[3] It is based on the following assumptions:

1. The costs of measurement increase with the quantity and quality of data that is needed. Differently from manual measurement, the labor costs grow only when new tools have to be developed to cope with new types of measurements. Once the tools are developed, the cost of collecting data is independent from the number of measurements effectively collected, the precision, etc.
2. Some data cannot be collected automatically. We have to accept either not to use such data or to rely on indicators. We consider the cost of incomplete data in the graph as a decreasing cost. We think that the costs decrease since as the amount of data grows, other data, that may not be collected automatically, can be inferred.
3. Decisions based on incorrect data can cause costs. We expect that this cost develops in a similar way as the cost of inaccurate data of manual measurement. Initially, as more and more data are collected, also the possible cost of wrong decisions due to incorrect data rises. After a certain point, we collect enough data so that we can cross-check the data for its correctness and the costs can be reduced again.

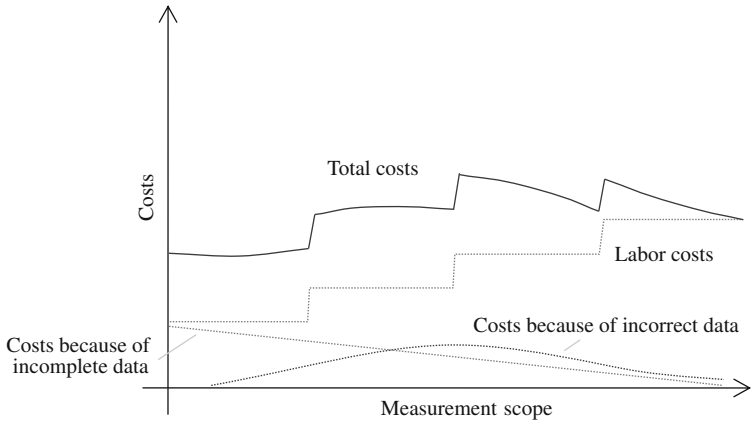


Fig. 9.9 Possible total measurement costs of an organization in relation to non-invasive measurement effort

The curves of Fig. 9.9 also change as the context changes. The labor costs rise or fall depending on the salary of the person that performs the measurement. Investing money into the development of measurement components instead of teaching employees how to collect data has two advantages:

[3]Figure 9.9 is not based on empirical data, but on our experience. The costs in a particular context can be different than the ones depicted here.

- measurement components can be reused throughout the company, i.e., the development costs pay off faster and
- measurement components embed the measurement knowledge, i.e., the knowledge of how a specific measurement is collected.

The incorrect data costs fall or rise depending on the difficulty to directly collect some specific data or on the quality of the indicator. Finally, the costs due to incorrect data falls or rises depending on the type of data and on the type of decisions that are based on that data.

Following the distinction between value-adding work, non-value-adding work, and waste mentioned in Chap. 2 (see Fig. 2.5), measurement itself is a non-value-adding work; it is not directly adding value to the output, and it is not required by the customer. However, in software engineering it is needed to improve the value-adding activities.

Software developers obtain considerable benefits from manually collecting data and analyzing it [25]. However, the considerable effort and the disciplined approach it requires are "too intrusive for many users who desire long periods of uninterrupted focus for efficient and effective development [32]."

Non-invasive measurement reduces the effort to collect data through the automation of the data collection steps.

Not all possible measurement activities can be automated, since the wide range of data often collected comes from sources which are difficult or impossible to read in an automated way. To overcome this problem, it is possible to measure indirectly: instead of measuring the attribute of an entity directly, we either measure another attribute that correlates with the one we are interested in or we combine other measurements to infer the desired measurement [2]. It can be that indirect measurement results in a lower precision, but in many cases this is acceptable.

An example of indirect measurement is the time a programmer spends on a project. A direct way to measure it is to require every developer to fill out a time sheet. An indirect way to measure the time spent on a project could be to measure the time spent editing files that are in the project folder on the server. A tool could trace the time a person edits documents and sum up all the editing time [51].

Summing up all the times recorded by this tool, it is possible to obtain the time spent editing the documents of a given project X (see Fig. 9.10), which (if most or all the time is spent editing documents, writing code, etc.) can be a good approximation.

A way to overcome a too low precision is to combine different indirect measurements to obtain a more detailed picture. In our example it could be possible to write a plugin for the calendar application of every developer that collects the time spent on meetings for each specific project and add that time to the project time obtained using the method above (see Fig. 9.11).

As said above, sometimes the obtained precision is too low. For example, if we need the exact time spent per activity, e.g., for billing purposes, manual data collection is the only possibility.

The advantages of non-invasive measurement lie not only in lower data collection costs: through the automated data collection, the data are collected always in

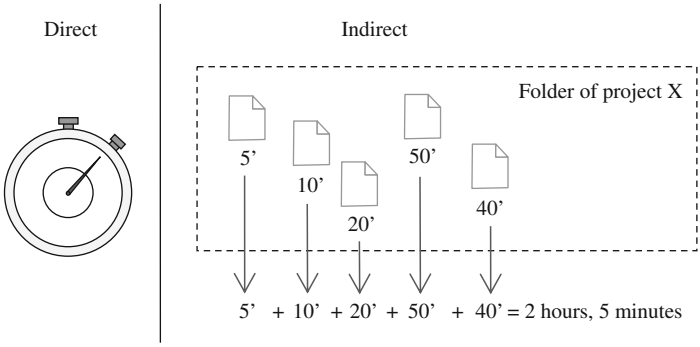


Fig. 9.10 Direct vs. indirect measurement

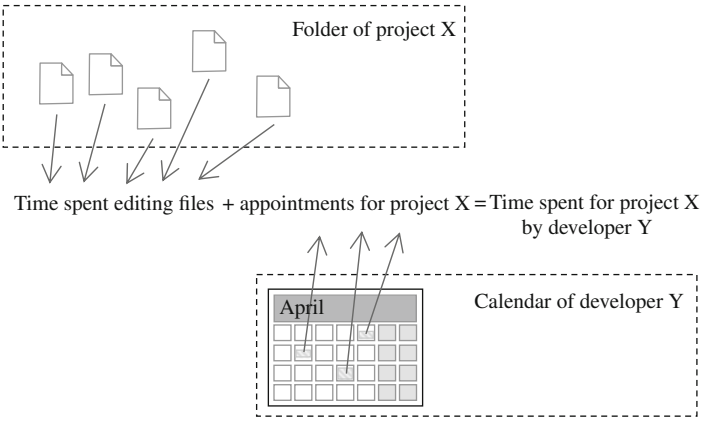


Fig. 9.11 Aggregation of different measurements to increase precision

the same way (ensuring correctness and consistency). Moreover, an automated collection process does not depend on the person performing it, which ensures objectivity.

In the context of Lean thinking, non-invasive measurement provides the input to the problem detection mechanism used within autonomation (see Chap. 2). Autonomation goes one step further then non-invasive measurement: not only the data collection is automated but also the interpretation.

Using autonomation, it is possible to define a specific set of data together with rules on how to interpret this data in order to warn developers or to stop the current production process (in software development, we cannot stop the assembly line, but we could, for example, block the check-in to the source control system if problems within the code are found) [1, 10, 21].

9.4 Implementing Non-invasive Measurement

In this section we want to give a more concrete idea of how non-invasive measurement can be implemented. Non-invasive measurement for software development should be able to capture all possible aspects of software engineering. Following Fenton's classification of software measurements, we are able to collect [20]:

- **process** measurements to describe the activities performed during software development;
- **product** measurements to describe the output of the activities (documentation, code, etc.); and
- **resource** measurements to describe the input of the activities (human effort, electricity, etc.).

These aspects can be analyzed about their internal attributes (describing their properties) or about their external attributes (describing how a process, product, or resource relates to its environment). Some examples of measurements are shown in Table 9.5: structure measurements analyze the components of an entity and how these components are combined with each other. This helps, for example, to estimate the quality of a component [19]. Size measurements describe the extent of entities to estimate, e.g., the amount of test cases needed to cover all paths through the code [35]. Resource costs describe in general how much it costs to operate a resource for a given amount of time.

Table 9.5 Risks connected to the data collection method

	Internal	External
Process	Structure	Duration
Product	Size, structure	Performance
Resource	Costs	Utilization

A non-invasive measurement framework for software development provides a set of components to measure internal and external attributes of processes, products, and resources [10,29,50]. These components can be seen as measurement probes or sensors put into the software development process. These probes then report events to a central repository where they can be analyzed in a second moment. Examples for such probes are:

- An application for Google Android [23] that logs all phone calls from customers;
- A plugin for Eclipse [17] that logs the time spent developing and executing test cases using JUnit [33];
- A plugin for Microsoft Visual Studio [39] that logs the time spent developing code;

- A plugin for Microsoft PowerPoint [38] that logs the time spent developing presentations;
- A plugin for GIT [22] that connects to a source code repository and scans the code for anomalies; or
- A custom plugin embedded in an application that reports specific events.

In general, if we analyze a software development process, we can collect [20]:

- **Product measurements**: systems that contain data about the product, such as source code management systems, bug tracking systems, testing environments, project management systems, etc., can be interfaced and the necessary information extracted [27, 50].
- **Process measurements**: systems that are involved in software development activities can be used to keep track of the process that is followed to develop the needed output (such as mobile devices [8], software applications such as word processors or integrated development environments, etc.). Measurement probes can interface with the involved applications and log the performed activities.
- **Resource measurements**: resources are usually used during software development. Their current utilization, workload, availability, efficiency, etc. can be monitored and logged by measurement probes and reported to the measurement server.

We distinguish two approaches to extract data:

- batch mode: the data are extracted on a regular basis (e.g., every night),
- background mode: the data are extracted instantly.

The batch mode is useful when detailed logging of the ongoing events is not needed or if the data extraction process slows down the interfaced device or application and therefore is too costly in terms of performance. Vice versa, the background mode extracts data as soon as it becomes available.

We also distinguish two approaches to submit data to the central repository:

- online: the collected data are immediately submitted to the server,
- cached: the collected data are kept locally and submitted later.

A cached submission is necessary for devices that are not always connected to the network, such as laptops, smart phones, tablet computers, etc. Also if the available bandwidth is low, a cached approach that collects a chunk of data, compresses it, and sends it later helps to keep the necessary transfer time low.

The component shown in Fig. 9.12 shows the UML diagram of a measurement probe that checks out the last version of a source code repository, calculates source code measurements, and sends the result to the server. Following our categorization above, it works in batch mode and submits data using an online approach.

The component of Fig. 9.13 shows the UML diagram of a probe that collects data (from two sources) online, transmitting the data using a cache.

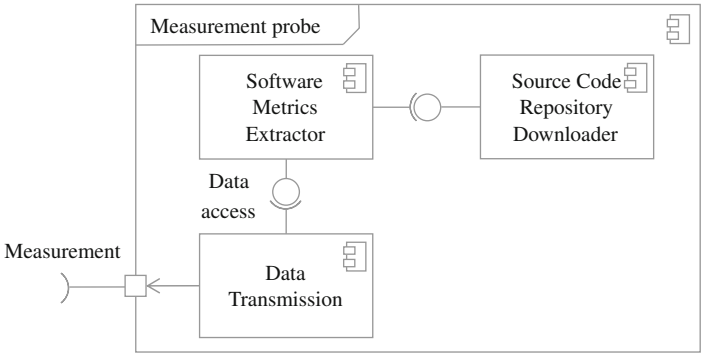


Fig. 9.12 An online, batch mode measurement probe

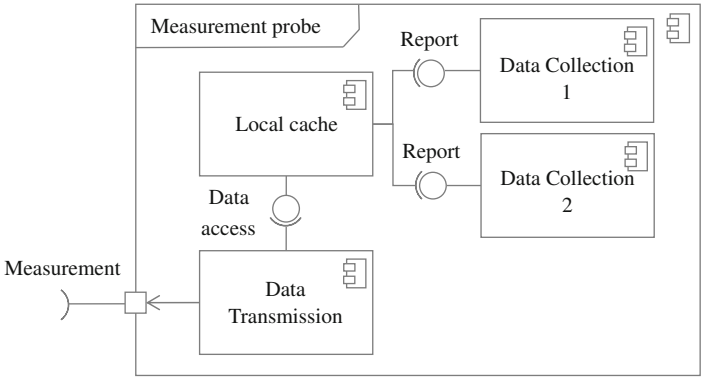


Fig. 9.13 A cached, background mode measurement probe

If not otherwise possible, this architecture can be extended to allow also manual data input using a "measurement probe" that collects the manual input and sends it in the same format as non-invasive probes (see Fig. 9.14) [43].

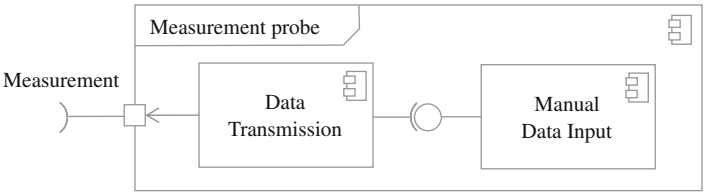


Fig. 9.14 A measurement probe for manual data collection

The data collected by the different measurement probes has to be transmitted to a central repository. If for the transmission of the collected data a protocol is used that works over the Internet, the measurement framework can also be used for distributed teams.

A possible database schema using crow's foot notation is shown in Fig. 9.16 [28].

Crow's Foot Notation in Short

The crow's foot notation is a notation in which to indicate cardinality; a fork or crow's foot indicates "many" by its many "toes," hence the name.
The cardinality is the maximum number of times one record of one table can be associated with records of the other table in the relation. The cardinality can be either 1, then it is indicated by a line, or many, then it is indicated by a fork.

The modality is the minimum number of times one record of one table has to be associated with records of the other table in the relation. The modality can be either 0, then it is indicated by a circle, or 1, then it is indicated by a line.

The cardinality and modality are indicated on the line that connects two tables (see Fig. 9.15).

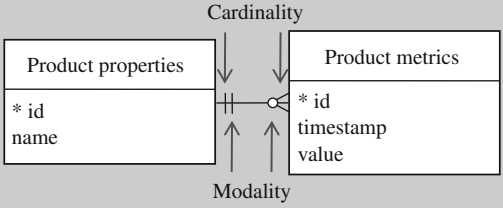


Fig. 9.15 Cardinality and modality in the crow's foot notation [6]

A table is denoted by a rectangle that contains the name of the table and the names of the columns. The primary key columns are marked by a star at the beginning of their name. A detailed explanation of the notation can be found in [6].

In this database we store measurements for product, process, and resource measurements. The database depicted in Fig. 9.16 is an idealized schema. In fact the first version of a non-invasive measurement framework we implemented had a similar schema as the one depicted here. Unfortunately, as the number of measurements one wants to collect increases and as the number of developer increases, the amount of data to handle can become very large. In Appendix B we present a detailed architecture of the actual implementation we use.

Product measurements and resource measurements are stored following the same pattern: they describe a product or resource using a defined set of properties. The properties are stored in the tables "product properties" and "resource properties" and are linked to the relative products and resources through the tables "product measurements" and "resource measurements."

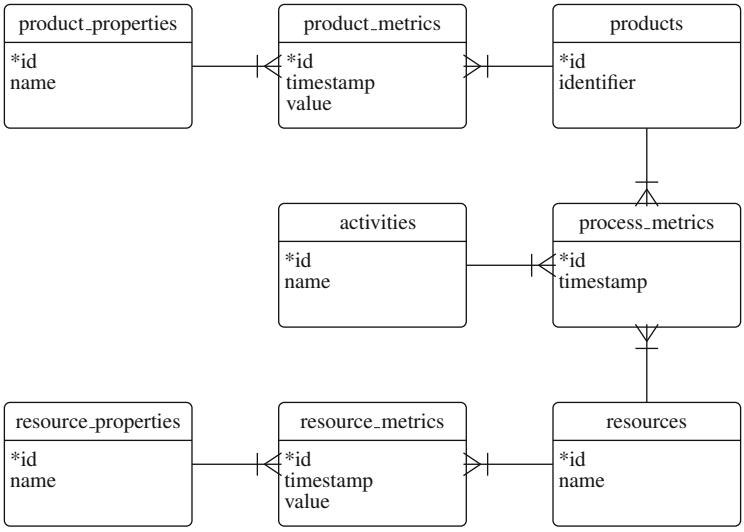


Fig. 9.16 A possible database schema for collecting measurements

In our example database, process measurements describe which activities modify (create, update, delete) which artifacts under the utilization of which resources. Therefore, process measurements relate the three tables: "products," "activities," and "resources."

The concrete implementation of the measurement infrastructure depends on the requirements it has to fulfill. An infrastructure that aims to collect data for a team of 12 people on a daily basis is different from an infrastructure for 100 that provides real-time information on all events taking place.

9.5 The "Big-Brother" Effect of Non-invasive Measurement

Non-invasive data collection has the advantage of being invisible to the user so that he has no additional effort in collecting data about the software development process.

To manually collect data about one's own development process helps to better understand what the collected measurements mean, how they should be interpreted, and what to do to improve the results. Past work on collecting and analyzing data on one's own development process has shown to help developers to understand and improve their performance [25].

Any company that follows Lean Thinking also collects data to improve. Non-invasive measurement allows to collect a large variety of data without human intervention. On the one hand, this frees workers from the measurement effort and motivates them because they can dedicate their attention to more important,

interesting aspects of their work. On the other hand, there is the risk that constant observation is perceived as intimidating, is felt as a permanent stress factor, and leads to an overall lower performance [4].

Being constantly observed can be seen as a decrease in autonomy, i.e., that the available alternatives for taking action that one has [11, 40] diminish **because** of the observation. A decrease in autonomy, i.e., feeling that one has not the freedom to choose what is best for him, has a negative impact on the motivation, which furthermore can have negative consequences in the productivity of the single employee [12, 13].

Software development depends mainly on the people involved. When we measure, we have to consider that software development relies on creative, motivated people. People that have a personal interest in what they are doing are far more productive than those that are not [24]. According to Fred Brooks, a good programmer can be up to 5–10 times more productive than a bad programmer [5].

Therefore, companies are interested to keep their motivated programmers, not to scare them away, and to help them to stay productive.

The concrete measurements that are collected and how they are used depends on the management and on their perception of the situation inside the company. The way in which management sees and interprets the current work of the employees has an impact on the selection of the information in which the management is interested [34].

We have to distinguish two possible uses of measurement: to enhance understanding or to control. If an input to the developers is experienced as an information, it enhances their autonomy, whereas if it is experienced as controlling, it diminishes autonomy [11]. The GQM approach presented in Chap. 7 helps to communicate what is collected and how the collected data are useful as an instrument to improve.

Trying to monitor ongoing processes with the aim to identify problems in the process and to come up with possible solutions before the problem becomes critical is a legitimate goal for the management of a company. Monitoring employees to verify that the performed work meets the expectations is a known issue in the management literature and known as "principal-agent theory": "Information enters the picture because the principal needs to know if the agents are carrying out the desired actions or at least are delivering the desired outcomes or results [18]."

Through monitoring, employees can be induced to behave according to the expected behavior and are observed for this reason. This type of monitoring is a form of external intervention intended to change a person's behavior and therefore poses a threat to autonomy: "... Surveillance displaces autonomy, mistrust undermines self-regard, absence of support and help minimizes achievement, likelihood of punishment for noncompliance reduces risk-taking and innovation, rigidity of standards and administrative procedures precludes the individual's use of his own know-how [18]."

The suggestion is to focus on informational feedback that empowers people: "If such summary data indicate to the manager that something is wrong within the organizational unit for which he is responsible, he will turn not to staff [i.e., not to the accounting or finance staff providing the data], but to his subordinates for help

in analyzing the problem and correcting it. He will not assign staff 'policemen' the task of locating the 'culprit.' If his subordinates have data for controlling their own jobs, the likelihood is that they will already have spotted and either corrected the difficulty themselves or sought help in doing so [18]."

One of the most traditional theories that deals with the way in which management sees and interprets the current work of the employees is McGregor's "Theory X and Theory Y" [36]. "Theory X and Theory Y" assumes that the behavior of management depends on the "implicit personality model" that the manager has about the employees. The assumptions on which both theories are based are summarized in Table 9.6.

Table 9.6 Assumptions of Theory X and Theory Y [36]

Theory X	Theory Y
The average human being has an inherent dislike of work and will avoid it if possible	The expenditure of physical and mental effort in work is as natural as play or rest
Because of this dislike of work, most people must be coerced, controlled, directed, and threatened with punishment to get them to give adequate effort towards the achievement of organizational objectives	External control and threat of punishment are not the only means for bringing about effort towards organizational objectives. People will exercise self-direction and self-control in the service of objectives to which they are committed
The average human being prefers to be directed, wishes to avoid responsibility, has relatively little ambition, and wants security above all	Commitment to objectives is a function of the rewards associated with their achievement. The average human being learns, under proper conditions, not only to accept but to seek responsibility
	The capacity to exercise a relatively high degree of imagination, ingenuity, and creativity in the solution of organizational problems is widely, not narrowly, distributed in the population. Under the conditions of modern industrial life, the intellectual potentialities of the average human being are only partially utilized

Superiors are convinced that their employee's personality pertains to the "Theory X" and are convinced that in general the employee has a born antipathy against work and tries to avoid it wherever possible. Additionally, the employee prefers to be guided, does not like responsibility, and prefers security of employment. As a consequence, the employee must be guided using advice and threatened with punishment in order to convince him to fulfill the needed work tasks.

On the other side of the continuum, superiors that see their employee's personality belonging to the category "Theory Y" believe that working is a natural activity like playing or resting. To supervise and to threaten with punishment is not the only

way to induce somebody to work for the companies' goals, but it is also possible that employees internalize the goals of the companies and consider them their own goals and therefore comply with the rules and needs of the company. How much somebody feels committed towards the companies' goals depends on the rewards that are connected to the fulfillment of the given task. The average employee, under appropriate circumstances, does not only learn how to assume responsibility but also to seek it.

The interesting result of McGregor [36] is that if a manager sees his employees as "Theory X" and treats them as Theory X, they will become like Theory X. They will start to dislike work and will do just the minimum that is required. Theory X is a so-called self fulfilling prophecy: since a manager thinks his employees are lazy, he uses coercion and control. Then, it is exactly this coercion and control that takes away any motivation and makes the employees become as the manager expects.

Theories X and Y are extremes on a continuum of possible management styles. It seems that the best solution is to adapt its style to the task and the person executing it [42]. Nevertheless, in those teams where we had the impression that a rather Theory X management style is in place, non-invasive measurement had negative effects on motivation. On the other hand, where we had the impression that a Theory Y management style is in place, non-invasive measurement had no negative effects on motivation.

9.6 Summary

This chapter presented non-invasive measurement, a way to collect data without distracting programmers during their work. We stressed that non-invasive measurement should be used in combination with measurement to clearly communicate what data are collected and why they are collected, so that employees do not feel controlled by the collected data but that they can use it to improve the software development process.

Problems

9.1. Assume you are a manager convinced that Theory X is true. Which non-invasive measurement probes would you want to develop to maximize productivity? Now assume you are convinced that Theory Y is true. Which non-invasive measurement probes would you need now?

9.2. We discussed that we foresee two ways to collect measurements non-invasively: in batch and in background mode. What are the advantages and disadvantages of each approach?

References

1. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: Supporting governance in disciplined agile delivery using non-invasive measurement and process mining. Cutter IT J. **26**(11), 25–29 (2013)
2. Atteslander, P.: Methoden der Empirischen Sozialforschung, 10th edn. Studienbuch Series. Walter de Gruyter, Berlin (2003)
3. Back, M.D., Schmukle, S.C., Egloff, B.: Measuring task-switching ability in the implicit association test. Exp. Psychol. **52**(3), 167–179 (2005)
4. Brödner, P., Knuth, M.: Nachhaltige Arbeitsgestaltung: Trendreports zur Entwicklung und Nutzung von Humanressourcen. Bilanzierung innovativer Arbeitsgestaltung. Hampp, München (2002)
5. Brooks, F.P., Jr.: The Mythical Man-Month (Anniversary edn.). Addison-Wesley Longman, Reading (1995)
6. Carlos Coronel Steven A., Morris, P.R.: Database Systems: Design: Design, Implementation, and Management, 9th edn. Cengage Learning, Boston (2009)
7. Collins: Collins English Dictionary — Complete & Unabridged, 10th edn. HarperCollins. Online: http://www.collinsdictionary.com (2009). Accessed 4 Dec 2013
8. Corral, L., Janes, A., Remencius, T., Strumpflohner, J., Vlasenko, J.: A novel application of open source technologies to measure agile software development process. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W., (eds.) Proceedings of the International Conference on Open Source Systems: Long-Term Sustainability (OSS), IFIP Advances in Information and Communication Technology, vol. 378. Springer, New York (2012)
9. Czerwinski, M., Horvitz, E., Wilhite, S.: A diary study of task switching and interruptions. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI). ACM, Vienna (2004)
10. Danovaro, E., Janes, A., Succi, G.: Jidoka in software development. In: Harris, G.E. (ed.) Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, Nashville (2008)
11. Deci, E.L., Connell, J.P., Ryan, R.M.: Self-determination in a work organization. J. Appl. Psychol. **74**(4), 580–590 (1989)
12. Deci, E.L., Ryan, R.M.: The "what" and "why" of goal pursuits: human needs and the self-determination of behavior. Psychol. Inq. **11**(4), 227–268 (2000)
13. Deci, E.L., Ryan, R.M., Gagne, M., Leone, D., Usunov, J., Kornazheva, B.: Need satisfaction, motivation, and well-being in the work organizations of a former eastern bloc country a cross-cultural study of self-determination. Pers. Soc. Psychol. Bull. **27**(8), 212–225 (2001)
14. DeMarco, T.: Controlling Software Projects. Yourdon Press, New York (1982)
15. DeMarco, T.: Software engineering: an idea whose time has come and gone? IEEE Softw. **26**(4), 95–96 (2009)
16. Ebert, C., Dumke, R.: Software Measurement: Establish, Extract, Evaluate, Execute. Springer, New York (2007)
17. Eclipse Foundation: Eclipse ide. Online: http://www.eclipse.org (2013). Accessed 4 Dec 2013
18. Ellerman, D.: Mcgregor's theory vs. bentham's panopticism: toward a critique of the economic theory of agency. Knowledge Technol. Policy **14**, 34–49 (2001)
19. Emam, K.E.: A primer on object-oriented measurement. In: Proceedings of the International Software Metrics Symposium (METRICS). IEEE Computer Society, London (2001)
20. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, Boston (1998)
21. Gašparič, M., Janes, A., Heričko, M., Succi, G.: Metrics-based recommendation system for software engineering. In: Heričko, M. (ed.) Proceedings of the International Multiconference Information Society, Collaboration, Software and Services in Information Society (CSS). Ljubljana, Slovenia (2013)
22. GIT contributors: Git. Online: http://git-scm.com (2013). Accessed 4 Dec 2013

23. Google: Google android. Online: http://www.android.com (2013). Accessed 4 Dec 2013
24. Herzberg, F.: One more time: How do you motivate employees? Harv. Bus. Rev. **46**(1), 53–62 (1968)
25. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Professional, Boston (1996)
26. Ikonen, M.: Leadership in kanban software development projects: a quasi-controlled experiment. In: Abrahamsson, P., Oza, N.V. (eds.) Proceedings of the International Conference on Lean Enterprise Software and Systems (LESS), Lecture Notes in Business Information Processing, vol. 65. Springer, Helsinki (2010)
27. Janes, A., Scotto, M., Pedrycz, W., Russo, B., Stefanovic, M., Succi, G.: Identification of defect-prone classes in telecommunication software systems using design metrics. Inf. Sci. **176**(24), 3711–3734 (2006)
28. Janes, A., Scotto, M., Sillitti, A., Succi, G.: A perspective on non invasive software management. In: Proceedings of the IEEE Instrumentation and Measurement Technology Conference (IMTC). IEEE, Sorrento (2006)
29. Janes, A., Sillitti, A., Succi, G.: Non-invasive software process data collection for expert identification. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE). Knowledge Systems Institute, San Francisco (2008)
30. Jenkins, S.: Concerning interruptions. IEEE Comput. **39**(11), 66–72 (2006)
31. Johnson, P.: You can't even ask them to push a button: toward ubiquitous, developer-centric, empirical software engineering. In: Proceedings of the NSF Workshop for New Visions for Software Design and Productivity: Research and Applications. Nashville (2001)
32. Johnson, P.M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S., Doane, W.E.J.: Beyond the personal software process: metrics collection and analysis for the differently disciplined. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society, Portland (2003)
33. JUnit contributors: Junit. Online: http://sourceforge.net/projects/junit (2013). Accessed 4 Dec 2013
34. Lueger, G.: Beschaffung und auswahl von mitarbeitern. In: Kasper, H., Mayrhofer, W. (eds.) Personalmanagement — Führung — Organisation, 2nd edn. Linde (1996)
35. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
36. McGregor, D.: The Human Side of Enterprise. Mcgraw-Hill, New York (1960)
37. Meiran, N.: Reconfiguration of processing mode prior to task performance. J. Exp. Psychol. **22**(6), 1423–1442 (1996)
38. Microsoft: Microsoft powerpoint. Online: http://office.microsoft.com/en-us/powerpoint (2013). Accessed 4 Dec 2013
39. Microsoft: Microsoft visual studio. Online: http://www.microsoft.com/visualstudio (2013). Accessed 4 Dec 2013
40. Mikl-Horke, G.: Industrie- und Arbeitssoziologie, 3rd edn. Oldenbourg Wissenschaftsverlag, München (1995)
41. Monsell, S., Yeung, N., Azuma, R.: Reconfiguration of task-set: is it easier to switch to the weaker task? Psychol. Res. **63**, 250–264 (2000)
42. Morse, J., Lorsch, J.: Beyond theory. Harv. Bus. Rev. **48**(3), 61–68 (1999)
43. Moser, R., Janes, A., Russo, B., Sillitti, A., Succi, G.: Prom: Taking an echography of your software process. In: Proceedings of the XLIII Congresso Annuale AICA. Forum Editrice Universitaria Udinese, Udine (2005)
44. Rico, D.F.: ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers. J Ross Publishing Series. J. Ross Publishing, Boca Raton (2004)
45. Rogers, R.D., Monsell, S.: Costs of a predictable switch between simple cognitive tasks. J. Exp. Psychol. **124**, 207–231 (1995)
46. Rubinstein, J.S., Meyer, D.E., Evans, J.E.: Executive control of cognitive processes in task switching. J. Exp. Psychol. Hum. Percept. Perform. **27**(4), 763–797 (2001)
47. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, integrating and analyzing software metrics and personal software process data. In: Proceedings of the EUROMICRO Conference,

New Waves in System Architecture (EUROMICRO). IEEE, Belek-Antalya (2003)
48. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Measures for mobile users. In: Al-Ani, B., Arabnia, H.R., Mun, Y. (eds.) Proceedings of the International Conference on Software Engineering Research and Practice (SERP), vol. 1. CSREA Press, Las Vegas (2003)
49. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Non-invasive measurement of the software development process. In: Orso, A., Porter, A., (eds.) Proceedings of the International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS). IEEE, Portland (2003)
50. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Measures for mobile users: an architecture. J. Syst. Archit. **50**(7), 365–444 (2004)
51. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Measuring the architecture design process. In: Arabnia, H.R., Reza, H. (eds.) Proceedings of the International Conference on Software Engineering Research and Practice (SERP), vol. 1. CSREA Press, Las Vegas (2004)
52. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Monitoring the development process with eclipse. In: Srimani, P.K., Abraham, A. (eds.) Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC), vol. 2. IEEE, Las Vegas (2004)
53. Stevens, S.S.: On the theory of scales of measurement. Science **103**(2684), 677–680 (1946)
54. Stevens, S.S.: Mathematics, measurement and psychophysics. In: Stevens, S.S. (ed.) Handbook of Experimental Psychology. Wiley, New York (1951)
55. van Solingen, R.: Measuring the roi of software process improvement. IEEE Softw. **21**(3), 32–34 (2004)
56. van Solingen, R., Berghout, E., van Latum, F.: Interrupts: just a minute never is. IEEE Softw. **15**(5), 97–103, (1998)
57. Viertl, R.: Einführung in die Stochastik: Mit Elementen der Bayes-Statistik und der Analyse unscharfer Information, 3rd edn. Springers Lehrbücher der Informatik. Springer, New York (2003)

Part III
Lean Software Development in Action

The third part illustrates how we combine the tools presented in the second part to enact Lean Software Development. In particular Chap. 11 presents the experience we gained in applying the described technologies to help teams to implement Lean ideas in their software development process.

Chapter 10
The Integrated Approach

Again, a beautiful object, whether it be a living organism or any whole composed of parts, must not only have an orderly arrangement of parts, but must also be of a certain magnitude; for beauty depends on magnitude and order.

Aristotle, The Poetics, VII

Uli sat down and let his body being absorbed by the comfortable chair that J supplied him. Needless to say, most of the people in the room liked him. Only Ari looked disappointed and started again attacking: "Could you please tell us, or, at least, summarize us in clearly understandable language what you want us to understand with your presentation?" J laughed: "What else do you want? A lecture? But, Uli, let us make this old boxer happy. Recap him in few words the essence of the story."

"OK Ari, here we are. To make the approach working, we need three key ingredients. The first is our ability to organize the measures and to give them meaning. Measuring alone is not enough. There are many paradoxes of measures." "Uli, give me a break" said Ari "this is not a university course, here we are in real business! Measuring it not easy but is doable. For instance, if I want to get fit I measure my fitness—it is as simple as that! And I do the same with software." "Ari, you are smart indeed." laughed Uli "but, how would you measure your fitness?" "Uli, I measure my weight! I do not need a nobel prize to understand this." He laughed and everyone in the room also laughed. Also Uli laughed and then continued. "Interesting, you are really smart! And so you will find that the best is not to exercise! After exercising you will initially gain weight, because your muscle will get stronger and absorb water. Now you will answer me that you will also measure the calories consumed, and here again, if I measure the consumed calories is better to stay fat, so I consume more calories: the energy consumption is mv^2, mass times the square of velocity, remember Ari, so the higher the mass the higher the consumption!" Ari became angrier and angrier but did not say a word. Uli continued "Therefore, it is important to build a system where we can organize the measurements in a way that make sense, starting from our business goals, then translating the business goals into technical goals, and then down defining suitable questions to ask on whether we are achieving the goals, and lastly to define measurements that would support answering the questions, which, once answered,

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_10

will tell us if we are achieving our goals. In the case of fitness, I would propose that we ask ourselves whether we are getting fitter and more effective in running. For the first question I would measure not only our weight but also our heart rate after 30 minutes of jogging, and the fat material in our body. For the second question I would measure the time it takes to us to run 5 kilometers. So, by using these measures, we will first answer our questions, and then, using the answers, we will be able to determine if we are moving toward our goal. Ari, is this crystal clear?" Ari did not say a word and Uli took this as a yes.

"The second ingredient is that we need to take advantage of the experience we get from other projects, not simply running post-mortem sessions, but really collecting the experience also in quantified forms way and determine how such experience can be reused later on. The best is if this can be done directly by the team members themselves, but sometimes an external support can be beneficial.

The third ingredient is an effective measurements program, with the constraint that, once installed and configured, this measurements program should not in any way require any significant intervention from the side of the developers, so that the data will be really and effectively collected. In other terms, the measurements program should be non-invasive."

J nodded. Everyone understood that it was a go and none objected to avoid getting one of the withering glance from J. The meeting was adjourned.

10.1 Introduction

We began our journey looking at Lean production and wondering how to transfer its ideas to software development. So far we collected many little pieces that we will assemble now.

As we explained in Chap. 2, the main goal of Lean production is to eliminate waste. The practices like just-in-time production, autonomation, standardizing work, avoiding inventories, Kanban, pull and not push, etc. are not a goal itself; they are means to achieve efficiency.

We are convinced that no other software development approach has internalized the concept of just-in-time as much as the Agile Methods. We therefore think that Agile Methods are intrinsically a Lean software development approach. What we see as problematic, as we say in Chap. 5, is that Agile was not based on a methodology based on measurement, but on a manifesto. Agilists did not learn the skills to derive a Lean approach; they were taught what to do to be Lean.

A comparison with mountaineering may help to understand the difference. Let us take the mountaineer Amy: she wants to climb the Ortler, a 3,905 m (12,812 ft) high mountain in South Tyrol, Italy. The route is difficult and requires skills in climbing, glacier hiking, and crevasse rescue techniques (see the route description on Fig. 10.1).

Amy wants to reach the top and learn basic climbing techniques. Therefore, she attends a course organized by the local mountaineers club to learn how to safely

[htbp]

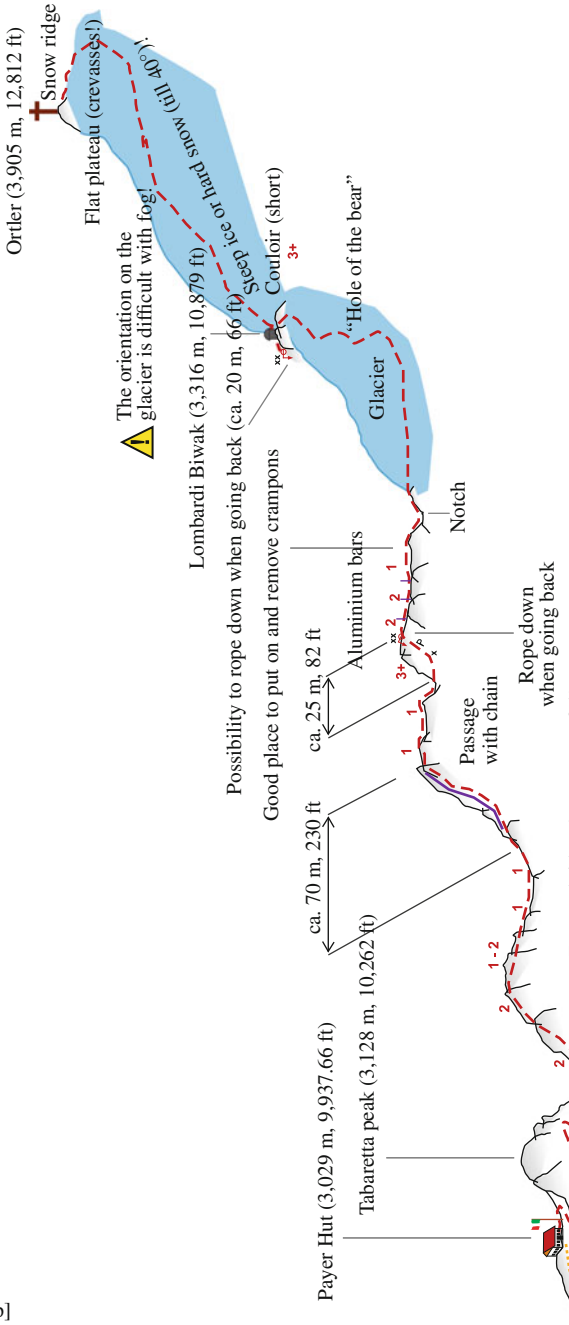


Fig. 10.1 The route to the Ortler (image © bergsteigen.com, courtesy of [16])

hike on glaciers and learns how to rescue herself and others from a crevasse. After having acquired the required skills, she goes to the Ortler and reaches the top safely since she has the competence to do it.

Henry is also a mountaineer, and he really wants also to reach the top of the Ortler, but he is not confident that he can acquire the necessary skills, and anyway, he has no time to train 1 year for that mountain. He hires the experienced mountaineer Kent to go with him. Kent takes care of everything and just tells Henry what he needs to do, where to hold himself, where to step, etc. Henry is not interested in the reasons for this instructions; he is just interested in the result. Henry makes it also to the top.

The difference between the two mountaineers Amy and Henry is that Amy will learn from her experience since she was understanding what she was doing and she could directly observe how well her choices worked. Henry executed the advice, he stepped on one place and not on another—as Kent said—but he did not understand the reasons behind these advices.[1] In the future, if Henry wants to reach the top of another difficult mountain, he will need Kent again.

In Chap. 6 we looked at existing proposals to implement Lean software development. Their ideas are a faithful translation of Lean Thinking into software development. However, in all their translations, we miss a part in which the future Lean software developer learns how to do the translation himself or herself, how to understand that he became Lean enough, how to understand what is missing, where to start, what is too much, etc.

In Chap. 2 we abstracted Lean Thinking using three aspects:

1. **Value**: methods that support the organization to focus on the understanding and maximization of the delivered value;
2. **Knowledge**: methods that focus on the creation of a shared understanding of the know-how, know-where, know-who, know-what, know-when, and know-why within the company;
3. **Improvement**: methods that help to instill a culture of constant improvement.

As we described in Chap. 6, several authors have translated Lean Thinking into software development [9, 18, 19, 21]. A software developer that wants to do that himself or herself has to learn how—to use the metaphor we used above—to climb the mountain alone, i.e., how to learn how to focus on value, knowledge, and improvement himself or herself.

Such a "Lean software developer" that does not need the gurus must have the following skills [3, 4]:

- The Lean software developer is able to study the software development process to identify waste and focus on creating value using measurement. Such measurement is non-invasive (see Chap. 9) to minimize non-value-adding activities [10, 11].

[1] We do not want to say that Henry did not learn anything during his trip, but we assume that he learned much less than Amy.

- As we mentioned multiple times, software engineering is development and not production. The software discipline is evolutionary and experimental. Therefore, the Lean software developer is able to organize improvement organizing knowledge in a systematic way, using a method tailored to software development, such as the Experience Factory (see Chap. 8).
- The Lean software developer is able to package wisdom in a reusable way, for example, using the GQM$^+$Strategies approach to explain the know-what (the data to collect), the know-how (the goals to achieve), and the know-why (the strategies to pursue) (see Chap. 7). To collect wisdom in such a way allows to reuse it [1].

As the mountaineer Amy is able to climb the Ortler herself, the "Lean software developer" is able to plan and do the journey to a Lean software development process himself or herself.

To accomplish this, in this chapter we combine the five ideas presented in earlier chapters to what we call "Lean Software Development in action":

- Agile software development,
- Non-invasive measurement,
- GQM$^+$Strategies,
- The Experience Factory, and
- Lean Thinking.

As we see in Fig. 10.1, the route to the Ortler mountain requires several skills: from the so-called surefootedness (i.e., that one is able to walk over difficult terrain without stumbling or falling) to climbing, techniques to safely walk on glaciers, and to rescue somebody that fell into a crevasse.

In the previous chapters we explained each component we see as part of Lean software development separately; now we will see how they work together. We depicted the dependencies between each pillar as a concept map in Fig. 10.2. Do not be scared! In the following paragraphs we will depict each part of this concept map.[2]

In the following paragraphs, we will explain each part. On the top left corner, the **software development process** is further divided into three concepts: **activities**, employ **resources**, and produce **artifacts** (see Fig. 10.3).

The value stream is continuously monitored using **non-invasive measurement**. The **measures** so collected describe the **software development process** and are then interpreted by the **GQM$^+$Strategies model** (see Fig. 10.4). The GQM$^+$Strategies model is updated when needed, i.e., when one aspect, e.g., an organizational goal or a strategy, changes.

[2] A concept map is a graphical representation of the relationships among a set of concepts [17]. It can be also used as a "knowledge elicitation technique which stimulates learners to articulate and synthesize their actual states of knowledge during the learning process [14]."

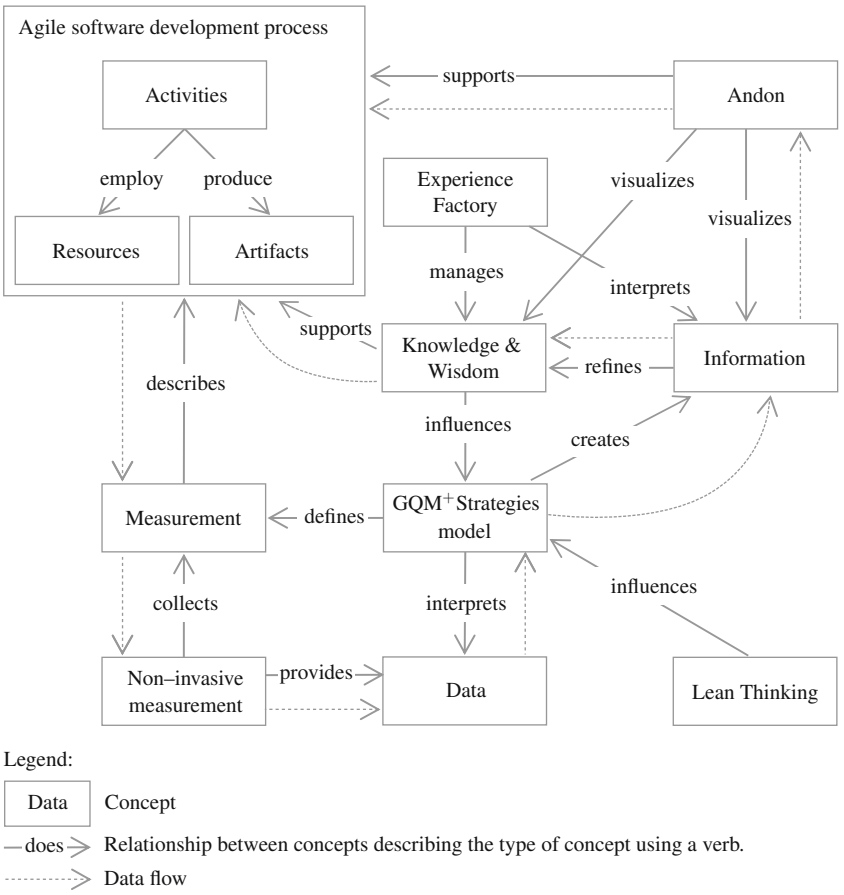


Fig. 10.2 Concept map describing the interactions between the building blocks of Lean software development

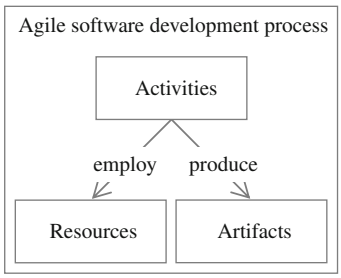


Fig. 10.3 The concepts **software development process**, **resources**, and **artifacts**

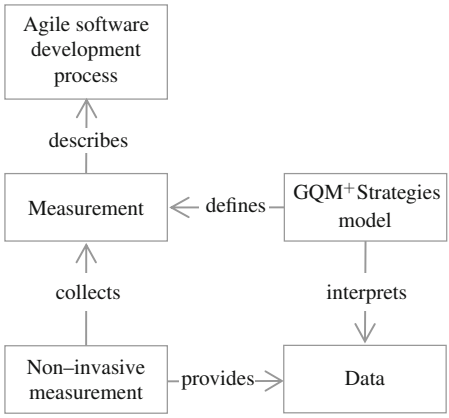


Fig. 10.4 The concepts to monitor the software development process

For an organization or a team, **Lean Thinking** is a strategy that is operationalized using the **GQM+Strategies model**, which furthermore defines which **measurements** are needed to understand how to control and improve the **software development process** (see Fig. 10.5).

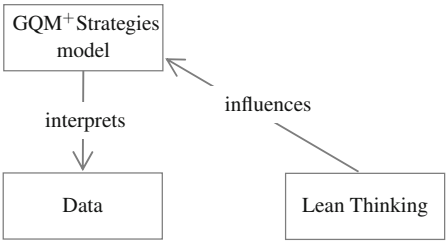


Fig. 10.5 The concepts to monitor the software development process

Once collected, the collected data can be converted to **information** by interpreting using the **GQM+Strategies model**. Let us assume we measure the cyclomatic complexity [15] of a method and obtain the value 20. Up to now, "20" is just data. Let us further assume that in the **GQM+Strategies model**, this measurement is linked (through a question) to a measurement goal that wants to analyze if the testing coverage of our most critical test cases is improving.

Using the **GQM+Strategies model**, we understand why we collect this number and how to interpret it. It is transformed into the **information** "the testing coverage of our most critical tests is too low" or "the testing coverage of our most critical tests increased by 10 % compared to last month."

The Experience Factory takes that **information** and collects and elaborates **knowledge and wisdom**, i.e., lessons learned, field-tested process definitions,

training and consulting material, checklists, frequently asked questions, software patterns, and so on.

The **Experience Factory** provides to the software development process experience and wisdom about past projects and advice for upcoming issues. **Activities** that are identified as non-value adding and not needed (see Chap. 2) by the **Experience Factory** are constantly removed (see Fig. 10.6).

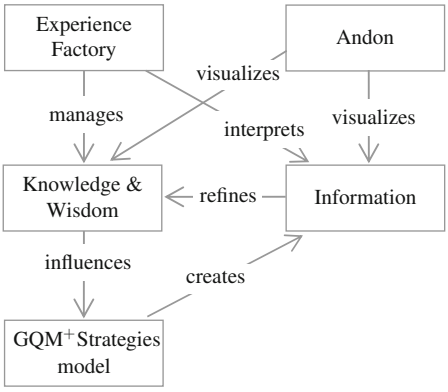


Fig. 10.6 The concepts to interpret and visualize the collected data

The **Experience Factory** provides a platform for team members to diffuse their knowledge and give them possibilities to influence current software development practices.

The **wisdom** collected by the **Experience Factory** is used to improve the measurement model, i.e., the goals of the team and the way how the goals are measured.

Andon is finally used to support the software development process closing the feedback loop and providing **information**, **knowledge, and wisdom** when needed.

10.2 The Role of Autonomation

When we discussed autonomation in Chap. 6, we already stated that its application in software is not trivial since software is invisible. We cannot directly see if something is wrong, and we cannot construct a machine so easily that checks the quality as in the picture below.

The Lean software development process we envision in this chapter implements autonomation. Autonomation in software engineering cannot be as exact as the one encouraged by Lean production. As said before, software development is evolutionary and experimental. It is largely human based so that software development is less

predictable than production, model building is more difficult, the models are less accurate, and we have to be cautious in the application of the models [5].

Autonomation consists of a mechanism to detect a problem and a mechanism to notify everybody that an error was detected. We use the measurements collected through non-invasive measurement together with the GQM$^+$Strategies model as the mechanism to detect a problem. The interpreted data about the current development process is studied, compared to expected values, analyzed to identify anomalies, etc. The GQM$^+$Strategies model is used to understand if the measurement goals are being achieved or not.

The second aspect of autonomation is a mechanism to notify all stakeholders about the problem. In Lean production, signal boards called "Andon" are used to notify everybody about the status of the production and to trigger countermeasures [2] (see Fig. 10.7).

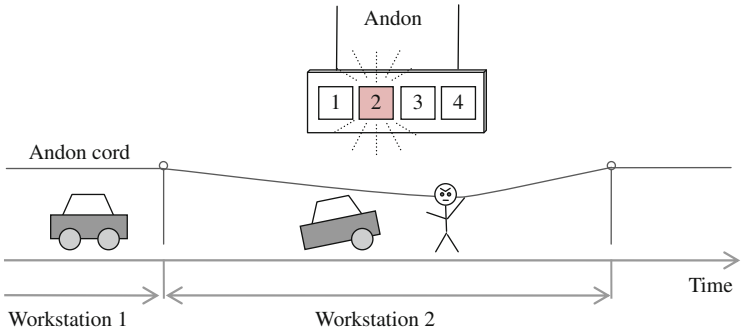


Fig. 10.7 The Andon notifies everybody of the state on the production line

The updated concept map of Lean software development is depicted in Fig. 10.8. We marked the concept responsible of notifying stakeholders about the current status of the project with "mechanism to notify" and the concepts responsible to detect problems with "mechanism to detect a problem."

In Lean production every worker that identifies a problem has the right to stop the production line to prevent further damage and waste of resources. To implement this concept in software development, an example is the use of "check-in policies" in source control systems, which define conditions that a developer has to meet to check in modified code.

In our concept map, the information to decide whether to stop software development or not is visualized in the Andon, gained through the interpretation of the collected data using the GQM$^+$Strategy. The interruption of the software development line cannot be accomplished as in a car factory. It would be an interesting experiment to install a button that turns off all computers at once in case of need.

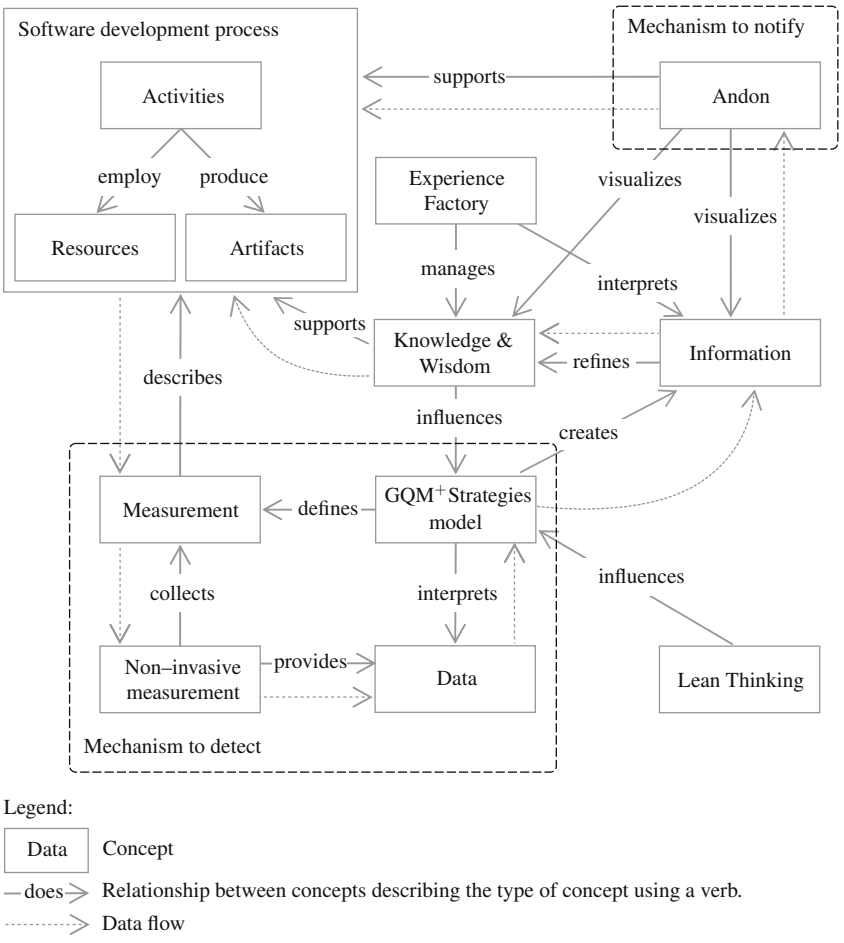


Fig. 10.8 The concepts responsible for Autonomation in Lean software development

Stopping the software development line is something that all team members have to respect with their behavior; it cannot be enforced, since software development is a socio-technical system.

An interesting example of how the "software development line" can be stopped is the translation of the Kanban concept into software development. As in Lean production, in software development, Kanban is an instrument to minimize work-in-progress inventory. It achieves that by setting a maximum number of work items in each development phase. If one phase has reached the maximum number of work items, it is not allowed to move any additional item into that phase. Practically, this stops the entire software development process. As Lean production, the idea is that all workers come and help to solve the issue before they can continue their work.

Henrik Kniberg and Mattias Skarin created the following comic strip that explains how the Kanban idea can be applied within software engineering [13].

At the beginning all team members are at their place: we see a task board consisting of five columns: "Backlog," "Selected," "Develop," "Deploy," and "Live!." The actors in this comic are customers (in Fig. 10.9 on the left under the "Backlog" column), developers (in Fig. 10.9 in the middle, under the "Develop" column), and system administrators (responsible for deployment; in Fig. 10.9 on the right, under the "Deploy" column).

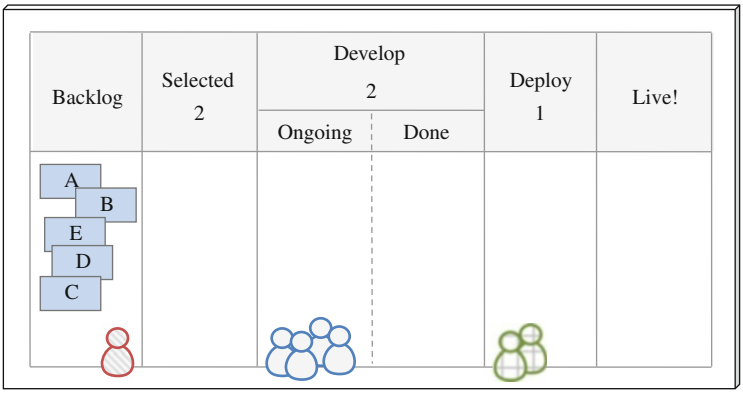


Fig. 10.9 The initial situation [13]

The usual path of a requirement is as follows: the customer puts planned, but not yet scheduled tasks into the "Backlog" column. The "Backlog" column is owned by the customer; he can decide what stays there, how it is ordered, etc. When the customer decides that a task should be developed, he moves it into the Selected column. As soon as two developers (the programmers in this comic develop in pairs) are available, they pick up the task from "Selected," move it to "Ongoing," and start to work on it. When a task is done, they move it to "Done." From there, the system administrators (also working in pairs) pick it up, move the task to "Deploy," and work on the deployment. Finally, as they are done with the deployment, they move the task to "Live!."

In Fig. 10.10 the customer selects two items that developers should take and implement.

In Fig. 10.11 we see that two start to work on A and two developers on B. The customer already selected the next two requirements to develop, C and D.

In Fig. 10.12, the two developers working on A finished their task and move it to Done. The system administrators see that and prepare to work on A. In the meantime the customer is thinking which requirements to choose as next steps.

While the system administrators work on deploying A, in Fig. 10.13, the developers pick C and start to work on that.

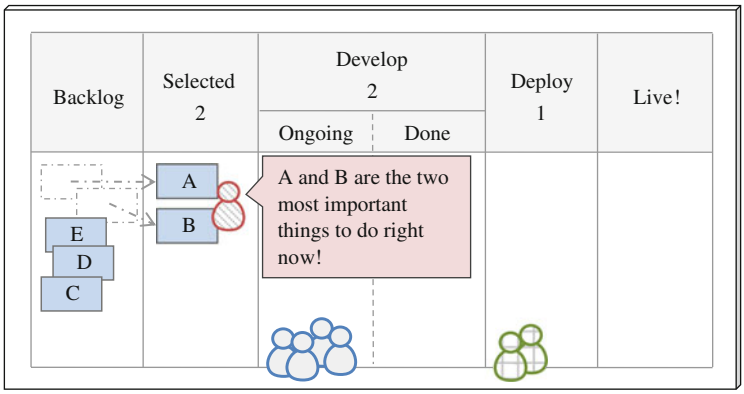


Fig. 10.10 The customer selects next tasks to develop [13]

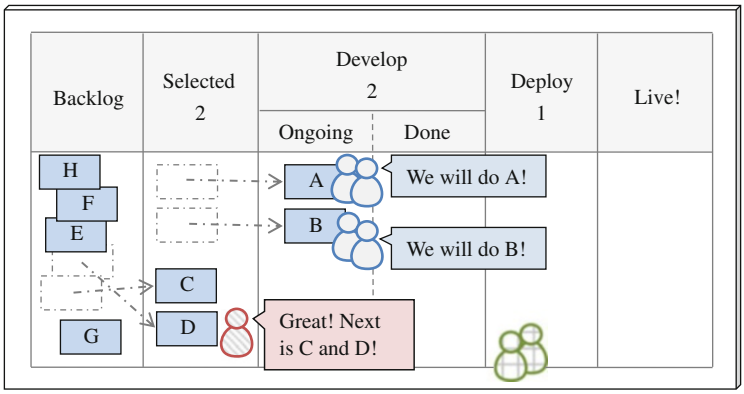


Fig. 10.11 Developers are working on A and B [13]

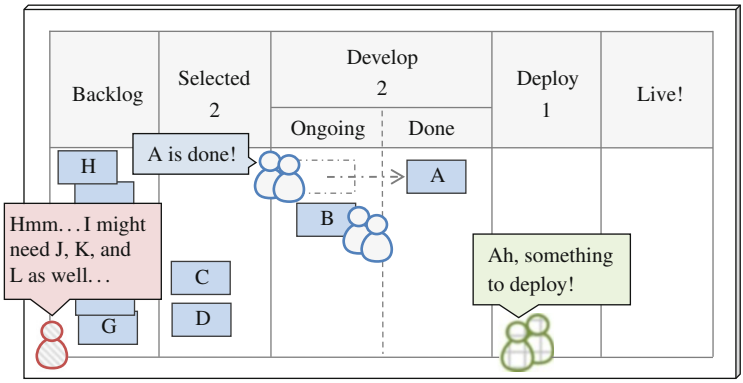


Fig. 10.12 The system administrators begin working on A [13]

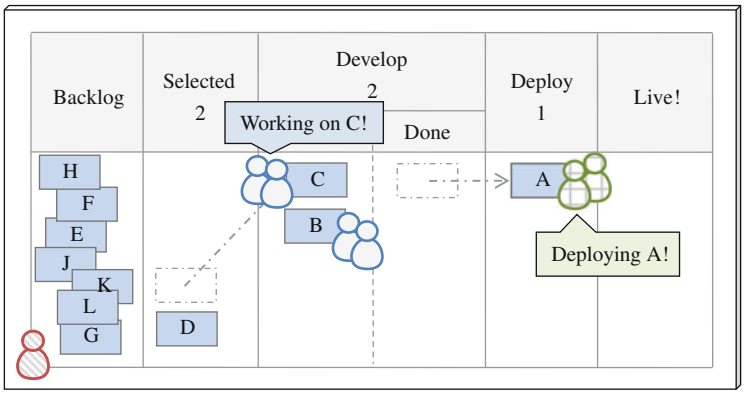


Fig. 10.13 The developers begin working on C [13]

Up to this point everything worked well, but now in Fig. 10.14, the deployment team has problems deploying A.

The rest of the team continues to work as planned, the developers finish to develop B, and the customer is thinking what to develop next.

It is also interesting that in this scenario, the developers do not care that A cannot be deployed.

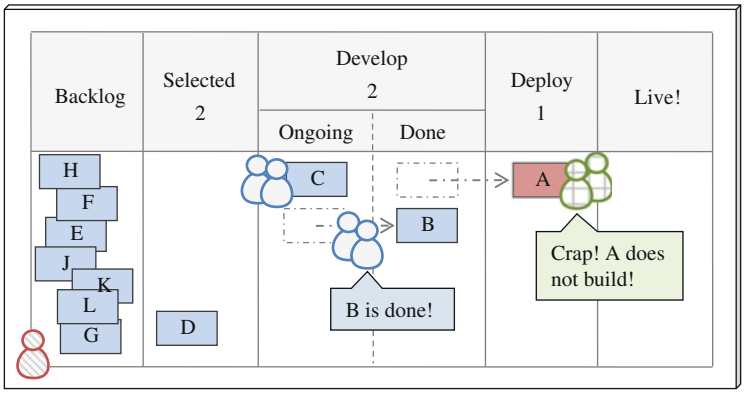


Fig. 10.14 Component A cannot be deployed [13]

In Fig. 10.15, the developers that just finished developing B want to pick a new task (D), but they cannot do that because the Kanban limit is reached.

The Kanban limit is displayed at the top of some columns and sets the maximum amount of tasks that can be in that column at the same time. It limits the amount of work-in-process items: without that limit, the developers would pick D and continue programming, ignoring that task A cannot be deployed.

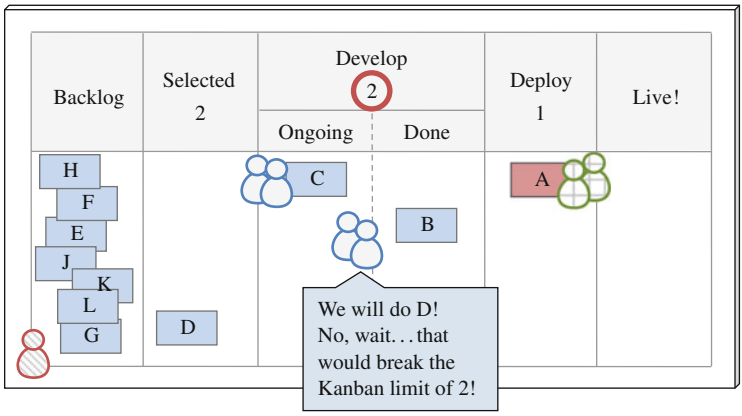


Fig. 10.15 The Kanban limit blocks developers from ignoring the deployment issue [13]

In Fig. 10.16, the pair of developers that cannot proceed goes to the bottleneck of the process—the pair of system administrators—to help out.

Kanban assumes here that the developers—since they actually wrote the code—are able fix the issue faster than the system administrators. This is why it makes sense to stop the work of skilled specialists, the developers, until the problem is solved.

Until now, the customer did not notice that there is any problem going on; he continues his work and picks item K to be developed next.

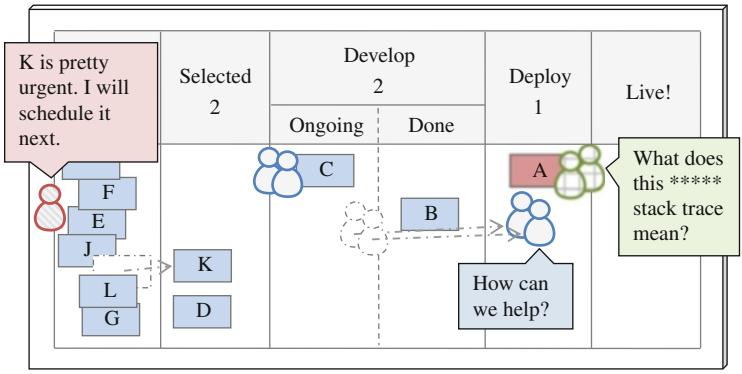


Fig. 10.16 The first developer pair helps to fix the deployment issue [13]

In the next figure (see Fig. 10.17)—while task A is still not deployed—the first team of developers finishes C and the customer would like them to begin with the next task K, but they realize that they have to help the others to fix the deployment of A since the Kanban limit of two is still stopping them to begin the development of a new task.

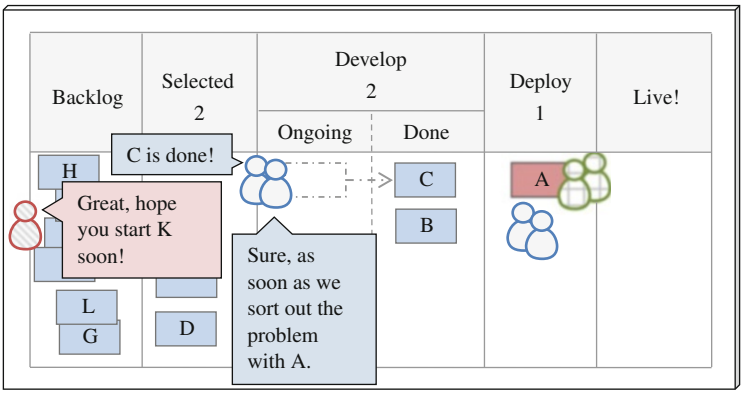


Fig. 10.17 The second developer pair finishes to develop C and goes to help fix the deployment issue [13]

The situation now escalates: the second pair of developers goes to help the others and—since they found the cause of the problem and are working on it—get the task to develop a test case that will signal that type of problem in the future.

At this point the entire development is stopped, and also the customer notices that there is some problem since he reached the maximum amount of tasks that he or she can put into the "Selected" column (see Fig. 10.18).

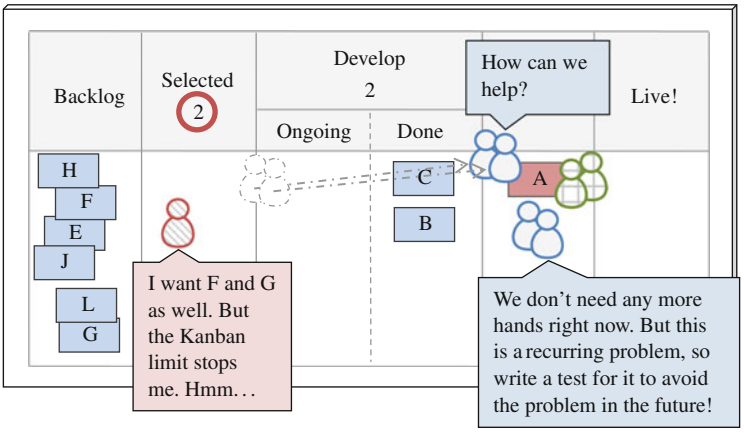


Fig. 10.18 The customer notices that he is not allowed to add requirements anymore [13]

The customer now understands that there is some problem with one of the previous requirements. In Fig. 10.19, the customer goes to the developers and asks if he can help.

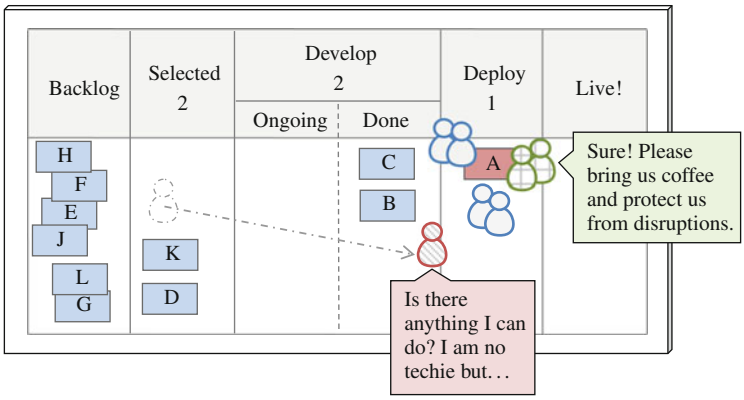


Fig. 10.19 The customer tries to help to fix the deployment issue [13]

Instead of being confronted with a delay at the end of the project, the customer can help redefining or simplifying the requirement now (similar to what we discussed in Sect. 3.5).

According to our experience, the customer is often able to change the requirement so that it provides the same value for him as the original requirement but is easier to implement. In this example, the customer cannot do much and the developers send him to bring coffee.

Figure 10.20 shows how the team thinks back about the times where work-in-progress inventory was not limited. It shows happy developers, system administrators, and a busy customer that is concentrated on identifying what provides business value for him.

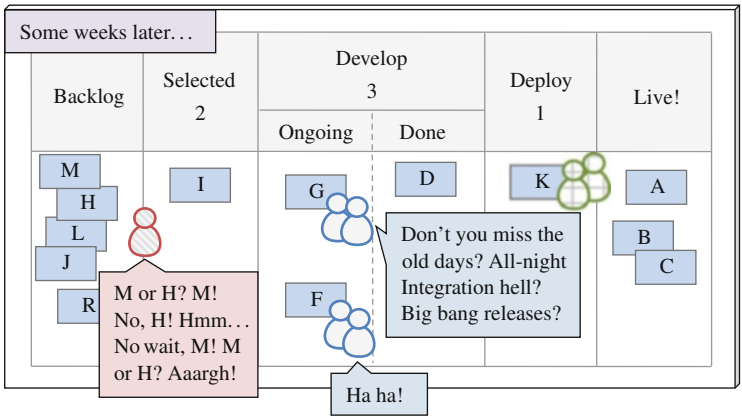


Fig. 10.20 The final scene [13]

This comic strip shows how Kanban implements the assumption of Lean production that it is better to stop the line and fix the problem instead of continuing and solving the problem later. Moreover, the strip shows how this idea can be implemented in software development. The Kanban limits define the size of work-in-progress inventory that the team considers acceptable and have to be adapted to the team size.

Jidoka means not allowing defective parts to go ahead in the development process. Therefore, we need a clear criteria, or rule, to decide whether a software artifact is ready to proceed to the next production step. To evaluate these criteria, suitable data have to be collected.

Interrupting production is done to prevent further damage and waste of resources. Within software development an example hereof is the use of "check-in policies" in source control systems, which define conditions that a developer has to meet to check in modified code.

As a mechanism to "stop the production line," we use the Andon concept in the form of a dashboard that visualizes the problem. This creates visibility since every developer working on that project sees that there is a problem.

10.3 Closing the Loop with an Andon Board for Lean Software Development[3]

Ideally, dashboards inform about the state of a system and advise the user what to do. Unfortunately, many dashboards are designed as mere data displays instead of helping the user to make better decisions.

The term dashboard was already used in the nineteenth century when it represented a board in front of a carriage to stop mud from being splashed (dashed) into the vehicle by the horse's hoofs [6] (see Fig. 10.21). Later, in cars, the dashboard was used to inform the driver about the status of the car. The indicators on the dashboard help the driver to operate the car, and the colors of the indicators show how urgent the matter is. Red indicators typically mean that the problem is serious and that some action is required immediately.

Red indicators such as "check engine" or "low oil" require that the driver halts the car immediately to stop further damage (see Fig. 10.22). Safety issues, such as a non-working air bag, are also shown using red lamps. Yellow indicators show that some action is required soon, such as the yellow low fuel light. Green indicators inform the driver that something is turned on, such as the low beam lights.

[3]A version of this section originally appeared in the January 2013 issue of Cutter IT Journal, published by Cutter Consortium www.cutter.com, © 2013 Cutter Consortium. For more information, please visit http://www.cutter.com/itjournal/fulltext/2013/01/itj1301c.html. All rights reserved. Reproduced with permission.

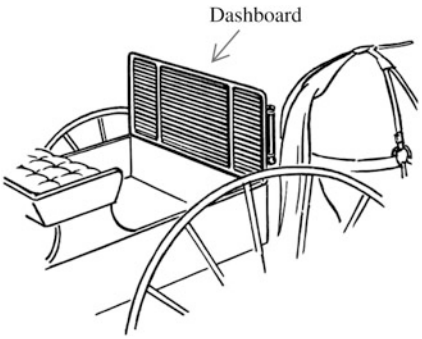


Fig. 10.21 The dashboard of a horse carriage (image courtesy of Pearson Scott Foresman)

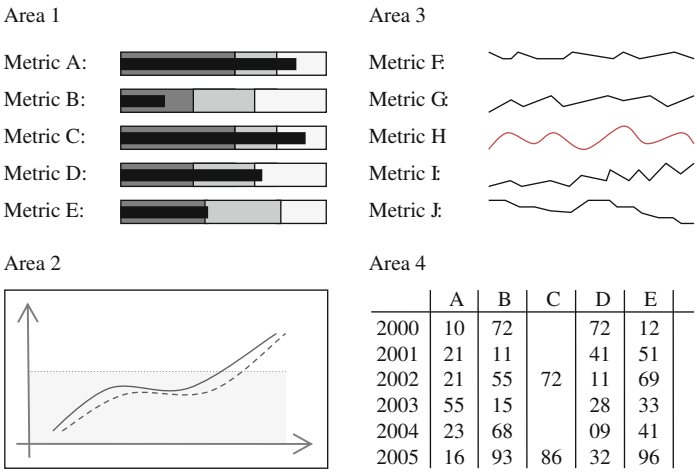
Fig. 10.22 Typical indicators in a car dashboard.

The dashboard is designed so that the driver is able to intervene if the correct functioning of the car is at risk. It is, so to speak, aligned to the business goal of the driver. It helps the driver to achieve the goal of driving from point A to point B. In an organization, the term "dashboard" is used to describe a system that visualizes data useful for decision making [8]. Dashboards, as in the car, have the goal to inform a user, but not to distract from the actual task. Therefore, the data in dashboards is summarized using charts, tables, gauges, etc., as in Fig. 10.23.[4]

We use the term "dashboard" for a data visualization technique that aims to inform and alert a user about the current state of the system. It operates in a "push" mode, and it aims to distract and gather the attention of the user for a short moment to inform about something important [12]. We see a dashboard as the opposite of a visualization in which the user operates in "pull" mode, where he is examining the visualized data, so to say "dives" into the data to explore, to understand the reasons behind a problem, and to solve an issue. To distinguish dashboards from visualizations that require interaction, we call the latter an information display.

These two visualization techniques do not exist in isolation. Frequently, a dashboard allows to switch from the alerting mode into exploration mode and allows the user to analyze and study the data behind the alert.

[4]Some authors (e.g., [7]) distinguish between dashboards and scorecards, depending if the data measures performance (dashboard) or charts progress (scorecard). We consider the terms dashboard and scorecard as synonyms.



	A	B	C	D	E
2000	10	72		72	12
2001	21	11		41	51
2002	21	55	72	11	69
2003	55	15		28	33
2004	23	68		09	41
2005	16	93	86	32	96

Fig. 10.23 A typical dashboard [8]

The Balanced Scorecard described in Chap. 3 can be either dashboard, if it conveys information in the "push" mode, or an information display, if it operates in "pull" mode.

To be useful, a dashboard supports its users to fulfill their goals. Unfortunately, many dashboards are not designed to do that [8]. Some dashboards are designed to visualize as much data as possible, to demonstrate the graphical abilities of the dashboard, to impress potential customers, to display all the available information, etc. In this case they are not dashboards, but information displays.

To obtain a Lean dashboard, i.e., a dashboard that—as Andon—indicates when the user has to intervene, we have to [12]:

- visualize the "right" data and
- choose the "right" visualization technique.

10.3.1 Visualizing the "Right" Data

To choose the right data, we visualize data that is obtained through the measurement model defined by the GQM$^+$Strategies model, which specifies which data we collect, together with the reasons why we collect it. For example, we could define that we collect McCabe's cyclomatic complexity of every method (what) because using that measurement, we can decide how much we have to test that code (why).

Once the collected data are linked to the reason why we need it, it is possible to correctly interpret the data and to reuse it for future projects since we are able to put it into the correct context [20].

The GQM$^+$Strategies measurement model describes what we call the "ideal dashboard": it measures the achievement of the organizational strategy, i.e., the business goals. It adds to measurements the context, meaning the reasons for collecting it and the business strategy that justifies it [12].

It is not always feasible to elaborate a detailed GQM$^+$Strategies measurement model, for example, if the strategy is frequently changing as a reaction to a volatile market. It is up to the organization to decide which level of monitoring and which kind of information is worthy to collect and to visualize.

Typically authors distinguish between strategic, tactical, and operational dashboards. Seen from the perspective of the GQM$^+$Strategies measurement model, a strategic dashboard visualizes the achievement of the upper goals of the goal hierarchy; a tactical dashboard, the goals that aim to enact the strategy through the so-called tactics; and an operational dashboard, the most detailed goals at the bottom that measure the achievement of the tactics.

10.3.2 Visualizing Data "Right"

Dashboards can be designed in a variety of ways. There is not one right way; it depends on the requirements the dashboard has to fulfill. Generally speaking, aspects of technology acceptance are important, mainly the perceived usefulness and the perceived ease of use [22]. Important considerations are [8]:

1. The dashboard should help the user to understand the context of the data, i.e., state why this data was collected, how it should be interpreted, how we can use this data in future projects, etc.
2. The dashboard should help the user to understand the meaning of the data, i.e., choose visualizations that require a minimum of effort to get the conveyed message, be coherent in the chosen visualization strategies, allow the user to choose the level of detail of the data, etc.

To design a dashboard that "pushes" information to the user, i.e., captures his attention, we have to understand how much effort a user has to invest to see the dashboard. A dashboard should inform the user in unexpected, unforeseen situations about problems, anomalies, etc. Then, switching to exploration mode, it should support the user to explore the data, to filter and to search, to investigate the reasons that caused the data, etc.

To set up a dashboard that is used in a "push" scenario, we found the following considerations important [8, 12, 23]:

1. The user should see the dashboard without any effort. For example, in the car, the dashboard is built in such a way that it is in the range of vision of the driver. An organizational dashboard should be displayed on a monitor in the corridor or in the office where many are passing by. The information will be pushed to the

users without their active participation. An example for such a dashboard is the Andon board, used in Lean manufacturing and placed visibly so that everybody can see it (see Fig. 10.7).
2. The user should not need to interact with visualizations to understand the data. The charts have to be designed in such a way that an interaction is only necessary when the user switches into the "pull" mode, i.e., the dashboard got the attention of the user and he wants to investigate further.
3. Arrange the data to minimize the time needed to consult the dashboard. Place the same information always on the same spot. Allow the user to develop habits, e.g., every morning, when passing by with the coffee in the hand, she can check the current size of the error log that is displayed in the upper right corner of the dashboard.
4. Guide the attention of the user to indicate important information. There are different mechanisms that draw the attention of the user. If they are overused, the user neglects them. For example, if everything on the dashboard is blinking, the user will ignore it.
5. Since we want that the users look at the dashboards by choice, also aesthetical considerations can increase the interest for the user to look at the dashboard.

The next step is to choose the right visualization techniques. For dashboards we focus on visualizations that minimize the time needed to understand what has to be communicated instead of trying to convey as much information as possible.

To highlight important data, we use "pre-attentive processing," which we will explain now. Researchers have identified different graphical properties that are processed pre-attentively and grouped them into form, color, motion, and spatial position [23]. Pre-attentive processing elements have the advantage that they are processed (i.e., understood) faster than not pre-attentive elements [23].

An example is provided in Fig. 10.24: both boxes a and b contain numbers. If we look at the box on the left (see Fig. 10.24a) and try to count the number of 3s, we have to process the numbers sequentially, i.e., we have to look at each number separately and decide if it resembles the form of the number 3.

It is much easier to count the number of 3s in the right box (see Fig. 10.24b) than in the left box (see Fig. 10.24a). This is because we identify color differences faster than the meaning of a symbol.

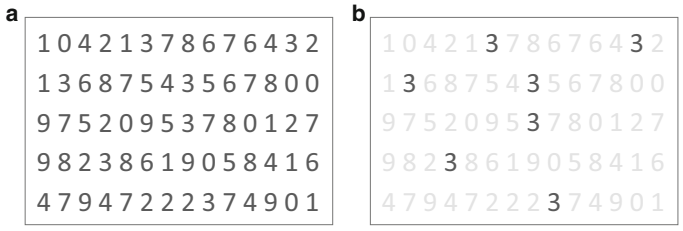


Fig. 10.24 How many 3s are present in the boxes?

How strong something is noticed pre-attentively depends from how different the highlighted element is from the others and how different the other elements are among each other. Moreover, combining two pre-attentively processed properties (such as color and shape in Fig. 10.25) cannot anymore be processed pre-attentively and requires again a sequential processing of the information.

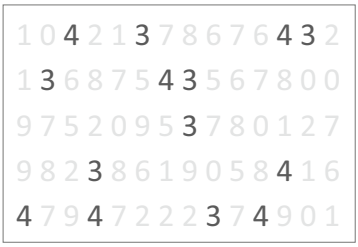


Fig. 10.25 How many 3s are present in the box?

The information visualized on the dashboard is based on the measurement strategy operationalized as GQM$^+$Strategies measurement model. Single measures are operationalized as GQM measurement models like the one in Fig. 10.26.

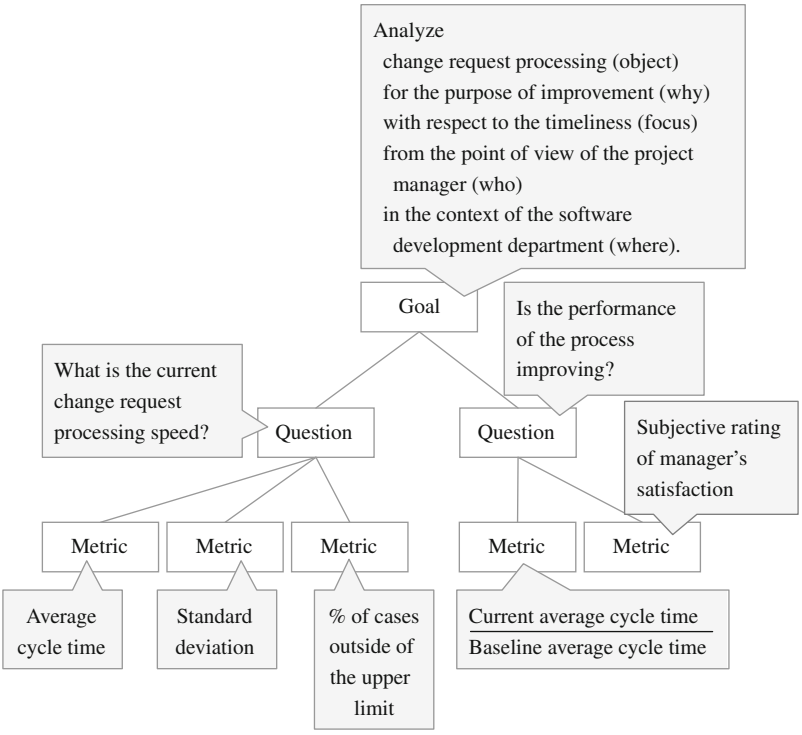


Fig. 10.26 GQM model to evaluate the timeliness of change request processing

To visualize the measurements, we used colored tiles that visualize the outcome of the measurement, the trend (if the value is decreasing, stable, or increasing over time), and the classification of the measurement as "good" (green), "warning" (orange), or "critical" (red).

It is also possible to use shades of gray, patterns, or a different line thickness if the dashboard has to be accessible to color-blind people. In the United States, the Section 508 Amendment to the Rehabilitation Act of 1973 states that "Color coding shall not be used as the only means of conveying information, indicating an action, prompting a response, or distinguishing a visual element." This legal requirement applies only to federal agencies in the United States, but it is a good idea to add a second distinctive element, such as line thickness and a pattern, or to choose colors that are so different that they become distinctive shades of gray for a color-blind person.

If a tile (like the one in Fig. 10.27) is red, that means that the measurement is classified as "critical" and requires the attention of the team. The name of the measurement is "A" and the current value is 10. The arrow shows that from the last time the measure was evaluated, the value has increased.

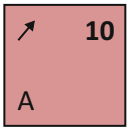


Fig. 10.27 A tile that represents a measurement outcome [12]

10.3.3 Putting the Pieces Together

Figure 10.28 depicts a dashboard based on tiles. Each tile represents the outcome of a measurement. The measurements are grouped with their questions and goals. We use colors to depict the status of a measurement to allow users to look at the dashboard using pre-attentive processing. The idea is that green tiles do not need to be read; they can be ignored. Orange and red tiles require attention; developers and managers should have a look on them.

The dashboard represents the measurement goals connected to one business goal. In our dashboard implementation, the values displayed within a measurement tile can either originate from actual data or it can be the result of another measurement goal. Figure 10.29 illustrates how the dashboard allows users to navigate through the GQM$^+$Strategies measurement model. In step 1 the user clicked on the red tile related to the business goal "Increase reputation." That tile is not calculated using actual data, but it is the result of a measurement goal that belongs to the business goal "Increase software quality." Therefore, after clicking on the tile, the users obtain the GQM model for the business goal "Increase software quality." Moreover, the users see why the tile of the business goal "Increase reputation" was marked as "critical": one of the tiles of the business goal "Increase software quality" is critical.

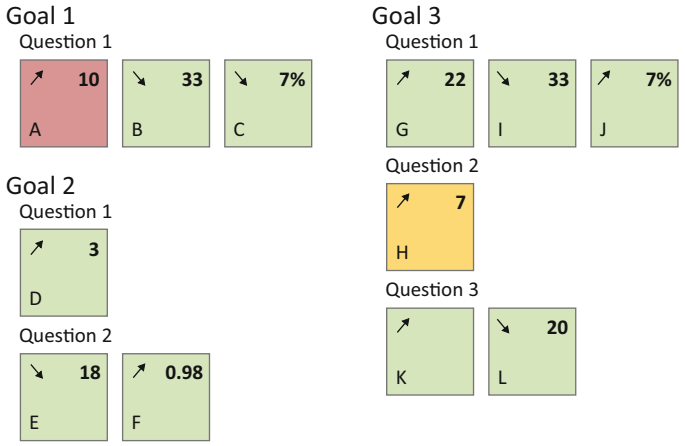


Fig. 10.28 A dashboard showing the measurement outcomes as tiles [12]

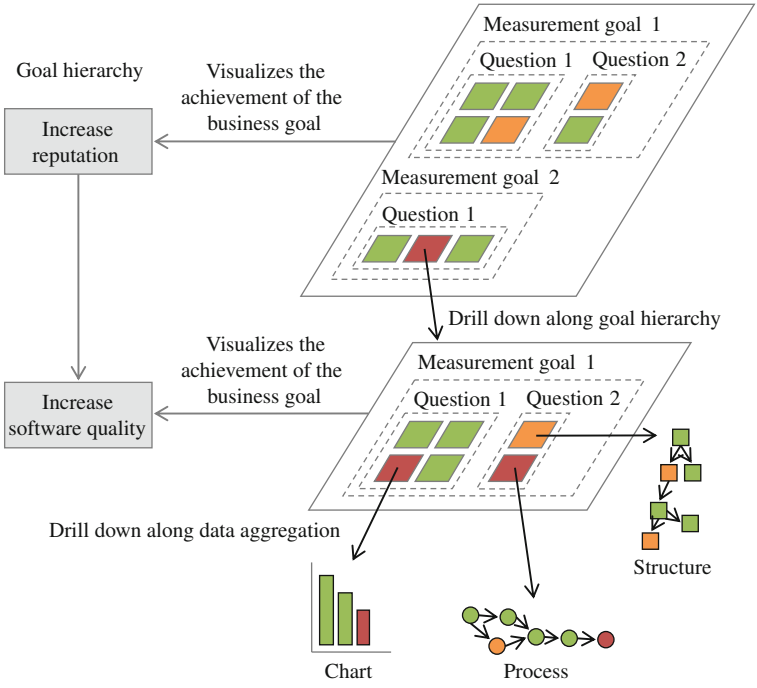


Fig. 10.29 Possible drill-down path in the dashboard

The above described navigation from the generic to the detail (from a visualization that summarizes to the data behind that summarization) is called "drill down." In Fig. 10.29, a user drills down from the first tile that caught his attention, i.e., he switches into what we called above "exploration mode." He is interested in finding the reasons why the dashboard indicates a problem and how to solve the problem. He drills down along the goal hierarchy, i.e., he sees the measurement goals, questions, and measurements that visualize the achievement of a lower-level business goal.

It is not necessary to drill down along the goal hierarchy; also other forms of navigation from generic to detail are possible, e.g., as depicted in Fig. 10.29, from an aggregated measurement to a visualization or table that displays the data behind the aggregation. The original data can be shown, for example, as a table, as a chart, a petri net, a hierarchical structure, etc.

10.4 Summary

This chapter presented how we combine the concepts of Agile software development, non-invasive measurement, GQM measurement, the Experience Factory, and Lean Thinking to obtain Lean software development, i.e., the translation of Lean concepts into software engineering. As we said in Chap. 3, this translation can occur in different ways; nevertheless, we think that the presented technologies are essential to enact Lean Thinking in software engineering following the recommendations of Womack and Jones [24] already mentioned in Chap. 2:

1. **Specify value from the standpoint of the end customer**: understand what is valuable for the customer and why the customer is willing to pay money for a certain product or service;
2. **Identify the activities along the production process that contribute in creating what is valuable from the standpoint of the end customer**, i.e., identify all the steps in the value stream;
3. **Align the value-providing steps in a way that every product and service is built or provided along a simple, predefined path**, i.e., make the value-creating steps flow towards the customer;
4. **Start an activity only at the moment that it provides value to a concrete customer requirement**, i.e., let customers "pull value" from the next upstream activity;
5. **Pursue perfection**: continuously improve.

In the last part of this chapter, we presented the Andon concept as an instrument to communicate to the team the collected information, knowledge, and wisdom. We described an Andon concept that is based on the idea that the team members get informed without interacting with the visualization. We recommend to place the Andon in a clearly visible place but where it does not distract the team members (as, for example, in Fig. 10.30).

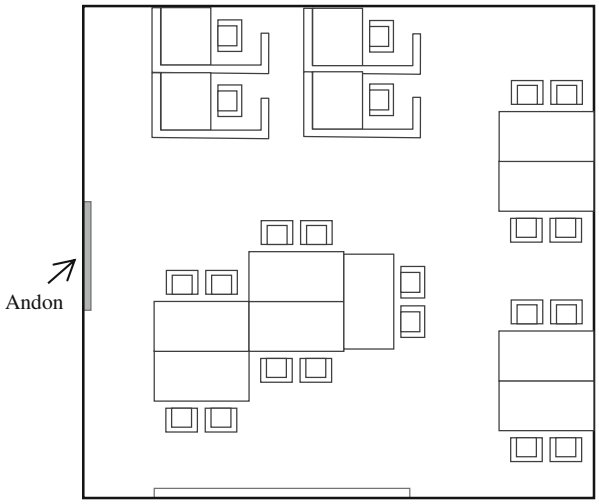


Fig. 10.30 Possible position of the Andon board in the Agile office space presented in Chap. 4

The presented technologies to measure, reason about, and improve the software development process are necessary to enact Lean Thinking in software engineering. The next chapter presents three case studies that report our experiences in applying them.

Problems

10.1. Each concept in Fig. 10.2 contributes to obtain a software development process that is able to determine its value stream, create knowledge, and improve. What types of data are handled by each concept?

10.2. Assume you set up a fantastic dashboard for your team. As you collect the data and visualize it, you notice that all the measurements show problematic values. You let the dashboard in place for some days and also show it to your collaborators, but nobody cares; everybody continues his job as if everything would be fine. What is going wrong?

References

1. Astromskis, S., Janes, A.: Towards a gqm model for is development process selection. In: Proceedings of the tarpuniversitetinė magistrantųir doktorantų konferencija. Kaunas University of Technology, Kaunas, Lithuania (2011)
2. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: Andon for dentists. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute, Boston (2013)

3. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: Implementing organization-wide gemba using non-invasive process mining. Cutter IT J. **26**(4), 32–39 (2013)
4. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: Supporting governance in disciplined agile delivery using non-invasive measurement and process mining. Cutter IT J. **26**(11), 25–29 (2013)
5. Basili, V.R.: The experience factory and its relationship to other improvement paradigms. In: Sommerville, I., Paul, M. (eds.) Proceedings of the European Software Engineering Conference (ESEC). Lecture Notes in Computer Science, vol. 717. Springer, New York (1993)
6. Collins: Collins English Dictionary — Complete & Unabridged, 10th edn. HarperCollins (2009). Online: http://www.collinsdictionary.com. Accessed 4 Dec 2013
7. Eckerson, W.W.: Performance Dashboards: Measuring, Monitoring, and Managing Your Business, 2nd edn. Wiley, New York (2010)
8. Few, S.: Information Dashboard Design: The Effective Visual Communication of Data. Oreilly Series. O'Reilly Media, Cambridge (2006)
9. Hibbs, C., Jewett, S.P., Sullivan, M.: The Art of Lean Software Development: A Practical and Incremental Approach. Theory in practice. O'Reilly Media, Sebastopol (2009)
10. Janes, A.: Providing decision-making support using non-invasive business process metrics collection. In: Biffl, S., Friedrich, G., Grünbacher, P., Succi, G. (eds.) Proceedings of the Alpine Software Engineering Workshop (ASEW). Institute for Systems Engineering and Automation, Heiligenblut (2004)
11. Janes, A., Russo, B., Succi, G.: Using non-invasive measurement techniques in agile software development: a swot analysis. In: Proceedings of the XLII Congresso Annuale AICA. AICA, Benevento (2004)
12. Janes, A., Sillitti, A., Succi, G.: Effective dashboard design. Cutter IT J. **26**(1), 17–24 (2013)
13. Kniberg, H., Skarin, M.: Kanban and Scrum — Making the Most of Both. InfoQ enterprise software development series. lulu.com (2010)
14. Kornilakis, H., Grigoriadou, M., Papanikolaou, K.A., Gouli, E.: Using wordnet to support interactive concept map construction. In: Proceedings of the International Conference on Advanced Learning Technologies (ICALT). IEEE Computer Society, Joensuu (2004)
15. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
16. Mitsche, E.: Ortler normalweg (2013). Online: http://www.bergsteigen.com/klettern/trentino-suedtirol/ortler-alpen/ortler-normalweg. Alpinverlag Jentzsch-Rabl. Accessed 4 Dec 2013
17. Novak, J.D., Gowin, D.B.: Learning How to Learn. Cambridge University Press, Cambridge (1984)
18. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley Professional, Boston (2003)
19. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, Upper Saddle River (2006)
20. Rombach, H.D., Ulery, B.T.: Improving software maintenance through measurement. Proc. IEEE **77**(4), 581–595 (1989)
21. Shalloway, A., Beaver, G., Trott, J.R.: Lean-Agile Software Development: Achieving Enterprise Agility. Lean-Agile Series. Addison-Wesley Professional, Boston (2009)
22. Venkatesh, V., Bala, H.: Technology acceptance model 3 and a research agenda on interventions. Decis. Sci. **39**(2), 273–315 (2008)
23. Ware, C.: Information Visualization: Perception for Design, 3rd edn. Morgan Kaufmann, Boston (2012)
24. Womack, J.P., Jones, D.T.: Lean Thinking: Banish Waste and Create Wealth in Your Corporation, 2nd edn. Free Press, New York (2003)

Chapter 11
Lean Software Development in Action

ché non fa scïenza,
sanza lo ritenere, avere inteso.
(There is no knowledge
without the ability to retain what has been heard.)

Dante Alighieri, La Divina Commedia, Paradiso, Canto V, 41-42

Uli took a big breadth, look at the watch—exactly 24 hours have passed since all this has started. He decided he was happy with his work and he deserved to devote some time to himself. Picked up Argo, his dog, and went jogging.

11.1 Introduction

The here reported cases are the result of three action research [9, 66, 72] initiatives in which we introduced aspects of Lean software development as described in this book within three teams.

Before we begin describing the three case studies, we describe how these case studies were carried out and evaluated, and how we introduced the software development teams to the developed measurement tools. This is needed to know how a case study has to be performed to be valid and which conclusions we can make based on them.

Ideally, research is seen as in Fig. 11.1: a somehow linear process that passes the phases of understanding the problem, devising a plan, carrying out the plan, and reviewing the solution [91].

Doing applied research about software development process models is not that easy. Sometimes we look enviously to our colleagues that study more theoretical aspects of computer science. They are often able to reduce the complexity of a problem ignoring many aspects that do not influence their problem.

As researchers, we create models. Models are simplified descriptions of the reality. We create such models so that we can reason about them. Typically, a researcher studies models to understand, evaluate, and improve them. To evaluate

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_11

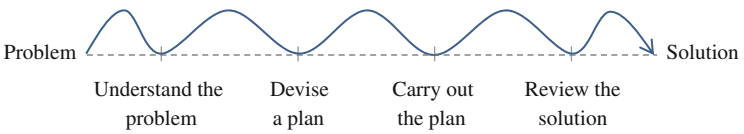


Fig. 11.1 An ideal, "rational" model of research

if the result is valid, the researcher will try to apply the model back to the reality, i.e., to verify if what he found out about the model is also valid in the reality.

The abstraction and simplification of the reality into models can be useful to understand the big picture, but it can also lead to wrong conclusions. For example, the waterfall model is the result of an approach to software engineering that ignores many sources of complexity. It assumes the case in which the requirements are clear to everybody, software developers know how to construct the desired software, and coordination problems do not exist.

In the previous chapters, we have discussed the objectives of Lean, i.e., flexible, iterative, just-in-time approaches. Such approaches work better than plan-driven, linear, "systematic" approaches in contexts that require frequent changes and adaptations, such as those we find in wicked problems. The same applies to research where the goal is clear, but the methodology to achieve it is influenced by many aspects, i.e., it is complex.

The introduction of Lean software development itself is a wicked problem that requires a research approach that is based on an iterative approach. We encountered the research about how to introduce Lean software development as in Fig. 11.2, an approach characterized by many iterations, repetitions, and errors.

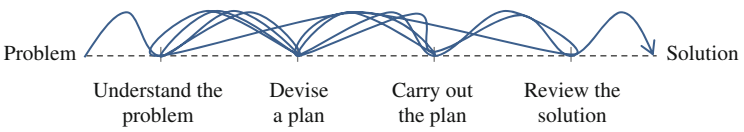


Fig. 11.2 A real model of research (sometimes referred to as the "Garbage can model" [20])

A research method that supports such an iterative approach is action research. Action research is a qualitative research approach, i.e., an approach that aims to get an in-depth understanding and the reasons behind the observed phenomena, studying them in their real-life context. To better understand what action research is, we first briefly discuss quantitative and qualitative research.

Both, quantitative and qualitative research, aim to investigate theories, i.e., explanations about some aspect of the world. A theory is a hypothetical story about why acts, events, structure, and thoughts occur [112]. A theory could be: "The code written on Monday's contains more defects than the code written during the rest of

the week. This is because programmers are still thinking about the weekend and are distracted from their work."

Theories should consist of [111]:

- **What**: which factors are part of the explanation of the phenomena of interest?
- **How**: how are the factors related?
- **Why**: what justifies the selection of factors and the proposed causal relationships?
- **Context**: what are the boundary conditions, the who, where, when, under which the theory is valid?

Empirical theories are explanations that are formulated and then tested against experience by observation and experiment [93]. To be scientific, such theories have to be falsifiable, that means that it is possible to find a "reproducible effect which refutes the theory" [93].

Theories can be studied using quantitative or qualitative research. To study a theory, we define hypotheses, which are (as theories) explanations about some aspect of the world, but while a theory is a collection of already accepted, consistent statements [8], a hypothesis is a proposition, a statement that has to be yet confirmed or rejected.

Quantitative research wants to infer characteristics about a population, based on the data gathered from a sample [112]. The researcher has to collect as much data as is necessary to generalize the findings with a small possibility of error. Non-invasive measurement, as we presented it in Chap. 9, is a quantitative method to evaluate the software development process. The GQM approach presented in Chap. 7 is a method to precisely define what we want to analyze to avoid making wrong conclusions (see below the concept of "construct validity"). A typical quantitative research approach is the experiment. For example, we might want to study how a treatment group reacts to a particular drug compared to a (not treated) control group. The goal of a researcher performing an experiment is to exclude everything he did not consider and that might influence the experiment and therefore falsify the conclusions he makes. This includes also himself: he has to obtain the data without influencing the observed phenomena, since this might falsify the results [8].

It is not always possible to study something quantitatively, for example, if statistical inference cannot be used since there are not enough items to study, or if it is not possible to bring the studied item into the laboratory or to interview it. In such cases qualitative research approaches developed methods to study the phenomena. It is important to remember that the outcome of quantitative and qualitative is not the same: qualitative research is not suited for the type of generalization done within quantitative research (statistical generalization), since it does not collect enough data about the studied subject [112]. It studies the reality in-depth to enlarge our understanding of it and to develop new (or modify existing) theories about it.

This type of generalization is called analytical generalization. The aspects of the reality we study are not selected because they are representative or typical; they are selected because we think that they will contribute to our understanding. An interesting aspect to study can be a typical situation but also the strange, exceptional,

or unique one. In both cases we want to evaluate if we can confirm the studied theory or not [112]. The exceptional case might even contribute more to our understanding than the typical one.

A typical qualitative research method is the case study, which we will now analyze further. A case study "investigates a contemporary phenomenon in depth and within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident. The case study inquiry copes with the technically distinctive situation in which there will be many more variables of interest than data points, and as one result relies on multiple sources of evidence, with data needing to converge in a triangulating fashion, and as another result benefits from the prior development of theoretical propositions to guide data collection and analysis [112]."

Typically, "scientists select a portion of the world to investigate and carry out disciplined observations in experiments. If the results of the experiments are repeatable, they count as part of the body of knowledge and progress can be made in sequences of experiments trying to falsify hypotheses. Scientific knowledge is then the accumulation of hypotheses which have not (yet) been refuted [19]."

Action research originated from the observation that quantitative approaches work very well in natural science since the investigated phenomena there do not change over time: "the inverse square law of magnetism is always, demonstrably, an inverse square law [19]." Social phenomena, such as a software development process, **do** change over time, hence the idea to participate in its change to investigate the change process itself [19].

A researcher applying the action research method uses similar instruments as those developed by the case study research community; in fact "both case-study research and action research are concerned with the researcher's gaining an in-depth understanding of particular phenomena in real-world settings [12]." The difference between action research and case study research is that action research actively interacts with the studied phenomena, trying to accomplish something.

In action research, also used in software engineering [97], the researcher acts to accomplish something and at the same time analyzes his and the actions of others to learn from it. It is "action disciplined by enquiry, a personal attempt at understanding while engaged in a process of improvement and reform [50]."

Action research involves the following steps [66]:

- planning a change,
- acting and observing the process and consequences of the change,
- reflecting on these processes and consequences,
- replanning,
- acting and observing again,
- reflecting again.

This research process is depicted in Fig. 11.3. This research process is suited to address wicked problems, as described in Sect. 1.2, that can not be formulated exhaustively and stated containing all the information needed for understanding and solving the problem. Using action research, we can tackle a problem in an iterative way. In fact, Fig. 11.3 resembles the Plan-Do-Study-Act cycle discussed in Chap. 2.

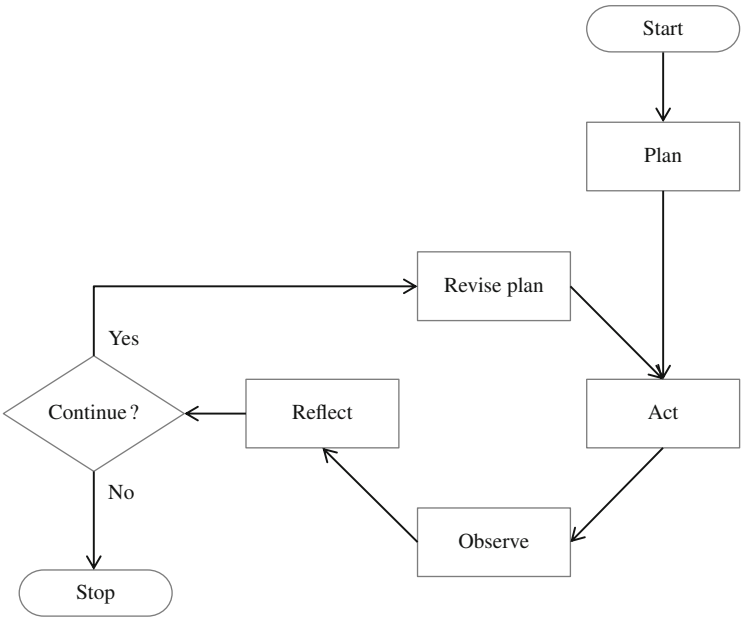


Fig. 11.3 The action research process [66]

The actual steps performed within the "observe" step of Fig. 11.3 resemble what we do in a typical case study: we want to understand why and how something happened [112]. This understanding can be gained either from outside the observed group or from inside, depending on the participation of the researcher to the activities of the group. A classification that uses four degrees of participation [39] distinguishes:

- **"complete participant"**: the observer participates to the observed activity in the same way as all other participants. The others do not know that he is observing.
- **"participant-as-observer"**: as "complete participant," except that the other participants know that he is an observer. He dedicates a small part of the available time to formally observe the group.
- **"observer-as-participant"**: similar to "participant-as-observer," just that the observer dedicates most of his time to formally observe the group and a small amount of time to participate in the activities of the group.
- **"complete observer"**: does not participate in the activities of the group, only observes.

Our approach is usually "observer-as-participant": we participate in the activities concerning the introduction of the measurement program, i.e., the definition of the measurements, in part, the interpretation of the data, and the visualization of the results.

We summarize our discussion about quantitative, qualitative, and action research in Table 11.1.

Table 11.1 Quantitative, qualitative, and action research [8, 65, 112]

Aspect	Quantitative research	Qualitative research	Action research
Suited for statistical generalization	×		
Suited for analytical generalization		×	×
Objective	×		
Subjective		×	×
Concise and narrow focus	×		
Complex and broad focus		×	×
Tests theory	×		
Develops theory		×	×
The researcher influences the studied phenomena to achieve a goal			×

11.2 Evaluating Action Research

The power of scientific method lies in the replicability of its results [19]. To distinguish replicable research from non-replicable research, scientists have identified quality criteria that estimate how credible and how trustworthy the research results are. The four most used quality criteria in case studies are [112]:

- **Construct validity**: this concept is identical to the "representation condition of measurement" presented in Chap. 9. To ensure construct validity, the researcher has to "select the specific types of changes that are to be studied, relate them to the original objectives of the study, and demonstrate that the selected measures of these changes do indeed reflect the specific types of change that have been selected [112]."
- **Internal validity**: to ensure internal validity, we have to demonstrate that the inferences made within the study are correct, i.e., that there is really a causal

relationship between two factors x and y and that there is third factor z that influences both of them.
- **External validity**: this quality criteria evaluates if the findings of the research can be generalized beyond the reported study.
- **Reliability**: we have to make sure that if another researcher performs the same study again, he achieves the same results. We have to report the methodology detailed enough that it becomes **repeatable**.

In the case of action research, "recoverability" plays an important role: one has to be able to "appraise the judgments being made by the researcher in the course of the work [19]." With a low recoverability, it is not possible to evaluate the four criteria mentioned above.

To address validity threats, researchers use "tactics," i.e., actions that are known to lower the validity treat. We used the following tactics:

- **Frequent feedback**: we report the measurement results to the stakeholders of our inquiry and ask to verify if the data represents what it is supposed to measure. Moreover, we identify outliers (i.e., extreme values) and study in detail why this outlier occurred. The analysis of extreme values helped to better understand **what** a given measurement probe measures and how we have to change it to measure what it should measure. An example of such a modification was to introduce a timeout for the measurement of development effort. We thought that if a developer leaves his computer to attend a meeting, the screensaver would tell us that we should stop the collection of effort. Unfortunately there are several users without an active screensaver, which results in too high effort for those users. To ensure that what we measure corresponds to what we want to measure, we added our own timeout to the measurement probes. Frequent feedback was used to improve construct and internal validity.
- **"Dogfooding"**: to eat one's own dog food, or dogfooding, is an expression that describes that a company uses its own products. The use of one's own software helps researchers to understand how it is to be a user of a given tool and to imagine what users would require. It therefore helps to ensure quality, build up experience in using it, and better understand how to demonstrate its capabilities. We used the measurement framework on ourselves for the same reasons: to understand how valid our conclusions are, based on the collected data. Dogfooding was used to improve construct and internal validity.
- **Modeling**: the explicit definition of the measurement goals and means using the GQM$^+$Strategies approach makes it easier for others to decide whether the measurement approach can lead to the same results in their context. Modeling was used to improve external validity, i.e., to understand in which contexts our findings could be transferred.
- **Raw data**: the data that we collect is refined and summarized to fulfill the measurement goals. We do not perform our transformations on the raw data, but we generate new, refined data. Keeping the original data (and the ability to redo all the calculations on it) helped us to document how a result was obtained, to recover from defects (if we discovered a mistake in our analysis), and—if we

discovered new aspects that made it necessary to reinterpret the historic data—to ensure the relevance of the data. This practice also improved the reliability of our approach, that is, to come to the same conclusions if the data would have been collected another time.

11.3 Introducing Measurement Programs in Companies

To successfully introduce measurement in an organization or a team, one should not fall into the trap to consider it just a technical or organizational problem.

Indeed, it is a challenge to write software that automatically collects data about the value stream, analyzes it, and visualizes it so that stakeholders can guide the activities towards the maximization of value. On the one hand, the best technology is useless if it is not accepted because of psychological or organizational reasons. As we said in Sect. 9.5, measurement can be perceived as a limitation to ones autonomy. It is important to explain the goals of the measurement and to reach a consensus about the adopted methods on how to reach these goals. On the other hand, the best accepted measurement program is also useless if it collects the wrong data.

The literature distinguishes two types of success factors to introduce a measurement program: technical success factors and organizational success factors. If we want to ensure that a measurement program is successful, i.e., if the organization gains an advantage from the measurement program, we have to consider such success factors.

The technical success factors ensure that once it is clear what we want to measure, the measurement occurs in a reliable, objective way. This depends on [25, 41, 43, 55, 67, 108]:

1. the collected measurements,
2. the training of the collaborators involved in the data collection,
3. the training of the collaborators involved in the data analysis,
4. the data collection procedures,
5. the degree to which the data collection is automated, and
6. how easy it is for stakeholders to obtain the outcome of the measurement.

While technical success factors give an answer to the question “Are we collecting the data right?” organizational success factors focus on the question “Are we collecting the right data?” Organizational success factors ensure that the collected data has value for the stakeholders within the organization. The value of the collected measurements depends on [41–43, 89, 108]:

1. the value stakeholders attribute to them, i.e., if they see a value in having a given set of measurements,
2. that enough resources are dedicated to the measurement,
3. that the management supports the measurement,

4. the level to which there are disciplined processes for software development in the organization,
5. and if the institutional beliefs, i.e., the set of beliefs that exist in the organization see value in the collection of measurements.

Particularly in small or medium enterprises, point 2 of the organizational success factors, that enough resources are dedicated to the measurement, is a success factor that is often neglected but becomes critical. The ways to address this success factor are to [27, 43]:

- implement the measurement program incrementally;
- adapt the measurement program to the measurement maturity of the company;
- keep the measurement costs low through automation; and
- keep the training costs low through the use of known ways to access the data (e.g., spreadsheets).

Based on these success factors, the action research process we adopted to introduce Lean software development is the process depicted in Fig. 11.3. The specific activities we carry out in each phase will be described below.

11.3.1 Plan

As first step we meet with stakeholders of the organization to understand their business goals and how they think that Lean should influence their achievement.

After having understood the business goals, together with stakeholder representatives, we create a first set of GQM measurement models that measure the value stream of their most important processes, i.e., we measure properties of the activities along the production process that contribute in creating what is valuable from the standpoint of the end customer. What is valuable depends on the organization, for some it is the avoidance of bugs, for others time-to-market, and so on.

In this initial planning phase, we do not develop a full-fledged GQM$^+$Strategies model, but we begin with a small number of measurement goals, suggested by the management. In other words, we do not perform the measurement process following a waterfall approach (as might be suggested by the approach described in Chap. 9, see Fig. 9.6), but we implement the measurement program incrementally.

This phase, together with the "revise plan" phase (see below), ensures that the success factors mentioned above are fulfilled, i.e., that we pick measurements that stakeholders value (see also problem 10.2).

According to our experience, it is crucial in this phase to determine all hardware and software constraints that might be relevant for the development of the non-invasive measurement probes. Some organizations have a very low hardware budget, which means that their developers might work on machines with old operating systems or low hardware capabilities. For example, a measurement probe written in Java [86] or .NET [82] that stays in the memory all the time might not be accepted by

developers if this slows down the whole machine because the operating system has to swap the memory to the hard disk. A measurement probe written as a plugin for a development environment, which slows down the application because it collects data, might also not be an option.

The developers of the measurement probes should use them themselves (a practice called "eating your own dog food") to extensively test the measurement probe in day-to-day operations and so to avoid that the measurement probe disturbs the normal execution of the development tools on the computers of the final users. If the measurement probes will limit the users in accomplishing their work, they will remove them.

The outcomes of this phase are:

- an initial GQM$^+$Strategies grid measuring aspects of the companies' value stream and
- a conceptual model of the non-invasive measurement probes that have to be developed to collect the measurements described in the initial GQM$^+$Strategies grid.

11.3.2 Act

In this phase we develop the measurement probes and install them on the machines of the organization. Some probes can collect the data from a central server and can be installed on the server, and some probes need to be installed on every single machines where users interact. This part is one of the most challenging from a technical perspective: the developers of the measurement probes need to find a way to trace the events they are interested in.

If the events are retrieved in batch mode (see Sect. 9.4), the developers have two possibilities:

- the monitored application provides a functionality to extract the events in form of a log file and the developers convert that log file into the format they need; or
- the developers have to understand the format in which the application stores its data and write themselves a tool that exports the events in a suitable format.

If the events we want to collect take place using a specific program, and if the program provides such facility, we can instrument the program. To "instrument a program" means to be able to use mechanisms provided by the program or the operating system to detect when a given event occurs. If the program does not provide any possibility to monitor its internals, we can try to use the services provided on the operating system level to monitor the application or its behavior.

The concept map depicted in Fig. 11.4 describes how a typical installation looks like.

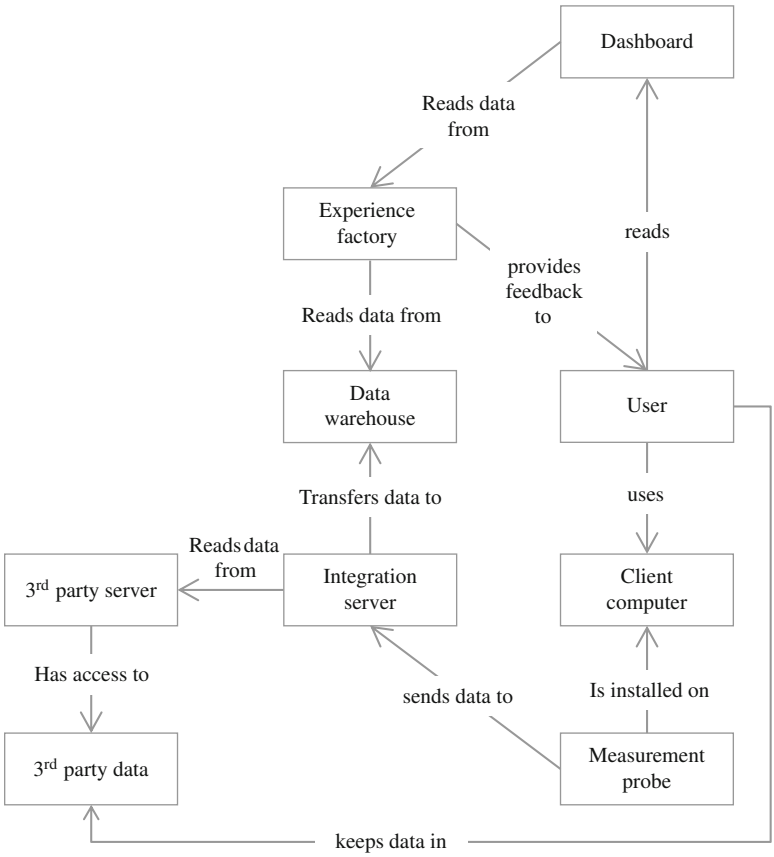


Fig. 11.4 A concept map that describes the installation of non-invasive measurement probes

We install measurement probes on the client computers that the user operates. The measurement probes send data to the measurement server. The measurement server collects all incoming data and stores it before it is transferred to the data warehouse. A data warehouse is a database that stores a "copy of transaction data [i.e., data that describe an event] specifically structured for query and analysis [68]." The measurement server not only collects the data coming from measurement probes installed on client computers but also from central data repositories in which users keep their data. Examples for such central data repositories are:

- source code repositories such as Subversion [5] or GIT [38],
- issue tracking systems such as Bugzilla [17] or Trac [107],
- workflow management systems such as Bonita BPM [14],
- document management systems such as Alfresco [3], and
- business intelligence systems such as Pentaho [90].

Such central data repositories are usually managed by a server, in which the measurement server can contact and extract data on a regular basis. Moreover, the measurement server can also integrate data coming from different data sources before transferring it to the data warehouse.

The data warehouse contains all the collected measurements of a given team. We use Apache Cassandra [4], an NoSQL database, to store all collected measurements. To support our software development team, the experience factory extracts data from the data warehouse and provides them in the form of raw data or charts to support project learning and organizational learning (see Chap. 8).

The dashboard queries the data provided by the experience factory and visualizes it showing (see Chap. 11):

- the name of the measurement,
- the value of the measurement,
- the change of the measurement compared to the previous measurement in the form of a rising or falling arrow, and
- an evaluation of the value of the measurement (see below), visualized changing the background color from green to red—that states if the measurement outcome is very good, good, fair, poor, very poor, or unknown.

A measurement can be evaluated to quickly indicate if some action is required or not. One way to evaluate a measurement is to compare it against a benchmark. A benchmark is a value that somebody finds important and against which we compare some value to understand whether this value is good or bad. The benchmark can be based on some reasoning or completely arbitrary. An example for a benchmark based on reasoning is to recommend a maximum team size of 12, since psychological studies have shown that 12 is the maximum number of people one person (e.g., the project manager) is able to guide. An example for an arbitrary benchmark is that in Europe, the maximum number of people that can be in a car is nine; otherwise, it is considered a bus, and to drive a bus, the driver needs a different driving license.

According to our experience, organizations do not have a clear understanding about their software development process before they begin measuring. Similarly, they do not have a clear understanding about the benchmarks they want to achieve. Not one of the companies we examined told us from the beginning about which benchmarks we should compare the collected metrics.

To overcome this problem, we perform the following steps:

- if possible, we compare the obtained measurements against recommended values of the literature,
- we do not benchmark the actual measurements but the changes of the measurements (if they are increasing, decreasing, or stable),
- we use anomaly detection techniques which first collect measurements (assuming that those measurements are the “normal” case) and then—using the first set of measurements to build a model—we evaluate the current measurements to detect anomalies, i.e., values that are very different from the “normal” case,

- we discuss benchmarks after we collected some data, for example, after seeing that a defect takes on average 2 weeks to be fixed, the team might decide—depending on the strategy of the company (see Chap. 8)—that we should define a benchmark lower than 2 weeks and we should monitor our processes to find out what stops us to achieve that benchmark.

Please note that in Chap. 7, we talked about the interpretation of measurements to understand whether a given measurement goal is achieved or not. The evaluation we describe here can be applied to all levels of the GQM$^+$Strategies model: we can evaluate the top goal, a subgoal, the answer to a question (i.e., the interpretation of the measurements that were collected to answer the question), or the measurements themselves.

If it is not possible to evaluate some measurement, i.e., decide whether the outcome is positive, neutral, or negative, the measurement might be irrelevant and not worth to collect.

The outcomes of this phase are:

- measurement probes that collect product, process, and resource measurements needed to answer the questions of the GQM graph defined in the previous phase;
- the necessary infrastructure to collect all measurements and transfer them to a central server; and
- rules that define how measurement outcomes are evaluated, i.e., the interpretation models that are part of the previously defined GQM$^+$Strategies grid.

11.3.3 Observe

In this phase we monitor the collected data as it is collected on the measurement server and intervene if a measurement probe collects wrong or no data.

To guarantee the correct working of our own installation of the system, we developed a watchdog application that regularly verifies if the components on the measurement server are correctly working (e.g., check if the databases are all online, the server answers to requests, etc.) and if some data was received within some defined number of days. If one of this conditions is not met, we notify this to the development team via e-mail. Additionally, this watchdog application has a REST interface [35], which can be used to read the status of the last evaluation.

To visibly notify everybody that something is wrong, in parallel to the e-mail alert, we use a device called "Karotz [83]." This device has the form of a rabbit and contains a Linux [105] computer that connects to the Internet and can be programmed using JavaScript [32].

After the start of Karotz, a green led is blinking on the rabbit to indicate that it is now ready to receive messages; see Fig. 11.5a.

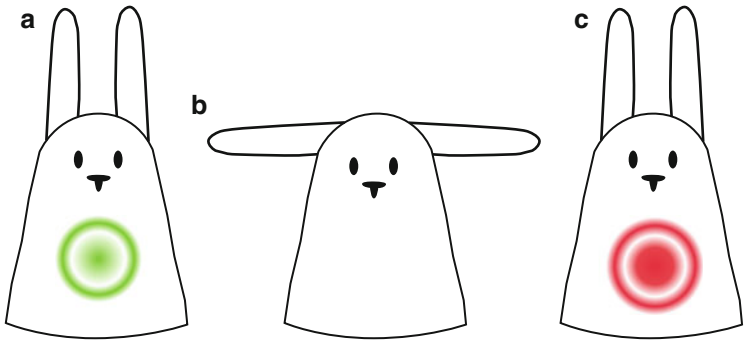


Fig. 11.5 Different ways on how the Karotz device can notify the user. (**a**) Normal state. (**b**) Moving ears. (**c**) Alert

The device exposes commands to:

- activate the lamp in different colors,
- deactivate the lamp,
- play a sound file,
- use text-to-speech to say something,
- move with the ears (used to draw attention before saying something); see Fig. 11.5b,
- react to a button installed between the ears,
- react to somebody moving the ears,
- react to a command identified through speech recognition, and
- react to an RFID tag put under the nose of the rabbit.

The commands exposed by the device can be triggered using two methods: either using a push approach, sending the commands to the device using REST, or using a pull approach in which a custom JavaScript program runs in background and calls the commands from inside the device.

We use the second approach and install a JavaScript application on the device that regularly checks the status of the watchdog application through the REST interface. If the watchdog program is not reachable, the Karotz device says "the watchdog application is not responding" and the lamp blinks in orange until somebody presses the button between the ears of the device. If the watchdog application responds, but the status returned some component that does not work correctly, the Karotz device blinks in red (see Fig. 11.5b) until somebody presses the button between the ears and says "there is a problem with component X." X, the name of the component not responding correctly, is returned by the watchdog program in the status response.

The pseudocode for this application is shown in Listing 11.1:

```
1 // read the status of the watchdog server
2 var response = '';
3 try {
```

```
  response = callUrlThatReturnsTheStatusOfTheServer();
}
catch (error) {
  useTextToSpeechToSayThatTheWatchdogIsDown();
  setTheLampToOrange();
  blinkUntilTheButtonIsPressed();
}

if (!response.equals('ok')) {
  useTextToSpeechToSayThatThereIsAProblem(response);
  setTheLampToRed();
  blinkUntilTheButtonIsPressed();
}
```

Listing 11.1 Pseudocode to read the status of the watchdog server

The advantage of using the Karotz device compared to other systems (e.g., continuous integration servers such as Jenkins [59]) is that—as we wrote for dashboards in Chap. 10—it uses a push approach to notify the team about some problem. The first thing in the morning that everybody sees is if the Karotz device shows that there is a problem. The same approach has been used by others to notify the team about build errors [29].

In fact, the Karotz device is one way to support autonomation: it obtains the current status of a system and notifies the team if something is wrong.

The outcomes of this phase are:

- the observation of which data are collected and how they are interpreted,
- ideas to improve the collected data or its interpretation, and
- a mechanism to understand if the measurement framework is working correctly.

11.3.4 Reflect

As we collect the measurements, we provide feedback to the users about what data has been collected. This feedback occurs daily through what we call the "daily mail" and in regular feedback meetings in which we ask users what they think about the collected data and the contribution of the collected data to achieve the stated measurement goals. This step corresponds to a validation of the defined measurements and questions.

Moreover, in this phase we analyze the collected data together with the stakeholders of the organization and discuss how to use it to progress towards a Lean organization. This step corresponds to a validation of the defined measurement goals.

The outcomes of this phase are:

- ideas to improve the collected data or its interpretation, and
- the understanding of how the data can be used to infer how Lean a team is and where it should improve first.

11.3.5 Revise Plan

The results of the reflecting phase are used to adapt the measurement probes and construct the complete GQM$^+$Strategies model, together with the definition of business goals following the template described in Table 7.1.

11.4 Case 1: Exploration or Exploitation?

In the first case,[1] we studied how a software development team is investing its time within small software company. The company management felt that there is a trade-off between two aspects: exploration or exploitation:

- **Exploitation**: with exploitation the company management understood the improvement of existing features of their software. This includes the fixing of defects as well as the enhancement of existing features. Exploitation, in this case, helps to maintain the existing customer base, but, as the competition is constantly improving their products, on the long run is not enough to ensure that the company remains profitable.
- **Exploration**: with this term the company management understood the innovation of their products and the implementation of new features to address new requirements by existing customers but particularly potential customers that could not be acquired so far because of missing features.

The revenue stream of the software company consists of two parts: the first part comes from the acquisition of new customers and the second part from a yearly maintenance contract stipulated with the majority of the existing customers. The maintenance contract is valuable for the customer and the software company:

- for the customer, the contract protects from unexpected expenses, e.g., if the state changes, the laws regulating what and when companies have to report to authorities and
- for the software company, the contract guarantees a certain revenue stream that can be considered in the budget.

In the studied software company, the revenue generated by the maintenance contracts is enough to cover a large part of the fixed costs. Therefore, it is important to satisfy the requirements of existing customers to keep them. The software company faces the problem that customers paying the maintenance fee expect every modification to be included in that fee. If, for example, a trading company using the software to handle its orders constructs an additional warehouse to handle shipments, the company expects that all the configuration of the software it bought

[1] We use the structure proposed by [78] to report about the three action research cases.

years ago is included. If the software company did not accept, it would be likely that the trading company changes the software on the next occasion (if, e.g., a competitor makes a tempting offer).

To alleviate this problem, the company does not upgrade customers with a maintenance contract automatically to newer versions of the software but provides only maintenance and support for the current version. If the customer wants to upgrade (e.g., because he wants to use advanced features available only in a newer version), this is not covered by the maintenance contract and he has to pay a separate fee for the acquisition of the new version, the conversion of the old data to the new format, and the training of the employees with the modifications in their working process because of the features present in the new version.

For the software company, this means that it is important to balance the development of its development team: a lot of improvement exploitation makes customers happy on the short run, but on the long run, the competitors will create a better product than the one of the company and the customers will switch. On the other hand, a lot of innovation will help to acquire new customers faster but make the existing customers unhappy. They will feel that their product is not maintained and start to watch out to find something better.

The management is therefore interested to understand how the development team is spending its time: some improvement is needed to justify the charged maintenance fee and to keep the software installed on the machines of the clients up to date, but too much improvement meant that the team had less time for innovation, i.e., to develop features that attract new customers or that convince existing customers to buy a version upgrade. Figure 11.6 depicts this trade-off.

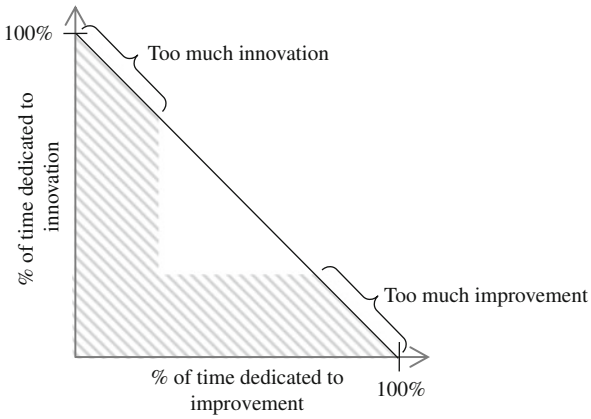


Fig. 11.6 Trade-off between innovation and improvement

On the abscissa we plot the % of time dedicated to improvement, and on the ordinate, we plot the % of the time dedicated to innovation. If we assume that the sum of both components (improvement and innovation) has to be 100 %, the line on which all combinations of improvement and innovation lie is the one shown in Fig. 11.6. If the team dedicates time to other activities, also points below the line are

possible. The management assumes that the shaded areas are problematic because—as said above—they represent a situation in which the company loses competitivity on the long run.

This problem can also be modeled similar to the one discussed in Chap. 5 about the determination of the sweet spot that minimizes risk exposure (see Fig. 11.7).

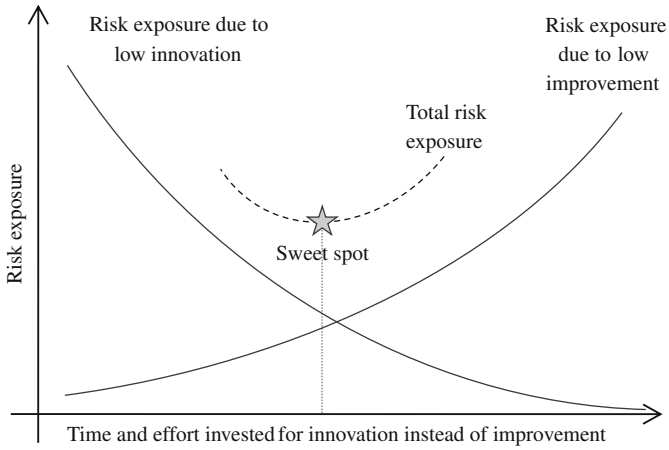


Fig. 11.7 The sweet spot between the risk of too much improvement and too much innovation

11.4.1 Theoretical Framework

A theoretical framework is the set of theories and of explanations (and the relationships among them) on which we base our study [61].

In this case, we assume that:

- Development costs are mainly determined by labor costs since writing software is a labor-intensive activity and hardware costs are low compared to personnel costs [56, 102].
- To ensure the financial liquidity (on the short run) and the profitability (on the long run) of a software company, it is important to balance between activities that generate revenue and maintenance activities that keep existing customers happy, i.e., to balance exploration and exploitation [75], as also suggested by the Balanced Scorecard [64].
- Developers spend the most time of their day in front of the computer, which means that using non-invasive measurement probes, we are able to track a significant part of the effort. Not all time is used to code, but also to communicate with others, gather information, etc.

The concept map for this case is the one depicted in Fig. 10.2, which we repeat in Fig. 11.8.

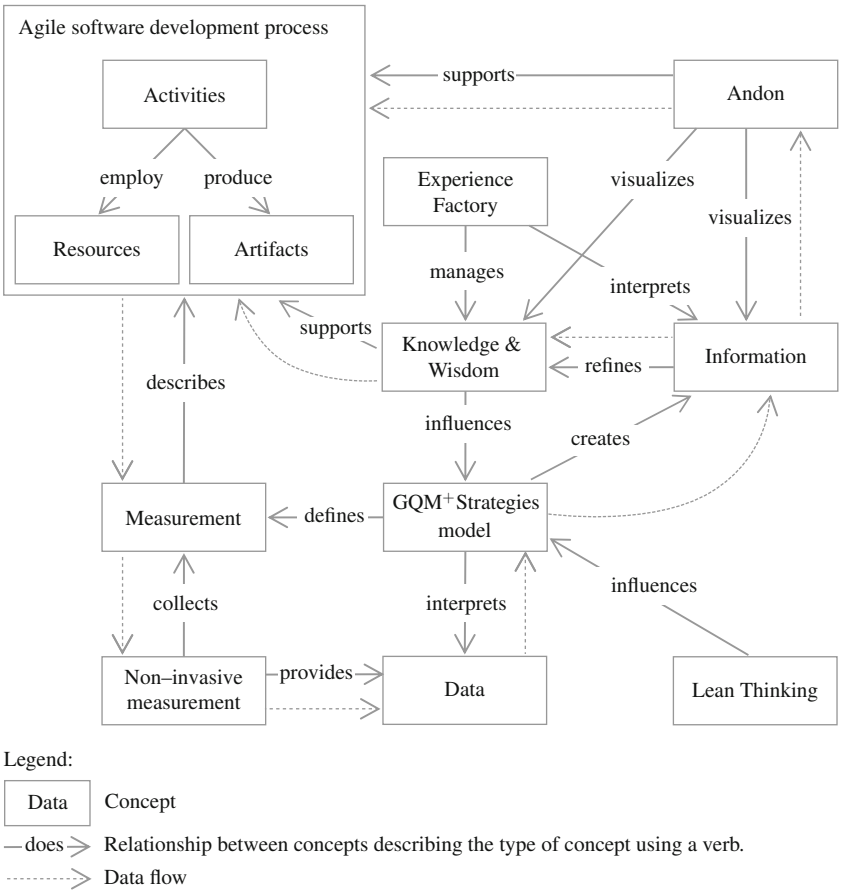


Fig. 11.8 Concept map for the first case study

The roles that the different concepts in this case have are:

- The Agile software development process is the object of study, in particular the activities.
- Measurement: we measure how much time a user spends modifying an artifact as well as descriptive properties of the currently modified artifact to classify the effort as exploration or exploitation, e.g., the name, containing folder, size, etc.
- Non-invasive measurement: we developed probes to identify process, product, and resource measurements, i.e., who was doing what, for how long, on which artifact.

- Data: the collected data are a log of the activities of all team members.
- GQM$^+$Strategies model: defines how an activity is classified as exploration or exploitation.
- Information: a list of activities with their classification as exploration or exploitation.
- Knowledge and Wisdom: understanding how much we invest in innovation or exploitation helps us to reason about the know-how: about how innovation and exploitation takes place, which other possibilities exist, and how we can better coordinate those two activities. Furthermore, this knowledge helps us to identify what stops us from innovating or exploiting previous innovations.
- Andon: visualizes the identified activities and their classification. Two visualizations are used: one to visualize the overall distribution between exploration and exploitation over time and one to visualize this distribution for logical code entities, i.e., packages (in Java), namespaces (in C#), classes, and methods.
- Experience Factory: we store the definitions on how we distinguish the time spent for exploitation and time spent for exploration. We also store the history of exploration and exploitation effort, together with the component that required the effort. This can help the team to estimate the maturity of a given component.

The GQM$^+$Strategies model consists of GQM$^+$Strategies elements and GQM graphs. Our starting point is the following GQM$^+$Strategies element:

- **Organizational goal**: balance the amount of resources invested in exploration and exploitation. Using the GQM$^+$Strategies goal template, this goal can be refined as follows:
  - **Object**: all software development products
  - **Focus**: invested time for innovation and exploitation
  - **Magnitude**: ensure that the invested time spent on innovation and exploitation is balanced. Initially this means that the employed resources should be equal for exploration and exploitation. Over time, this might change to adapt the software development activities to the market needs.
  - **Time frame**: continuously
  - **Organizational scope**: software development team
- **Context factors**: the optimal balance depends on different factors, e.g.:
  - Imminent deadlines are approaching (e.g., an upcoming release): exploration is (usually) not desired because the product has to be stabilized before delivery (see, e.g., [24]).
  - Critical defects present in the backlog: exploration is (usually) not desired because critical defects can induce/force clients to switch to competing solutions and can damage the reputation.

  - Promising new technologies become available: more exploration than exploitation is (usually) desired, since that could provide the organization a first-mover advantage [74] for a specific application.

- **Assumptions**:
  - Continuous innovation of the product helps to maintain the competitive advantage of the organization [106].
  - Continuous improvement of the existing features helps to maintain the usefulness of the product for the user [71].
  - Constant refactoring increases maintainability [36].
  - Development costs are mainly determined by labor costs.
  - Software development labor costs are mainly determined by interactions with the computer.
- **Constraints**:
  - To reduce measurement costs, we favor non-invasive measurement over manual measurement.
- **Strategy**: monitor the amount of time developers spend editing code that can be classified as innovation or exploitation and visualize the outcome so that the team can decide how to proceed.

The associated GQM graph consists of the following elements:

- **Measurement goal**: analyze the software development process for the purpose to balancing with respect to the time developers spend writing code from the point of view of the software developer in the context of a software development project (follows the GQM goal template of Fig. 7.5)
- **Questions**:
  1. Which artifacts are developers working on?
  2. To which project do these artifacts belong?
  3. How much time do they work on the different artifacts?
  4. To which class of activities does the work on artifacts belong: documentation, coding, testing, browsing, communication, or unknown?
- **Metrics**:
  1. For question 1 we extract properties useful to identify the artifact:
     - the application name of the focused artifact,
     - the file name of the focused artifact,
     - the package name of the focused artifact,
     - the class name of the focused artifact,
     - the class annotations of the focused artifact,
     - the implemented interfaces by class of the focused artifact,
     - the method signature of the focused artifact, and
     - the method annotations of the focused artifact.

2. For question 2 we extract properties to identify the project, since we want to balance the effort per project:
   - the project name of the focused artifact (if the focused artifact belongs to a software development project) and
   - the full path to the file name of the focused artifact (we use the full path to extract the project, assuming that all files belonging to one project are stored in the same folder).
3. For question 3:
   - the time spent working on the focused artifact.
4. For question 4 we classify the data collected because of questions 1 and 2 according to the rules defined in Table 11.2.

- **Interpretation model**: we consider the goal satisfied if we are able to track and classify at least 90 % of the daily working time of every developer.

Table 11.2 Rules to classify activities

Rule	Classify effort as					
	Documentation	Coding	Testing	Browsing	Communication	Unknown
File name ends with ".docx" (e.g., Manual.docx)	×					
File name ends with ".java" (e.g., Database.java)		×				
Method is annotated with "@Test"			×			
Window title contains "http" (e.g., https://www.site.com)				×		
Application is "Skype" or "Microsoft Outlook"					×	
All other cases						×

11.4.2 The Study

In this organization, the management understood what is valuable for the customer, why the customer is willing to pay money for a certain product or service, and how much it costs to the company (in terms of market share and in terms of salaries) to provide the product or service.

What was not clear was the cost distribution between the different activities performed by the team. We started implementing measurement probes to monitor development process activities. We developed three probes:

1. an application written in Microsoft C# [79] that tracks the current application and the current document (if the application shows the current document in the caption) through the operating system API,
2. a plugin for Eclipse [30] that tracks the current method (together with the package, the class name, the file name, and the project), and
3. a plugin for Microsoft Office [80] that tracks the current application (editing documents, spreadsheets, presentations, or databases) and the current document name that is being edited.

The measurement probes were implemented as depicted in Fig. 9.13. The measurement probe 1 obtains the currently focused window and its title text from the operating system (see Fig. 11.9) and reports any window switch or title change to the server. The title can—depending on the application—contain the full path to the document.

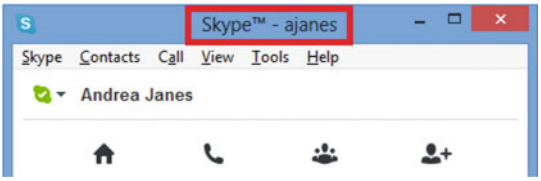


Fig. 11.9 The caption of a window

The other two probes (probes 2 and 3) are implemented as plugins for the hosting application: the plugin for Eclipse is written in Java [86] and the plugin for Microsoft Office in C#. Both plugins regularly poll the current method, namespace (in C#), package (in Java), file, project, or document from the hosting application and report changes to the measurement framework (see Fig. 11.10).

```
public class Main {

    public static void method1() {
        System.out.println("Hello");
    }

    public static voi method2(int count){

    }

}
```

Fig. 11.10 The cursor within a method in Eclipse

Table 11.3 shows sample data as it is collected by the three described probes.

Table 11.3 Example data of case 1

Machine	Application	Item	Duration	Probe
1	Eclipse	at.company.project8.Action.run()	10:00	2
2	Eclipse	at.company.project8.ActionTest.test1()	12:00	2
2	Eclipse	at.company.project8.ActionTest.test2()	7:30	2
1	Google Chrome	http://www.codeproject.com	2:00	1
3	Google Chrome	http://stackoverflow.org	7:00	1
4	Microsoft Excel	c:\projects\Comparison.xlsx	9:00	3
4	Notepad	c:\projects\project8\summary.txt	3:00	1
2	Microsoft Word	c:\projects\project8\manual.docx	15:00	3

The so collected data represents the effort developers spend interacting with the computer. We did not collect all the effort a developer spends during the day, e.g., how long somebody was attending meetings, was reading a magazine, was thinking, or was talking. It would be possible to develop mechanisms to non-invasively track the position of people within the office using RFID tags or to detect if somebody is actually looking at the screen using an eye tracker. For the purpose of this study, we found that the higher accuracy would not have justified the additional costs to implement such a solution.

Since the team was using JUnit, it was possible to isolate the amount of time dedicated to testing in comparison to coding. Test cases in JUnit are annotated with "@Test." By logging the time spent in methods with that annotation, we were able to extract the testing effort.

The collected data was useful to get a picture of how the team was spending its time: through rules (see Table 11.2) it was possible to classify activities as "documentation," "coding," testing," "browsing," "communication," or "unknown" [7].

The information we gained in this way (see Table 11.41) helped to understand the distribution of activities [6] during, for example, a month and to visualize it using stacked bar charts such as in Fig. 11.11 or other visualizations that help to focus on the important activities [37].

From the visualization in Fig. 11.11, we see that the different projects required different amounts of documentation, coding, etc. Such information can be used by the team to discuss why for certain projects such differences exists, if this is a problem, and, in case it is a problem, how to address it.

At this point we were able to distinguish between high-level activities but not yet understand whether these activities were performed with the intention to innovate or to improve.

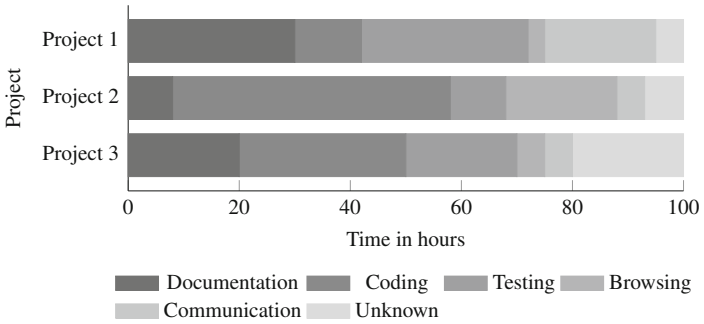


Fig. 11.11 Distribution of activities for three projects

While brainstorming with the developers of the team how to accomplish this, one developer noted that for certain types of applications, when they developed one or several new features, they were creating new files to host the new functionality.

For example, let us assume that a user wants to manage, together with the contact information of a customer, all the phone calls he receives from that customer. This requires to add fields to the database, create new dialog boxes or extend existing ones, and write code that stores to and reads the data from the database. If the modification is small, developers will just modify or extend existing files; if the modification is larger, developers will refactor existing files and add new files.

The developers hypothesized that when they are innovating a product, they more often create files than modifying existing ones, and when they are improving a product, they usually modify existing files instead of creating new ones. See Table 11.4 for some examples.

Table 11.4 Reasons to create new files for different application types

Application type	New files are created for
Data-driven applications	Several new Java files to implement the user interface and the interaction with the database
Web applications using JSP [87]	Several new JSP files to implement the web pages and Java files to implement servlets
REST services using Jersey [60]	One new Java file for each group of services

Some arguments **for** this hypothesis were:

1. Developers love to plan and implement their own ideas, i.e., write code. On the other hand, they hate to read code written by others [70]. Apparently developers perceive it as painful to try to understand the rationale behind code written by others. This might be one reason why we observe that developers tend to create new code structures before extending existing ones.

2. If it is important for the team to ensure maintainability, source code files cannot grow infinitely. They have to be structured into small, simple (intended as the antonym of complex), understandable, modular, reusable parts. This makes it more likely that a developer creates new files when adding functionality. This is confirmed by the descriptions in the literature of anti-patterns like "lava flow" [16].

Some arguments **against** this hypothesis were:

1. **False positives**: because of the argument 1 mentioned above in favor for the hypothesis, it can also be that new files are added to exploit not to explore.
2. **False negatives**: it might be true that the more code is added, the more likely it is that a developer adds new files, but the degree of innovation is not related to the amount of code generated. There can be very innovative good ideas that are implemented refactoring existing files.

To examine this hypothesis, we extracted the commit log from a JSP web application from its Subversion [5] repository. As we see in Listing 11.2, the output states for every commit the user, the date, and the list of files that were added (marked with an "A" at the beginning of the line), modified (marked with an "M" at the beginning of the line), or deleted (marked with a "D" at the beginning of the line).

```
svn log --username (user) --password (password) (repository)

------------------------------------------------------------------------
r32 | mike | 2013-01-13 00:43:13 -0600 (Sun, 13 Jan 2013)
Changed paths:
   M /foo.c

Added defines.
------------------------------------------------------------------------
r31 | joe | 2013-01-10 12:25:08 -0600 (Thu, 10 Jan 2013)
Changed paths:
   A /bar.c

Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2013-01-07 21:48:33 -0600 (Mon, 07 Jan 2013)
(and so on)
```

Listing 11.2 Extraction of the commit log from a Subversion repository using the command line tool "svn"

The developers found that the result depicted in Fig. 11.12 supported the hypothesis. If we define the exploration phase as one in which many new files are created, we can see that the first version of the software contained the files 0–270, and then for some time (1 month) the existing files were improved and the existing functionality was stabilized. The next (slower) innovation phases can be seen for

the files from 270 to 320 and from 320 to 380. Currently, another innovation phase (starting from file 390) is ongoing.

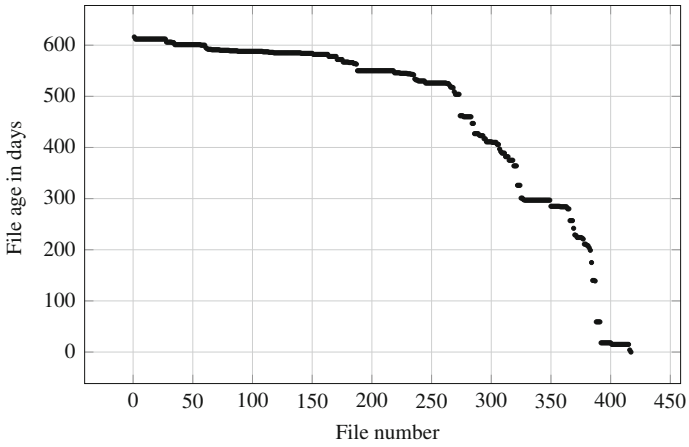


Fig. 11.12 File age in days for each file of a Subversion repository containing a JSP web application

Another example of such a study is depicted in Fig. 11.13. Here we see that the project was not developed from scratch, but copied into the Subversion repository 812 days ago. A similar thing happened for the files 6,100–9,950 and 10,000–11,000: a module developed in a separate repository was merged with the main production code.

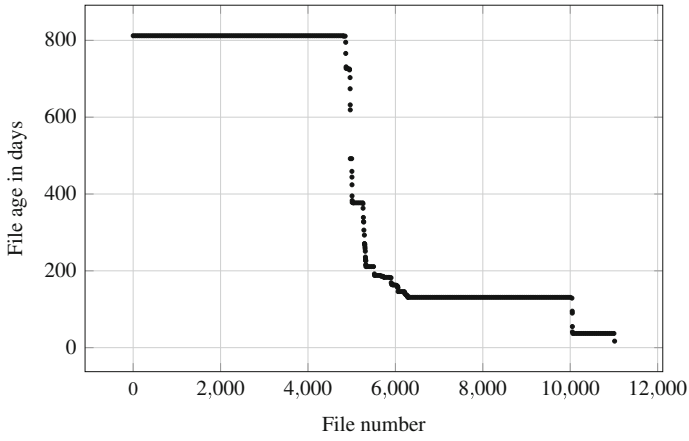


Fig. 11.13 File age in days for each file of a Subversion repository containing a .NET application

We decided to use this finding to differentiate between exploration and exploitation as follows. We measure the time a user spends editing source code files using the non-invasive measurement probes described above. The obtained data (see Table 11.3) describes the editing effort on the method level (and aggregating all the effort for one file, on the file level).

If we want to get an estimate of the amount of exploration (innovation) contained in the effort, we are interested in how much time was spend on new files. The newer the file, the more likely it is that the modification was an implementation of something new. Following the same logic, if the file is old, we do not want to count the editing time as much as for new files.

To achieve this, we multiply all editing times with $\gamma_{\text{exploration}}^{\beta}$, where $0 < \gamma_{\text{exploration}} < 1$ is the attenuation factor for exploration, a number that expresses how much we want to reduce the time if the file is old and β a number that expresses the age of the file. Therefore, the degree of **exploration** of a developer d that edited n files is given by:

$$d_{\text{exploration}} = \sum_{f=0}^{n} \text{effort}_d(f) \times \gamma_{\text{exploration}}^{\text{age(f)}},$$

where the function $\text{effort}_d(f)$ returns the effort a user spent editing the file f and age(f) returns the age of the file f. In our case we calculated the effort of a file in minutes, the age of a file in months, and chose $\gamma = 0.5$.

Accordingly, the degree of **exploitation** is calculated as follows:

$$d_{\text{exploitation}} = \sum_{f=0}^{n} \text{effort}_d(f) \times \gamma_{\text{exploitation}}^{\text{age(f)}},$$

where $0 < \gamma_{\text{exploration}} < 1$ is the attenuation factor for exploitation. By choosing

$$\gamma_{\text{exploitation}} = (1 - \gamma_{\text{exploration}}),$$

the sum $d_{\text{exploration}} + d_{\text{exploitation}}$ corresponds to the actual effort the user spent editing source code files and the values of $d_{\text{exploration}}$ and $d_{\text{exploitation}}$ correspond to a distribution of the effort collected by our measurement probes, weighted by the age of the files that were edited.

This approach follows a similar logic as the contemporary h-index [100]. This index is used to value the research output of a scientist in academia extending the idea behind the h-index [48]. The h-index estimates value of the output of a researcher considering the amount of citation his work has received. This index is calculated considering all the publications an author produced and the citations his work received. In brief, the higher the h-index is, the more the works of a researcher are cited. The authors of the h-index expect that a high number of citations means that the work has a higher relevance than a work that is not cited.

Similar to our idea to estimate exploration, the **contemporary** h-index weights the contribution of each work of a researcher according to its age, counting it less

for older articles. This index aims to measure how innovative the research of an author is.

Another similar approach is used by researchers to analyze the expertise level based on commits to source code repositories. To estimate how much somebody knows about a part of code, they study the commits of a given author, decreasing their weight the more in the past they occurred [46, 96].

11.4.3 Results

We visualize the results in a dashboard, distinguishing organizational learning and project learning (see Chap. 8). To support organizational learning, we show the development of exploitation and exploration, for example, in a stacked bar chart as in Fig. 11.14.

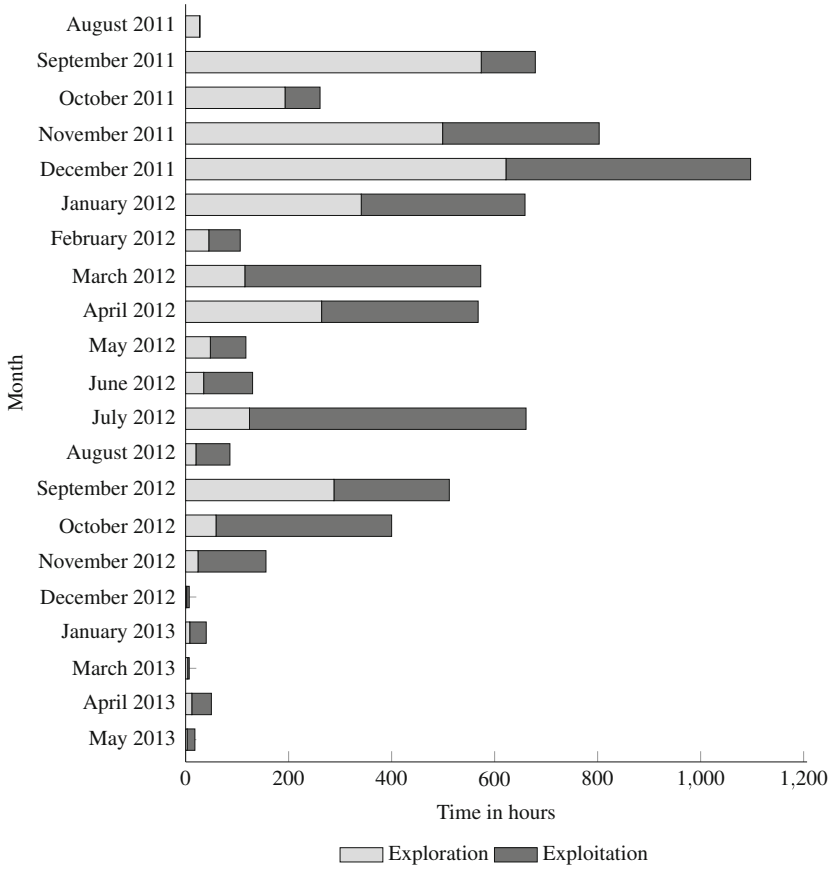


Fig. 11.14 Distribution of activities for three projects

To support project learning, a visualization that shows the current distribution of exploration and exploitation is useful, for example, in a hierarchical pie chart as in Fig. 11.15.

Fig. 11.15 Hierarchical pie chart relating the source code size with the degree of innovation (This visualization was created using the d3.js library [15])

This pie chart shows the structure of the source code together with two measurements. The most inner ring is the root namespace (in C#) or the default package (in Java). The next rings are the actual namespaces or packages. To explain the diagram, we added some labels to the diagram depicted here. In the real implementation, a label appears when the user hovers a specific segment with the mouse. The inner segments represent namespaces or packages and the outer segments classes. The size of the segment represents the logical lines of code, the color of the segment, and the degree of exploration as calculated by our approach. In this visualization we assign the colors green, orange, and red to different levels of exploration, depending on what the team thinks is adequate. These thresholds are likely to be different for a computer game compared to the software for a healthcare system.

11.4.4 Discussion

The here described action research was carried out together with the organization interested in a non-invasive approach to find out whether they spend too much time on exploration or exploitation.

The experience described here is not replicable to other environments without considering whether the way how programmers explore and exploit occurs as described here, and it therefore makes sense to measure exploration and exploitation using the heuristic described here.

Also, developers can sabotage our proposed way to measure, by creating files when not innovating and vice versa. In our case the management considered this acceptable and even positive since this demonstrated to the developers that there was no intention to spy on them or to judge their personal abilities but to visualize what was happening during the development process and to increase transparency.

A similar problem to the one described in this case study was addressed by Nord and Ozkaya [85]: the authors aim to "provide a framework for balancing the allocation of critical architectural tasks to development effort." They study the trade-off between having many small architectural increments with a high cost of rework but causing only short delays in the production process, or having few large architectural increments (what they call "rearchitecting") but causing long delays because of "significant rework beyond the expected limits of refactoring." They propose to highlight architecture-related tasks or to enforce work in progress limits for "tasks architecture-focused acceptance test cases, architecture prototyping, or rework to pay back architecture debt." The idea is to balance the implementation of features with the development of the architecture. Such an approach could have been also used to support the team described in this case study.

From a Lean perspective, the here described case study contributed to the maximization of value, the creation of knowledge, and improvement as summarized in Table 11.5.

11.5 Case 2: Non-invasive Cost Accounting

Cost accounting is the theory and system of setting up, maintaining, and auditing the documentation about the cost of items involved in the production [21].

Cost accounting can serve different purposes, for example [99]:

- to understand if past investments paid off or not, i.e., if the returns of an investment were enough to cover the initial expense,
- to calculate a minimum price that covers the production costs, or
- to gather data about the possible consequences of decisions.

In the previous case, we observed the interactions a developer had with the computer to, based on rules, infer the activities he was performing. In this case, we used a similar approach to develop a non-invasive cost accounting approach for software development. The goals were to first **identify** costs and then to **attribute** them to their cause based on data collected non-invasively.

Table 11.5 Lean aspects addressed by the first described case

Goal	Strategy	Rationale
Maximization of value	Identification and elimination of waste	We trace activities as they occur, using the developed measurement probes, and classify the data as exploration or exploitation. This helps to identify "waste of processing" (see Chap. 2), i.e., unnecessary activities. In our case, we cannot say that, e.g., a lot of exploration is not necessary in general; it is not necessary at some specific point in time and might become again necessary later. For example, around important deadlines, release dates, etc., exploration might be undesirable, while during normal operations, it might be desirable to identify new opportunities
	Pull not push	As advised in Chap. 8, the collected data was only evaluated to determine exploration and exploitation because this was a need identified by the management. Only if more information needs arise, more data should be collected, or new ways to analyze it should be implemented [57]
	Leveled production	Balancing exploration and exploitation supports the idea to level the production
Creation of knowledge	Andon	The dashboard creates transparency, increases the level of information (know-what), and motivates the development team to discuss and decide about the future strategy. The decided way to automatically extract knowledge is documented in the form of a GQM$^+$Strategies model. Thresholds that describe how much exploration and how much exploitation is desirable should be stored and documented in the Experience Factory to inform everybody about the goals of the measurement (see Chap. 9)
	Worker involvement	The worker involvement occurs in the regular discussion on how to improve the identification of exploration or exploitation and whether currently more of one or the other is needed
Improvement	Autonomation	Non-invasive measurement together with the Andon concept implement autonomation. Implementing autonomation creates the conditions so that it is easier to monitor and ensure quality

(A Not So Brief) Introduction to Cost Accounting

Cost accounting distinguishes between variable and fixed costs [99]:

- **Fixed costs** arise to provide what is needed to produce the desired products or services, such as machines, factories, personnel, etc. To a certain degree, they are independent to the produced output. For example, let us assume we rent a doughnut factory. The rent we have to pay during 1 month does not change whether we produce one doughnut or if we let the machines work at their full capacity day and night. On the other hand, fixed costs do change if we need to change the maximum output of our doughnut machines, for example, if we need to buy more machines to be able to produce more doughnuts per hour. Costs that we incur to increase our capabilities are also called "stand-by costs."
- **Variable costs** are costs that do change in relation to the output.

Variable and fixed costs (unless we change our production capacity) change in relation to the output quantity as depicted in Fig. 11.16.

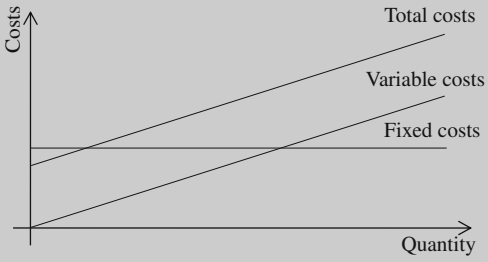


Fig. 11.16 Fixed, variable, and total costs [99]

Cost accounting also distinguishes between direct and indirect costs [99]:

- **Direct costs** are costs that we incur because we produce a specific product or we deliver a specific service. Since we know which product or service caused them, we can **directly** attribute the costs to what caused them. Examples of direct costs are the parts we need to produce to assemble a car or the amount of work we need to develop a given software development project.
- **Indirect costs** are "indirect" because [99]

(a) we cannot identify a specific product or service that caused the cost. This might be because all products or services benefit from the incurred cost. Examples for such costs are heating and lighting.

(continued)

(b) we do not want to identify the specific product or service that caused the cost because this would be too time consuming and therefore expensive.

Fixed, variable, direct, and indirect costs appear in all combinations. Table 11.6 gives some examples.

Depending on **how** we attribute costs to products and services, we will obtain different results and make different types of conclusions. In the following, we briefly look at three prominent approaches: Total Absorption Costing, Variable Costing, and Activity-Based Costing.

Table 11.6 Examples of fixed, variable, direct, and indirect costs

	Fixed	Variable
Direct	Stand-by costs need to produce one specific product	Costs for raw materials
Indirect	Heating costs	Accounting costs[a]

[a]Accounting costs (in part) change in relation to the amount and complexity of products and services an organization provides, but it is usually not worthy to attribute them directly to the products and services that caused them

Total Absorption Costing distributes all costs (direct and indirect) to the produced products or provided services. Let us assume, we are producing bikes in a workshop and selling them over an online store. We produce two bike models: model A and model B. Table 11.7 lists the costs to produce the two models.

Table 11.7 Costs for producing the bike models A and B

Costs	Model A	Model B
Material	€200	€500
Labor	€60	€120
Rent	€24,000 per year	
Heating	€1,000 per year	
Marketing	€10,000 per year	
Administration	€50,000 per year	

The price that Total Absorption Costing calculates is the minimum **long-term** price to cover all costs. It attributes the direct costs to the product that caused it and distributes the indirect costs over all produced items as depicted in Fig. 11.17.

(continued)

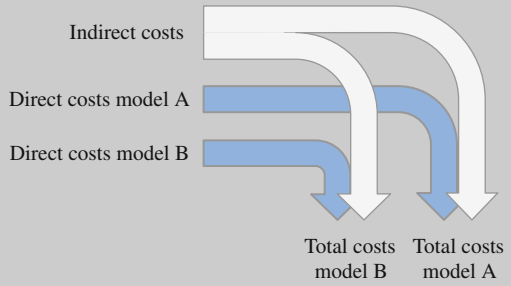


Fig. 11.17 Cost calculation in Total Absorption Costing

Table 11.8 shows the calculation of the production price of each model using Total Absorption Costing, assuming that we sell 200 bikes of model A and 50 bikes of model B per year. In this case we distribute the indirect costs equally over all bikes. We could choose a different distribution key such as machine hours, personnel hours, material used, etc.

Table 11.8 Total Absorption Costing example

	Cost	Model A	Model B
	Material	€200	€500
+	Labor	€60	€120
=	Total direct costs per bike	€260	€620
	Rent (€24,000 ÷ 250 = €96)	€96	€96
+	Heating (€1,000 ÷ 250 = €4)	€4	€4
+	Marketing (€10,000 ÷ 250 = €40)	€40	€40
+	Administration (€50,000 ÷ 250 = €200)	€200	€200
=	Total indirect costs per bike	€340	€340
	Total direct costs per bike	€260	€620
+	Total indirect costs per bike	€340	€340
=	Total costs (direct and indirect) per bike	€600	€960

Let us also assume that a customer makes us the following offer: he would buy 100 bikes of model B for €800 each. According to Total Absorption Costing, we should not accept the offer, since the costs of a model B bike is €960.

On the other hand, if we sold already 250 bikes, we covered already all fixed costs and it is not necessary to charge €340 indirect costs per bike. For the 100 additional bikes, €800 would be enough to cover the variable costs and to contribute with €180 per bike to the profit.

(continued)

This is what the proposers of **Variable Costing** criticize: that Total Absorption Costing can lead to wrong decisions. The indirect costs of €340 per bike are based on the expected number of sold bikes. If the demand for bikes increases (e.g., because of an increase in gasoline prices or due to a special advertising campaign), the price is too high and could lead to missed opportunities as the example above. On the other hand, if the demand for bikes decreases, Total Absorption Costing recommends to increase the prices: with a lower output, all indirect costs have to be redistributed over fewer items. Increasing prices when the demand decreases is not the best way to stimulate a stagnating market.

In summary, according to Variable Costing, the **short-term** minimum price for a product is the price that covers only the variable costs. Of course, on the long term, we have to cover also the fixed costs. Everything we earn above the variable costs contributes to cover the fixed costs and to the profit (see Fig. 11.18).

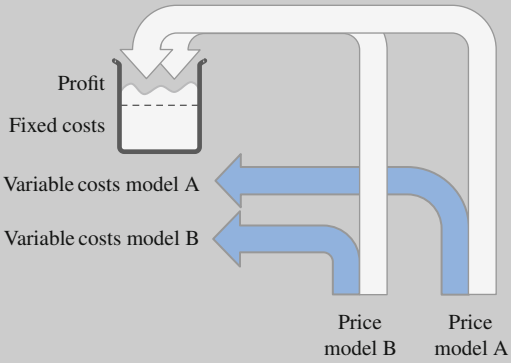


Fig. 11.18 Cost coverage in Variable Costing

To evaluate the business performance, the organization can calculate the contribution margin as in the example in Table 11.9. The contribution margin is the total revenue minus the direct costs. It contributes to cover all indirect costs and—if it exceeds them—contributes to the profit of the organization. (Strictly speaking, revenue minus costs is not (yet) the profit, since income taxes and interests still have to be subtracted. Therefore, revenue minus costs is called "Net operating income.")

Besides the problem that Total Absorption Costing can lead to wrong decisions, some scholars consider it logically incorrect since Total Absorption Costing redistributes the indirect costs **equally** over products and services

(continued)

instead of looking at what caused the costs [51, 99]. This step might make a product or service look more expensive than it really is and cause its discontinuation even if it is profitable.

To overcome this problem, **Activity-Based Costing** distributes indirect costs over products and services looking at the activities a product or service needs to be produced or provided. Activity-Based Costing identifies the performed activities and attributes the consumption of resources to those activities [110]. It is based on the idea that activities cause costs and it is possible to match these activities to the products and services that require them [23]. Matching products and services to activities and then activities to resource consumption costs identifies the reasons behind the cost of a given product or service [110].

Table 11.9 Example of the calculation of the contribution margin and the net operating income

	Cost	Model A	Model B	All models
	Total revenue per bike	€700	€1,100	
×	Total number of bikes produced	150	80	
=	Total revenue	€105,000	€88,000	€193,000
	Material per bike	€200	€500	
+	Labor per bike	€60	€120	
=	Total variable costs per bike	€260	€620	
=	Total variable costs	€39,000	€49,600	€88,600
	Total revenue	€105,000	€88,000	
−	Total variable costs	€39,000	€49,600	
=	Contribution margin	€66,000	€38,400	€104,400
	Rent			€24,000
+	Heating			€1,000
+	Marketing			€10,000
+	Administration			€50,000
=	Total fixed costs			€85,000
=	Contribution margin			€104,400
−	Total fixed costs			€85,000
=	Net operating income			€19,400

Table 11.10 shows an example of an ABC analysis for a customer service department.

(continued)

Let us assume it is the customer service department of the bike store described above. The €50,000 indirect costs for administration are attributed to three main activities: processing customer orders, handling customer inquiries, and performing credit checks. The costs are attributed according to the amount of time these activities take. For example, the assigned cost to process customer orders equals to $\frac{€50,000}{100} \times 70 = €35,000$. Then, after we know that this activity was executed 500 times, we know that, on average, one customer order costs $\frac{€35,000}{500} = €70$.

The idea is to use the calculated cost-driver rates to estimate the cost of future projects by just evaluating the activities we will perform. For example, Fichman and Kemerer [34] used Activity-Based Costing to study reuse activities and its cost structure.

Table 11.10 Activity-Based Costing example [63]

Activity	% of time spent	Assigned cost	Activity quantity	Cost-driver rate
Process customer orders	70 %	€35,000	500	€70.00
Handle customer inquires	10 %	€5,000	1,000	€5.00
Perform credit checks	20 %	€10,000	400	€25.00
Total	100 %	€50,000		

Activity-Based Costing attributes all costs (direct and indirect costs) to the produced products and services and therefore suffers from the same problems as Full Absorption Accounting. It violates the principle that costs should be attributed to products and services only if their production or provision caused the costs. Therefore, the calculated costs are useless to take any cost-related decision about the produced products or services [99]. Nevertheless, taking the perspective of the performed activities and—grouping all activities needed to obtain some outcome—processes and studying how activities and processes consume resources **are** useful to understand the causes behind the fixed costs, and to decide how fixed costs can be reduced, converted into variable costs (e.g., changing technology), or removed.

Following this line of thought, "Lean Accounting"—a cost accounting method based on Activity-Based Costing—maps costs to value streams [77].

Activity-Based Costing (as Total Absorption Costing) implements a "push" approach to cost accounting: it distributes (pushes) all costs over the cost-items. In Table 11.10 we distributed the costs that the customer service department generated over 100 % of the time that this department spent. We then look how many percent of the total time the department spends for the

(continued)

different activities to determine the costs that have to be distributed over the specific activities.

Time-Driven Activity-Based Costing [63] implements a "pull" approach to cost accounting [1]. Time-Driven Activity-Based Costing does not anymore distribute the costs over all the available activities, but introduced the concept of "standard costs."

Standard costs are estimated or expected costs, used instead of the real costs to do calculations in cost accounting. Once the real data are available, it is then interesting to study the differences between the actual costs and the standard costs and why these differences occurred. If the standard costs are lower (higher) than the actual costs, the profit will be lower (higher) than expected.

In Time-Driven Activity-Based Costing, we split the cost driver into [63]:

- cost per time unit of supplying resource capacity and
- unit times of consumption of resource capacity by products, services, and customers.

This means that instead of determining the cost-driver rate as in Table 11.10, we have to:

1. determine the capacity of each resource,
2. determine the cost per unit of using that resource, and
3. estimate the standard unit times of consumption of resource capacity.

In our example we have to identify the capacity of the customer service department. Let us say that this department consists of one employee and that per year we have about 250 working days. In this case, if we work 7.5 h per day ($8\,\text{h} - 2\times\frac{1}{2}\,\text{h}$ of break), we have a capacity of $1 \times (250 \times 7 \times 60) = 105{,}000$ min of work.

To determine the cost per unit of using the customer service department, we consider the generated costs of €50,000. Therefore, the cost of 1 min of work of this department is:

$$\frac{\text{total costs}}{\text{total capacity}} = \frac{50,000}{105,000} = €0.48.$$

Finally, we estimate the standard unit times of consumption for each activity as in Table 11.11.

What remains is to determine how many times we performed each activity. Once we know this number, we have to multiply it with the cost to determine the cost assigned to this activity. Table 11.12 shows the analysis for our example.

(continued)

This type of analysis shows the advantages of Time-Driven Activity-Based Costing: we do not anymore distribute the costs over the available time, but activities consume resources using standard cost rates. In this example, we have different possibilities to evaluate Table 11.12:

- The customer service department is really underutilized: we can assign new responsibilities to this department, i.e., new activities to lower the unused capacity.
- The estimated unit times are too low; they do not reflect the reality. In this case, we either increase the unit times or we investigate **why** they are higher than expected.

Table 11.11 Standard unit times of consumption for each activity

Activity	Standard unit times	Cost-driver rate
Process customer orders	10 min per order	$10 \times 0.48 =$€4.80 per order
Handle customer inquires	15 min per inquiry	$15 \times 0.48 =$€7.20 per inquiry
Perform credit checks	60 min per check	$60 \times 0.48 =$€28.8 per credit check

Also in Time-Driven Activity-Based Costing, when estimating we have to be aware (as mentioned in Chap. 9) that high precision often comes with a high cost. Spending a lot of time to find the "perfect" cost-driver rates or standard cost rates is not necessary. Usually, the objective is to be approximately right, rather than precise [63].

Table 11.12 Time-Driven Activity-Based Costing example [63]

Activity	Quantity	Unit time	Total time used (in minutes)	Cost-driver rate	Total cost assigned
Process customer orders	500	10	5,000	€4.80	€2,400
Handle customer inquires	1,000	15	15,000	€7.20	€7,200
Perform credit checks	400	60	24,000	€28.80	€11,520
Total used			49,000		€21,120
Total supplied			105,000		€50,000
Unused capacity			56,000		€28,880

In this case study, we describe how we supported an organization to implement a Time-Driven Activity-Based Costing approach.

11.5.1 Theoretical Framework

In this case, we assume that:

- Development costs are mainly determined by labor costs since writing software is a labor-intensive activity and hardware costs are low compared to personnel costs [56, 102].
- The time one spends to develop is mainly determined by the actual writing, rewriting, and refactoring of code in front of a computer.
- Developers spend the most time of their day in front of the computer, which means that using non-invasive measurement probes, we are able to track a significant part of the effort. Not all time is used to code, but also to communicate with others, gather information, etc.
- It is possible that one person works on several computers. This means that we have to remove the effort that is counted twice.

The concept map for this case, as in the previous case, is the one depicted in Fig. 10.2. The roles that the different concepts in this case have are the same as in the previous case study, with the following differences:

- Measurement: we measure how much time a user spends modifying an artifact as well as descriptive properties of the currently modified artifact, e.g., the name, containing folder, size, etc.
- GQM$^+$Strategies model: defines which costs are measured and how they are measured.
- Information: the list of artifacts with the amount of time each user modified it.
- Andon: visualizes the cost distribution within the code base.
- Knowledge and Wisdom: collects cost information of past projects to improve the estimation of cost drivers (see below) and, therefore, the estimation of the costs of future projects.
- Experience Factory: we store the cost accounting data of current and past projects.

The GQM$^+$Strategies element in this case is constructed as follows:

- **Organizational goal**: track software development costs using a cost accounting approach. Using the GQM$^+$Strategies goal template, this goal can be refined as follows:
  - **Object**: the software development process
  - **Focus**: costs
  - **Magnitude**: all costs generated by the work of the software development teams are tracked and assigned to its cost item.
  - **Time frame**: continuously
  - **Organizational scope**: software development division

- **Assumptions**:
  - Development costs are mainly determined by labor costs.
  - Software development labor costs are mainly determined by interactions with the computer.
  - Developers can work on more than one computer at the same time.
- **Constraints**:
  - To reduce measurement costs, we favor non-invasive measurement over manual measurement.
- **Strategy**: track the amount of time developers spend working on artifacts and assign this time to the modified artifact.

The associated GQM graph consists of the following elements:

- **Measurement goal**: analyze the software development process for the purpose of evaluation with respect to the costs from the point of view of the software developer in the context of the software development team (follows the GQM goal template of Fig. 7.5)
- **Questions**:

1. Which artifacts are developers working on?
2. How much effort do developers spend working on the different artifacts?
3. What caused the effort, i.e., what is the cost item?
4. At which computers are developers working?

- **Metrics**:

1. For question 1 we extract properties useful to identify the artifact:
   - the application name of the focused artifact,
   - the project name of the focused artifact,
   - the file name of the focused artifact,
   - the package name of the focused artifact,
   - the class name of the focused artifact,
   - the class annotations of the focused artifact,
   - the implemented interfaces by class of the focused artifact,
   - the method signature of the focused artifact, and
   - the method annotations of the focused artifact.
2. For question 2:
   - the time spent working on the focused artifact.
3. For question 3:
   - the metrics collected because of question 1 (to be able to attribute effort to source code items, e.g., a class that represents a feature) and
   - tasks currently assigned to the developer (to be able to attribute effort to the task that caused it).

4. For question 4:
   - log-on/log-off time reported by the operating system.

- **Interpretation model**: we consider the goal satisfied if we are able to track at least 90 % of the daily working time of every developer.

11.5.2 The Study

The company for which we implemented non-invasive cost accounting traditionally keeps track of the development effort of each developer by hand. Every developer uses a company-internal website to record on what he is currently working. The recorded data are similar to the data in Table 11.13.

Table 11.13 Manually entered data to keep track of costs in case 2

Date	From	To	Project	Notes
22.04.2013	08:00	09:00	Project A	Implemented feature #24
22.04.2013	09:00	11:00	PDF library	Implemented feature #15
22.04.2013	11:00	12:00	Project C	Discussion about requirements
22.04.2013	14:00	15:00		Group meeting
22.04.2013	15:00	16:00	Project B	Implemented feature #72
22.04.2013	16:00	18:00	Project C	Resolved issue #251
23.04.2013	08:00	12:00	Project A	On-site training
23.04.2014	14:00	18:00		Studied how to develop software for Android

The reported effort is either:

- development effort
  - for a specific project (as in line 1 in Table 11.13),
  - for a common library used by the team, therefore not attributable to a specific project (as in line 2 in Table 11.13),
- effort for supporting activities (e.g., meetings, learning)
  - for a specific project (as in the lines 3 and 7 in Table 11.13), or
  - not for a specific project (as in the lines 4 and 8 in Table 11.13).

Such an approach does not evaluate the costs of software development; it just keeps track of the consumed resources without assigning them to the produced

product or delivered service. In fact, the company did not know how much a module of a given piece of software costs; it just knew how much it costs altogether, how many resources were consumed, and for which tasks they were consumed.

Using this approach, we do not know **where** we invested our money; we just know **how much** (the salaries of our programmers) and **why** (the tasks that they were performing). Moreover, in our case the programmers were documenting the task they were performing in a textual comment, which made it impossible to perform any automated analysis over the recorded data.

The company was using Subversion [5] to store the source code in a central location and to handle the common work on the source code by two development teams. Moreover, after each completed task, team members were advised to write the task ID of the just completed task into the commit message. This practice helps to increase the traceability between tasks and source code, i.e., I know by which tasks which source code was modified, but it does not tell me how much effort the modification caused [44, 45].

Knowing how much we invested and if and how much we are still investing in some technology is helpful to:

- estimate the skills of the team and decide on how to improve,
- decide whether the proposal of some programmer or consultant to "rewrite the module in one day" is reasonable,
- estimate the complexity of some technology,
- estimate the required testing effort to ensure that some technology works,
- estimate the maturity of some technology.

We saw three solutions to estimate the effort behind produced source code:

1. estimate the effort based on cost drivers such as estimated size or complexity (as e.g., in COCOMO II [13]),
2. estimate the effort based on the manual time sheet entries and the manual entries in the commit log, or
3. measure the effort using a non-invasive plugin.

Cost estimation is a valid instrument to estimate the overall development cost of some module. It is used to get a rough estimate of future costs to know what to expect. On the other hand, cost estimation is not suited to estimate the costs of particular aspect or to discover yet unknown sources of costs. Cost accounting (and the company interested in non-invasive cost accounting) aims to document past costs to identify such yet unknown sources of costs and their reasons. Therefore, we excluded option 1.

We excluded also option 2 for the reasons mentioned in Chap. 9, i.e., that we did not want to distract the programmers from doing what they can do best: designing and developing software.

Cost accounting distinguishes between cost-type accounting (identifying all costs we generate in the organization) and cost-unit accounting (attributing them to a production unit, i.e., linking them to their reason) [99].

To implement cost-type accounting, we implemented similar measurement probes as described in Sect. 11.4:

1. a plugin for Microsoft Visual Studio [81] that tracks the current method (together with the namespace, the class name, the file name, and the project),
2. a plugin for Eclipse [30] that tracks the current method (together with the package, the class name, the file name, and the project), and
3. an application written in Microsoft C# [79] that tracks the current application and the current document (if the application shows the current document in the caption) through the operating system API.

The measurement probes were implemented as depicted in Fig. 9.13. The probes 1 and 2 were implemented as plugins for the hosting application. The plugin for Eclipse is written in Java [86] and the plugin for Microsoft Visual Studio in C#. The plugins regularly poll the current method, package, namespace, file, project, or document from the hosting application and report changes to the measurement framework.

The measurement probe 3 is not strictly needed in this case; we kept it for debugging reasons to understand what is happening on the machines in general.

Table 11.14 shows sample data as it is collected by the three described probes.

Table 11.14 Example data of case 2

Machine	application	Item	Duration	Probe
1	Eclipse	at.company.tools.Action.run()	10:00	2
2	Eclipse	at.company.tools.ActionTest.test1()	12:00	2
2	Eclipse	at.company.tools.ActionTest.test2()	7:30	2
1	Google Chrome	http://www.codeproject.com	2:00	3
3	Google Chrome	http://stackoverflow.org	7:00	3
4	Microsoft Visual Studio	com.company.project7.Settings.save()	8:00	1
4	Microsoft Visual Studio	com.company.project7.charts.Bar.draw()	15:00	1
4	Microsoft Visual Studio	com.company.project7.print.Print(Chart c)	17:30	1

The data collected by the probes 1 and 2 represent the coding effort developers spend interacting with the computer. As in Sect. 11.4, we did not directly collect all the effort a developer spends during the day since we assumed that the benefits of knowing what the developers were doing in that time would not justify the costs of collecting that additional data. In terms of cost accounting, we considered that time an indirect cost.

We store the collected coding effort data using a hierarchical data structure as depicted in Fig. 11.19. A node is either a project, a file, a package, a namespace, a class, or a method. Nodes can have child nodes and properties (depicted as little nodes attached to the nodes in the hierarchy such as "language" or "effort").

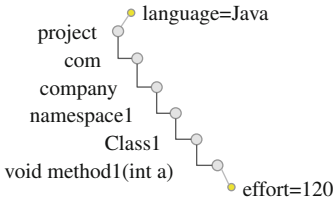


Fig. 11.19 One hundred and twenty seconds of effort spent editing "method1" in the class "Class1"

Figure 11.19 depicts 120 s of effort a user had to modify the method "method1" in the class "Class1," which is in the package "com.company.namespace1" and is part of the project "project." Figure 11.20 represents another example in which a user edited "method2" in the class "Class2."

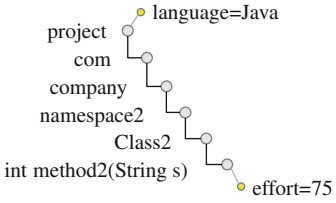


Fig. 11.20 Seventy five seconds of effort spent editing "method2" in the class "Class2"

To obtain the total effort spent by one user, we merge the threes as in Fig. 11.21:

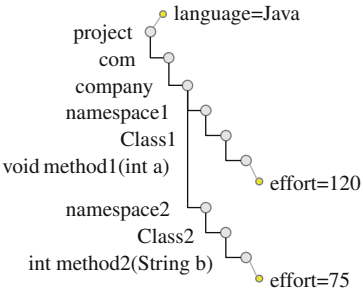


Fig. 11.21 Seventy five seconds of effort spent editing "method2" in the class "Class2"

As we implemented the first version of non-invasive cost-type accounting, we noticed that some developers (apparently) were able to work more than 24 h per day. This was because these developers were using two or even more machines at the same time, and we were just summing up the effort collected on all machines. The total amount of time recorded on each machine was correct, but we wanted

to measure the amount of work in terms of human effort, not computational effort. Therefore, we decided to merge the effort per person in the following way.

Let us assume that one user has three machines. He uses two to develop and one to browse and e-mail. Figure 11.22 shows such an example:

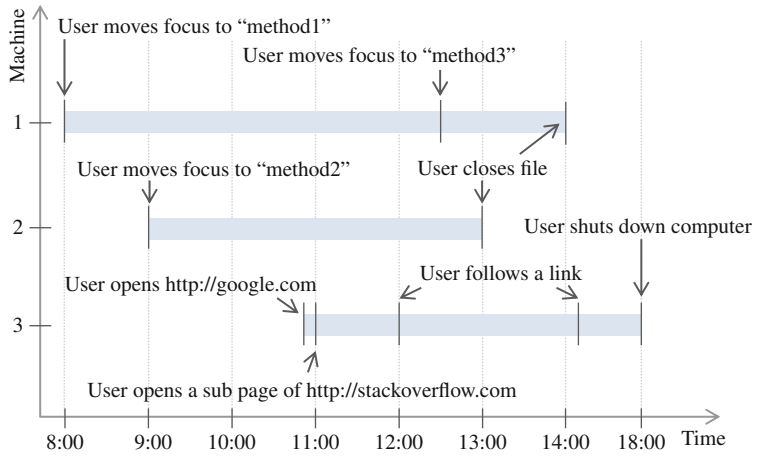


Fig. 11.22 Possible effort distribution between three machines

The user depicted in Fig. 11.22 begins the day opening the source code of "method1" on machine 1. He sees that there is something wrong and begins to look for the error. He opens "method2" on machine 2 but cannot find the problem. He opens a browser on machine 3 and searches for a possible solution on http://google.com. He finds a possible solution on http://stackoverflow.com, closes "method2," and edits "method3." He sees that everything works now and closes the file on machine 1.

Our initial measurement probes report effort from all three machines as in Table 11.15.

The total effort that our probes recorded from 8:00 to 18:00 is 17 h and 10 min. Since we wanted to estimate the human effort and not the computational effort, we decided to work on a way to merge the work one user performs on all his machines to obtain the effort per person.

The problem of counting effort multiple times arises from overlapping events. If two events are not overlapping, the aggregated time line can be calculated simply by copying the events from all machines (see Fig. 11.23).

To remove the overlapping parts of two events, we adopted the following rule: "if two events overlap, modify the older event so that they do not overlap anymore."

We distinguished two cases:

- The older event ends **before** the new event ends (see Fig. 11.24): we shorten the older event till the beginning of the newer event.

Table 11.15 Data reported for the events depicted in Fig. 11.22

Machine	Application	Item	From	To	Duration
1	Eclipse	method1	8:00	12:30	4:30
2	Eclipse	method2	9:00	13:00	4:00
3	Browser	http://google.com	10:50	11:00	0:10
3	Browser	http://stackoverflow.com/...	11:00	12:00	1:00
3	Browser	http://stackoverflow.com/...	12:00	14:10	2:10
3	Browser	http://stackoverflow.com/...	14:10	18:00	3:50
1	Eclipse	method3	12:30	14:00	1:30

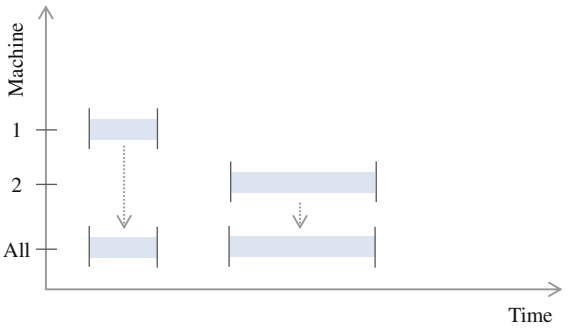


Fig. 11.23 Two events that are not overlapping

- The older event ends **after** the new event (see Fig. 11.25): we split the older event into one part until the newer event and one part after the newer event.

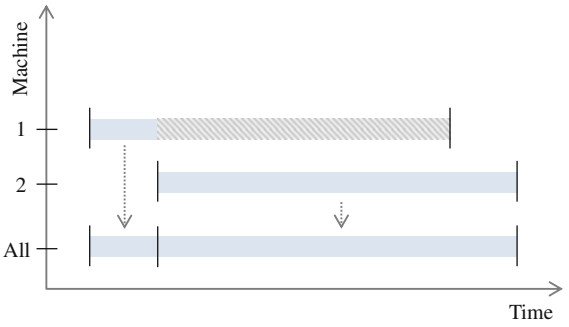


Fig. 11.24 The older event begins before the newer event and ends before the newer event

This aggregation process was performed reading all events from the database and inserting them according to the following algorithm:

1. Insert the first event (see Fig. 11.26).

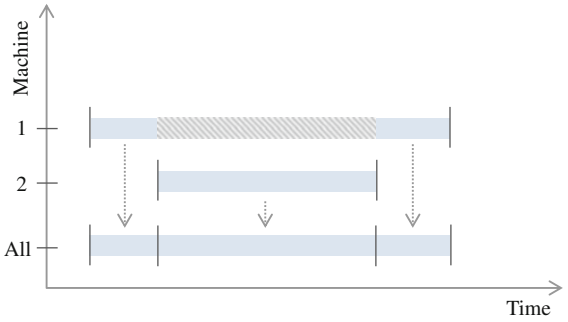


Fig. 11.25 The older event begins before the new event but ends after the new event

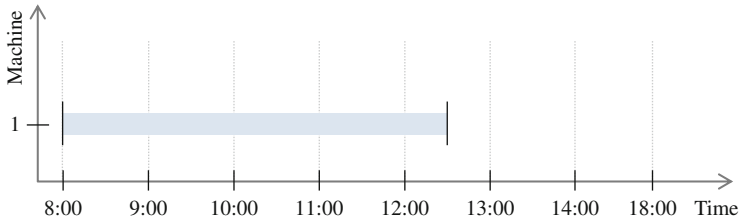


Fig. 11.26 The first event was inserted

2. Insert the next event and remove all overlapping parts of older events (see Fig. 11.27).

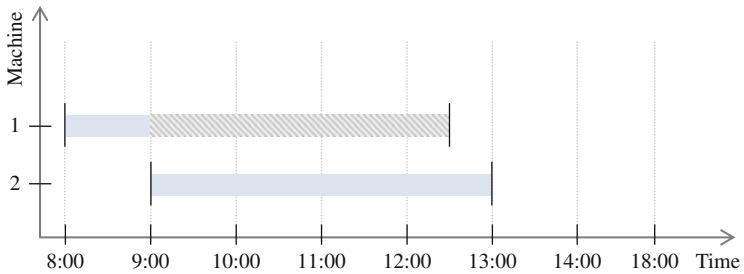


Fig. 11.27 The second event was inserted and the overlapping parts of the older event removed

3. Continue at point 2 until all events are inserted (see Fig. 11.28 for an example of an insertion of a third event).

According to this algorithm, the events in the example of Fig. 11.22 would be inserted as in Fig. 11.29. The corresponding data are listed in Table 11.16.

As we implemented this first idea and used it for several months, we noted that its performance was very slow. This algorithm requires that on every insertion, all older events are checked and (in case they overlap) are modified. For example, it could

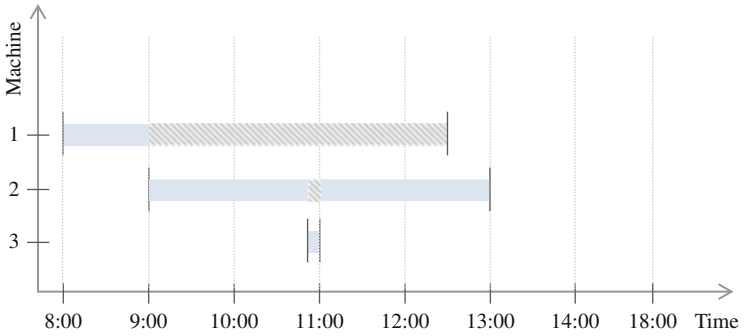


Fig. 11.28 The third event was inserted and the overlapping parts of the older event removed

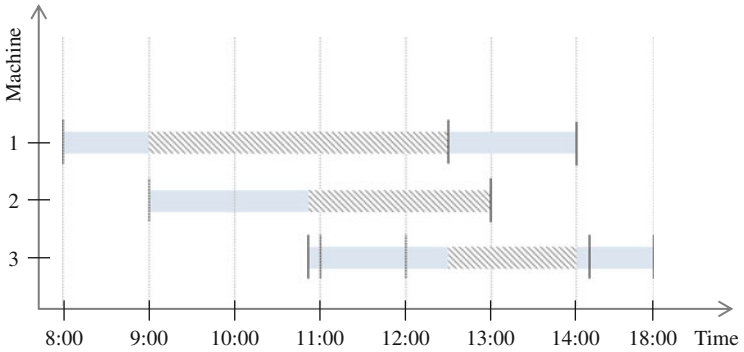


Fig. 11.29 First attempt to merge the effort of one user working on multiple machines

Table 11.16 Data reported for the events depicted in Fig. 11.29

Application	Item	From	To	Duration
Eclipse	method1	8:00	9:00	1:00
Eclipse	method2	9:00	10:50	1:50
Browser	http://google.com	10:50	11:00	0:10
Browser	http://stackoverflow.com/...	11:00	12:00	1:00
Browser	http://stackoverflow.com/...	12:00	12:30	1:30
Eclipse	method3	12:30	14:00	1:30
Browser	http://stackoverflow.com/...	14:00	14:10	1:30
Browser	http://stackoverflow.com/...	14:10	18:00	3:40

happen that some developer was working on his laptop while not connected to the Internet; therefore, his data was not uploaded for some time. When this developer reconnected to the Internet, all past data had to be reevaluated to obtain the true picture of the development effort.

What we did not consider is the amount of data that our system generated. In Fig. 11.30, we depict the number of events collected from January 22, 2013 to July 15, 2013 (ca. 6 months). During this time, 41 users generated 1,232,030 events.

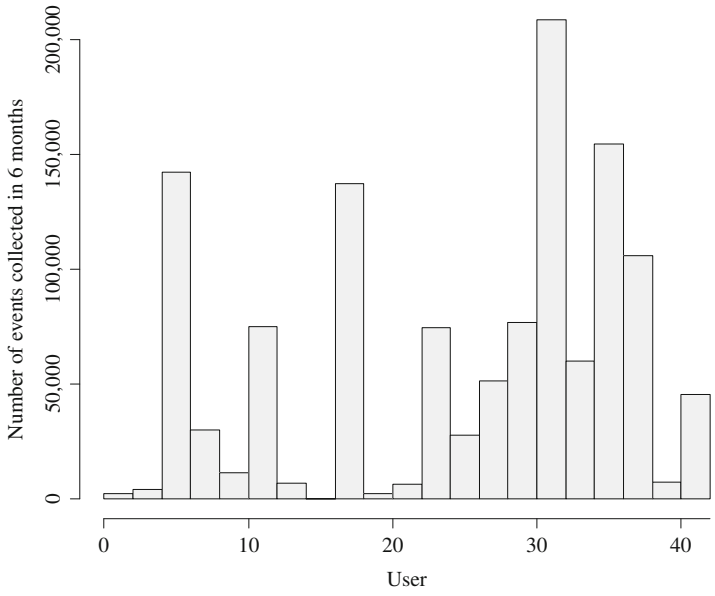


Fig. 11.30 Number of events uploaded by 41 users in 6 months

We see that the frequency with which users change focus during their work varies. Some reasons for this are that users use the computer in different ways, because of different working styles or because of a different experience in using the software [22].

To cope with the amount of data generated every day, we modified our measurement strategy as follows. We did not measure the effort anymore individually on each machine, but we collected only the focus changes on each machine.

A focus change occurs when a user changes the focus (i.e., the cursor or the selection) from one item to another. What an "item" specifically is, is determined by the measurement probes. In our case we captured the current method, together with the package, the namespace, the class name, the file name, the project name, and the project language.

As we merge all the focus changes of one user, and we sort it, we obtain the sequence of focus changes, i.e., a sequence of items on which the user moved his attention.

Figure 11.31 depicts this approach: all focus changes are projected to a common time line representing all machines. Once we copied all focus changes to the merged time line, we calculate the time differences from one focus change to the other to obtain the effort.

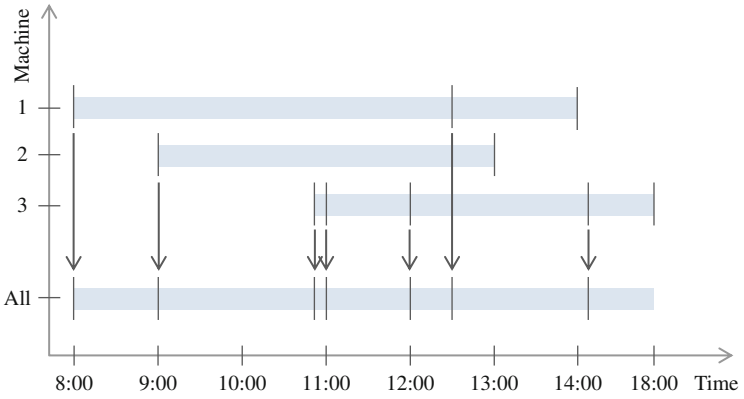


Fig. 11.31 Merging the effort based on focus changes

The effort resulting from the events on the common time line is shown in Table 11.17.

Table 11.17 Data reported for the events depicted in Fig. 11.31

Application	Item	From	To	Duration
Eclipse	method1	8:00	9:00	1:00
Eclipse	method2	9:00	10:50	1:50
Browser	http://google.com	10:50	11:00	0:10
Browser	http://stackoverflow.com/...	11:00	12:00	1:00
Browser	http://stackoverflow.com/...	12:00	12:30	1:30
Eclipse	method3	12:30	14:10	1:40
Browser	http://stackoverflow.com/...	14:10	?[a]	?[b]

[a]We do not know how long this event lasts since there is no event that starts after this one
[b]Since we do not know till when the event lasts, we cannot calculate the duration of this event

The computational effort for this new approach was much lower since the detection of overlaps—which required that every new event was compared with all older events of that day—is not required anymore. Three more aspects of the approach are illustrated in Fig. 11.31 and Table 11.17:

1. We only project focus change events to the common time line: we ignored the events in which the user closed a file or shut down the computer.
2. For the last event of the time line, the length is not defined: since after the last event there is no more focus change, we cannot calculate its length. We have to remove it.
3. If no focus change occurs after an item was closed, the effort is attributed to the closed item until a focus change occurs: "method3" was closed at 14:00 and at 14:10 the user continued browsing. In our second approach, we attribute the 10 min between 14:00 and 14:10 to "method3" instead of "method2" (see Table 11.17).

Aspect 3 is alleviated if users change their focus frequently. To verify this condition, we analyzed the collected data depicted in Fig. 11.30 looking at the duration of events, i.e., the time between one focus change and another. For this data the median was 2 s, which means that the most frequent events had a duration of 2 s.

To get a better understanding of the distribution of the duration of events, we used a visualization called hexagon binning [73], a form of bivariate histogram, useful for visualizing the structure in datasets with large n. The concepts of hexagon binning we used are:

1. we tesselate the xy-plane over the set (range*(x)*, range*(y)*) by a regular grid of hexagons,
2. we count the number of points falling in each hexagon, and
3. we plot the hexagons using a color gradient in proportion to the counts.

The result is depicted in Fig. 11.32. According to this visualization, the majority of events are very short, even though there are events that last up to 30 min.

Based on these results, we decided to adopt the new algorithm.

So far, we developed a cost-type accounting (identifying all costs we generate in the organization) method to collect the effort per person during the software development process. As a next step, we worked on cost-unit accounting, i.e., attributing the collected effort to a development unit, i.e., linking the costs to their reason.

The "right" attribution of effort to the cost object, i.e., the item that we consider responsible for that cost, depends on the goals of the organization.

In this case, the company—in parallel to the development projects for different clients—was developing and maintaining a class library that they called the "platform." This class library was the place where the team collected all sorts of code that was frequently needed. Using the terms of knowledge management described in Chap. 5, it was the place where the team embedded knowledge and wisdom. Refining and extending "the platform" was seen as a way to standardize and disseminate new ideas for improvement (as depicted in Fig. 2.18). For the rest of the case study, we will refer to this class library simply as platform.

From a cost accounting perspective, time invested in the platform was not directly attributable to one project or client; hence, those costs had to be considered indirect costs.

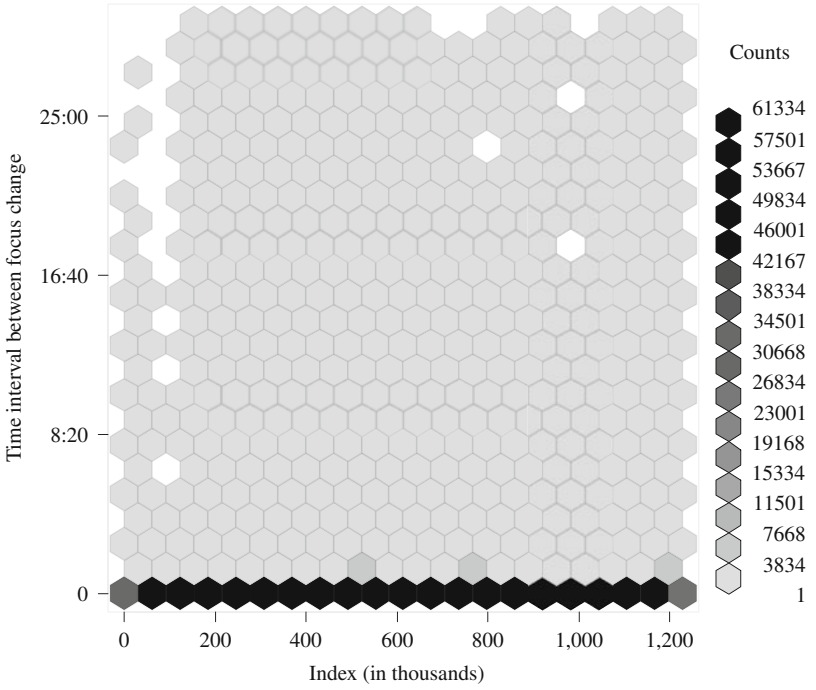


Fig. 11.32 Hexagon binning visualizing the time intervals between focus changes of ca. 1.2 million events

Using the developed measurement probes, we were able to collect the editing time and attribute it to the source code. In this case, the cost-item (the item to which we want to attribute the costs) was the project.

Capitalization of Development Costs [98]

Similar to cost accounting, financial accounting aims to record business transactions and operations [40], but while cost accounting is intended for stakeholders internal to the business, financial accounting aims to inform stakeholders external to the business. While organizations are free to adapt cost accounting to their needs, the rules on how an accountant has to perform financial accounting are regulated by law.

In financial accounting, two important concepts are [40, 54]:

- The balance sheet: it shows the financial situation of the business at a single point in time. It compares all things an organization owns (assets) with all the things an organization owes (liabilities) to others.

(continued)

- The profit and loss account: it shows the financial performance of the business over the past accounting period. It compares expenses with revenues.

As we said, the balance sheet describes a given point in time, and the profit and loss account describes a time span. At the end of an accounting period, we use the profit and loss account to calculate the difference between revenues and expenses. This difference is then called (this should not be a surprise now) the profit (if it is positive) or the loss (if negative).

Let us see what happens when we incur an expense in financial accounting. For example, we pay the salaries of our employees. The initial situation is depicted in Fig. 11.33 in which we depict the balance sheet and the profit and loss account.

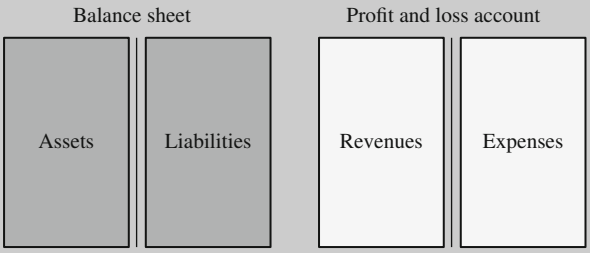


Fig. 11.33 Balance sheet and the profit and loss account

After the expense, the situation changed as it is depicted in Fig. 11.34: the bank account (an asset) is lower than it was before and we have an expense.

If a company incurs many expenses, the assets will continuously diminish. This can be a disadvantage for companies that produce software for their internal use. For external stakeholders, it might look like the organization is spending money without getting anything back. Moreover, the new software that allows the organization to work more efficiently and faster is not recorded anywhere.

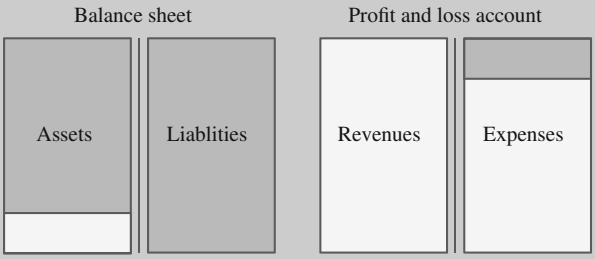


Fig. 11.34 Balance sheet and the profit and loss account after an expense

(continued)

This is different for tangible assets, e.g., a machine. If a company constructs a machine using their own resources, the costs for constructing this machine can be added to the assets part in the balance sheet. This allows, among other things, to [40]:

- Show to external stakeholders that the value of the organization increased.
- Distribute the development costs over the lifetime of the machine: instead of registering all the construction costs as expenses (and maybe incurring a loss in that year), the expenses can be registered so that the profit and loss account does not change, but only the assets increase. Then, every year, a part of the asset is registered as an expense, due to the depreciation of the asset. Depreciation means in this case that the asset loses its value over time because it becomes outdated or (like a machine) deteriorates.

The International Accounting Standard 38 (IAS 38) describes under which conditions a software has to be registered as an asset instead of an expense [26]:

- it is probable that the future economic benefits that are attributable to the asset will flow to the entity and
- the cost of the asset can be measured reliably.

Not all the costs can be considered; the standard distinguishes between research and development costs. Research costs cannot be considered part of the value of the asset; they have to be registered as expenses [26, 53]. If it is not possible to distinguish the research phase of an internal project to create an intangible asset from the development phase, "the entity treats the expenditure for that project as if it were incurred in the research phase only [26]." Some interpretations of this standard claim that this is the case if an organization uses Extreme Programming [10, 28]. As in Extreme Programming phases of research and development alternate (as well the Lean software development approach proposed in this book), the incurred costs have to be counted as expenses.

The measurement probes presented in Chap. 9 can provide help to "reliably separate research from development [10]." ("Reliably" here means "objectively" as explained in Chap. 6.) During the development, non-invasive measurement probes record the time needed to produce the entire source code, together with the source code itself. Once the development is finished, the source code contained in the final product is used to split the total expenses in two parts:

1. the expenses for producing the source code present in the final version of the product count as development costs and
2. the expenses for producing source code that was deleted or modified afterwards count as research costs.

Previously we looked at three ways to attribute costs to the cost-item: Total Absorption Costing, Direct Costing, and Time-Driven Activity-Based Costing.

Knowing the development effort of the entire code base ("development" now again intended as both research and development), we can compute the cost of the project according to all three approaches we discussed above:

- **Total Absorption Costing**: we need to distribute the indirect costs to the cost-items, the projects.
- **Direct Costing**: we only consider the direct costs and add a profit margin that, after a year, should cover also the indirect costs.
- **Time-Driven Activity-Based Costing**: we look at the different activities that take place during the development process and measure the amount of time they consume resources.

We will now go through all three ways of cost accounting and discuss how we implemented Time-Driven Activity-Based Costing for the organization. In this case study we look at two major cost blocks:

- Labor costs: are theoretically direct, variable costs. It is expensive to track them manually, but many organizations require from their employee to fill out some form of time sheet to document what they are doing. If an organization is not doing this, this cost block becomes an indirect, variable cost.
- Platform costs: the development of the shared library called "platform" occurs while there are no other ongoing projects or—if it is urgent—in parallel with other project activities. We consider it an indirect, fixed cost. The platform is developed by the same team that develops the actual projects.

Table 11.18 shows the costs of the last accounting period that we want to analyze using the three costing approaches. For the illustration of Total Absorption Costing and Direct Costing, we assume that the development team is at least tracking manually how much time they dedicate per project. We will explain later how we collect this information automatically.

Table 11.18 Costs for producing projects 1 and 2

Costs	Project 1	Project 2
Labor	€108,000	€252,000
Hardware costs[a]	€10,000	
Platform development costs[b]	€31,050	

[a]Costs to replace obsolete hardware
[b]Costs to replace outdated platform code due to changes of legal constraints

To compute the cost of a project according to Total Absorption Costing, we need to redistribute the development costs of the platform over all projects, based on the size of the project. The assumption is that if a project is large, it makes more use of the platform, and vice versa.

Table 11.19 shows a solution based on Total Absorption Costing. In this case we distribute the hardware costs according to the team size in each project and the platform development cost according to the number of packages.

Table 11.19 Total Absorption Costing solution

	Cost	Project 1	Project 2
	Labor	€108,000	€252,000
=	Total direct costs per project	€108,000	€252,000
	Team size	3	7
	Number of packages	150	120
	Hardware (€10,000 ÷ (3 + 7) = €1,000) per person	€3,000	€7,000
+	Platform (€31,050 ÷ (150 + 120) = €115) per package	€17,250	€13,800
=	Total indirect costs per project	€20,250	€20,800
	Total direct costs per project	€108,000	€252,000
+	Total indirect costs per project	€20,250	€20,800
=	Total costs (direct and indirect) per project	€128,250	€272,800

The team we worked with grouped all classes and interfaces that were meant to be used together to provide one reusable functionality in one Java package. A "component" is intended here as defined in [104], i.e., as a unit of composition with contractually specified interfaces and explicitly stated context dependencies only. Therefore, using the number of packages, we distribute the platform development costs in relation to the provided functionality.

Another reason for this distribution is that according to the experience of the team, larger projects profited more but generated also more new requirements from the platform project.

According to Direct Costing, we do not distribute the indirect costs over the cost-items. Table 11.20 shows the calculation of the net operating income for each project.

In the case of Time-Driven Activity-Based Costing, we have to change our way to look at costs: we have to think which activities occur and which resources they need. In our case we consider two resources: the developer and the platform. For both we continuously generate costs: the developer receives a wage and the platform requires regular updates and maintenance.

Table 11.20 Direct Costing solution

	Cost	Project 1	Project 2	All projects
=	Total revenue	€200,000	€300,000	€500,000
	Labor	€108,000	€252,000	
=	Total variable costs	€108,000	€252,000	€360,000
	Total revenue	€200,000	€300,000	
−	Total variable costs	€108,000	€252,000	
=	Contribution margin	€92,000	€48,000	€140,000
	Hardware costs			€10,000
+	Platform costs			€31,050
=	Total fixed costs			€41,050
=	Contribution margin			€140,000
−	Total fixed costs			€41,050
=	Net operating income			€98,950

So far, the non-invasive measurement probes are able to collect the effort spent on code, but we are not (yet) able to relate this effort to the project.

To accomplish this, we instrumented an existing system that the team was using: a software-based Kanban board (see Chap. 10). The team had it developed internally to manage its tasks.

Examples for a task are the implementation of a requirement, the correction of defective code, or the update of the database software on the server. All tasks were stored in a central database, from which we regularly extracted data to create our own activity log.

The instrumentation of the Kanban board allowed us not only to link development effort to a project but to each single task. The choice reflected the costing requirements of the organization that was interested to know how much each task costs the team. Such information is useful to:

- determine the price of a project based on the costs of implementing each requirement,
- price additional requirements requested by the customer, or
- estimate the cost of future projects based on the costs of similar activities performed in the past.

An example of data we collected is shown in Table 11.21.

Table 11.21 Data extracted from the Kanban board software

Task	Date	Status
125	4.2.2013	Backlog
125	6.2.2013	Selected
125	7.2.2013	Develop
125	7.2.2013	Done
125	8.2.2013	Deploy
125	11.2.2013	Live!
127	11.2.2013	Backlog
128	11.2.2013	Backlog
129	11.2.2013	Backlog
128	13.2.2013	Deleted

Using this data we can reconstruct the life cycle of a task as depicted in Fig. 11.35. The task in Fig. 11.35 was added to the "Backlog" and followed the expected life cycle from "Backlog" to "Live!."

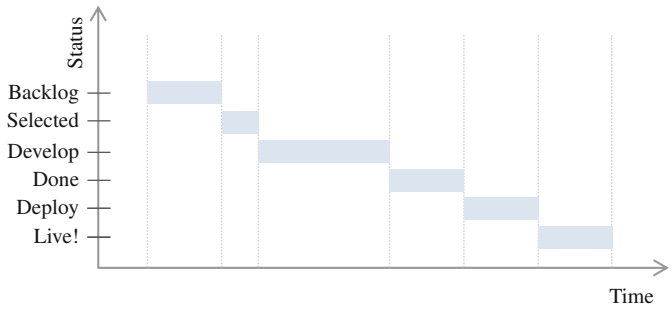


Fig. 11.35 Life cycle of a task

Since some tasks get deleted before being completed, we added an additional state "Deleted" to be able to reconstruct the life cycle of such tasks, too (see Fig. 11.36).

We now combined the information about when tasks are under development (when they have the status "Development") with the data coming from our non-invasive measurement probes as depicted in Fig. 11.37. We could also say: the activities measured by our measurement probes (the time spent editing artifacts)

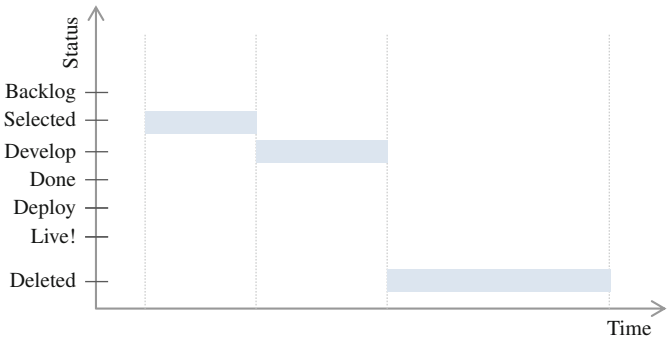


Fig. 11.36 Life cycle of a task that was deleted before being finished

are assigned to activities at a higher level, meaningful for the organization. We will now explain Fig. 11.37 step-by-step.

Figure 11.37a depicts the life cycle of task ① and task ② (similar to Fig. 11.35 and Fig. 11.36). Task ② was added to the backlog and selected as next task to develop before task ①, but the developer picked task ① first, implemented it, and then implemented task ②.

Figure 11.37a depicts the activities that the developer performs (similar to the data in Table 11.14). We see that:

- after the developer set task ① to "Development," he modified method d(), modified class A, and modified file C, and that
- after the developer set task ② to "Development," he used the application "Skype" and modified method b().

Knowing when a specific task had the status "Development," helps us to attribute lower-level activities (such as editing code) to higher-level activities (such as performing a task). Figure 11.37c depicts how we attribute the lower level activities to tasks. An example of the data that we can obtain with this approach is shown in Table 11.22.

It could happen that a developer had more than one task in the development state at the same time, a situation that is discouraged by the Kanban approach. In such case, we split the development effort, which occurred while the tasks were under development, into equal parts and assign to each task one part of the effort.

Using this approach, labor effort could be determined directly, without the need to log manually what everybody was doing (except updating the status of the Kanban board). To distribute the platform development costs, we used Time-Driven Activity-Based Costing.

For this purpose we have to begin defining what we consider an activity. The choice of what to consider an activity depends on the goals and information needs of the organization. Finding out how much an activity costs has to be useful for the organization. In our case, the term "activity" might refer to:

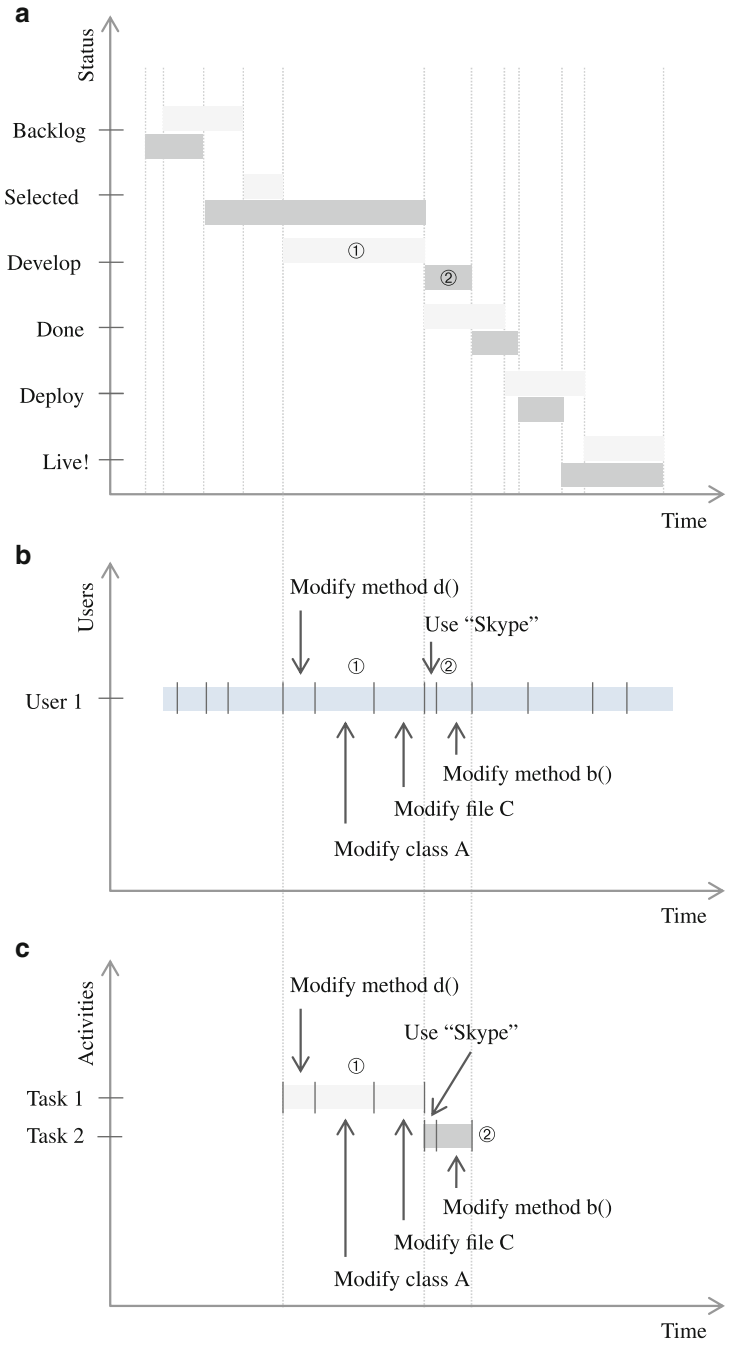


Fig. 11.37 Linking the data coming from non-invasive measurement probes to tasks

Table 11.22 Effort data together with the active task

Task	Application	Item	From	To	Duration
56	Eclipse	method1	8:00	9:00	1:00
56	Eclipse	method2	9:00	10:50	1:50
56	Browser	http://google.com	10:50	11:00	0:10
57	Browser	http://stackoverflow.com/...	11:00	12:00	1:00
57	Browser	http://stackoverflow.com/...	12:00	12:30	1:30
57	Eclipse	method3	12:30	14:10	1:40

- modifying files,
- clicking on some item on the screen,
- communicating with another team member,
- importing a package,
- creating a test case.

For example, Fichman and Kemerer [34], who investigate the use of Activity-Based Costing to distribute reuse costs over cost-items, use the following 13 reuse activities:

1. develop reuse infrastructure,
2. maintain reuse infrastructure,
3. communicate existence of components,
4. administer reuse measurement, accounting and incentives,
5. analyze reuse opportunities,
6. develop or acquire reusable components,
7. certify components,
8. document, classify and store components,
9. search for components,
10. retrieve, understand and evaluate components,
11. adapt and integrate components,
12. maintain reusable components, and
13. update reusable components.

To distribute platform costs over the projects, we wanted to find an activity that reflected the usage of the platform. An indicator for the usage is when another developer refers to the platform using an "imports" statement and then instantiates a class from the imported package. Our idea was therefore to observe how much other projects made use of the components, i.e., packages provided by the platform.

To accomplish this, we developed a tool to scan over all Java files and collect all package declarations (the same can be accomplished using JDepend [58]). The result of such an analysis is shown in Table 11.23 (we anonymized all package names).

Table 11.23 Dependency matrix for a project

Packages on which project packages depend	Project package													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Platform package 1	×						×							
Platform package 2	×			×	×		×							×
Platform package 3	×		×	×						×				×
Platform package 4	×	×	×	×	×	×	×	×	×	×	×		×	×
Platform package 5														×
Platform package 6	×													
Platform package 7	×	×	×	×	×	×	×	×	×		×		×	
Platform package 8	×		×	×		×				×		×		
Platform package 9										×				
Platform package 10			×											
Platform package 11	×			×	×		×							
Platform package 12					×	×	×			×				
Platform package 13	×									×				
Platform package 14				×										
Platform package 15	×													
Platform package 16	×													
3rd party library package 1													×	

(continued)

Table 11.23 (continued)

Packages on which project packages depend		Project package													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
3rd library package 2	party													×	
3rd library package 3	party													×	
3rd library package 4	party	×		×											
3rd library package 5	party										×			×	
3rd library package 6	party			×											
3rd library package 7	party			×											
3rd library package 8	party			×											
3rd library package 9	party												×		
Java Library package 1	Class	×	×	×	×	×		×	×	×	×	×	×	×	×
Java Library package 2	Class	×	×	×	×	×	×	×	×	×	×	×	×	×	×

(continued)

Table 11.23 (continued)

Packages on which project packages depend		Project package													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Java Library package 3	Class										×				
Java Library package 4	Class													×	
Java Library package 5	Class													×	
Java Library package 6	Class	×	×		×			×							
Java Library package 7	Class		×	×	×			×							
Java Library package 8	Class	×	×	×	×	×	×	×	×	×		×	×	×	×
Java Library package 9	Class						×								
Java Library package 10	Class	×													
Java Library package 11	Class													×	
Java Library package 12	Class	×	×	×	×	×	×	×	×	×	×	×		×	×

(continued)

Table 11.23 (continued)

Packages on which project packages depend		Project package													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Java Library package 13	Class	×	×	×	×	×		×	×	×	×	×		×	×
Java Library package 14	Class	×													
Java Library package 15	Class													×	
Project package 3		×													
Project package 6		×								×	×				
Project package 10		×		×											
Project package 11							×				×				

The column headers contain all package names of a project, and the row headers contain all package names on which the project packages depend on. Since the company organized packages according to the modules and components of the application, Table 11.23 was useful to understand how much certain components of the project were using components of the platform.

To calculate how much a project used the platform, we just had to examine the references identified by our tool. We did not distinguish if a project used a platform package only once or several times.

This procedure assumes that platform packages are referenced only if they are used. In our case the team used Eclipse, which displays a warning if a package is referenced but not used. We observed that programmers frequently used the keystroke `Ctrl+Shift+O` to automatically add package imports. This shortcut also automatically removes all unnecessary package references. Therefore, we were confident that only packages that were really in use were referenced since the team

had an internal policy to remove all warnings before committing the code to the repository.

Following the logic of Time-Driven Activity-Based Costing, we see the packages of the platform as a provided capacity to the team. If one project uses all available packages, it has to be charged more than another project that uses only one. In this way we can assign the software development costs to projects in relation to the importance of the platform for the projects.

Finally, we can now put all aspects together.

In standard Activity-Based Costing, one option would be to count all the unique references to the platform and divide the platform development costs by the total number of references. In Table 11.24 we assume that the platform has five components and that we developed seven projects. In total, the 7 projects referenced the platform 18 times.

Table 11.24 Standard Activity-Based Costing distribution of platform development costs

Activity	% of time spent	Assigned cost	Activity quantity	Cost-driver rate
Use of package 1	10 %	€3,105.00	5	€621.00
Use of package 2	20 %	€6,210.00	2	€3,105.00
Use of package 3	20 %	€6,210.00	3	€2,070.00
Use of package 4	20 %	€6,210.00	1	€6,210.00
Use of package 5	30 %	€9,315.00	7	€1,330.71
Total	100 %	€31,050.00	23	

This means that according to Activity-Based Costing, we would charge the two projects for each package they use according to the cost-driver rates in Table 11.24. If, for example, project 1 uses packages 1, 2, and 3, we would add 621.00 + 3,105.00 + 2,070.00 = €5,796.00 to the direct costs of the project.

In the case of Time-Driven Activity-Based Costing, we have to:

1. Determine the capacity of each resource: as capacity we wanted to specify the value of the platform to evaluate how much of the provided value was used. One possibility is to simply value every provided component equally (as suggested in [34]) or to value components differently, depending on their complexity, size, development effort, or some other criteria.

   We decided to differentiate between the different components using the relative development effort of each component as an indicator of its value. Therefore, the capacity of the platform was set to 100.
2. Determine the cost per unit of using that resource: if we assume that the total value of the platform is 100 %, the cost for using 1 % of the platform is the *value of the platform for one project* ÷ 100. As in the value for the platform, we

can set the entire development costs (€31,050), but this means that it is our target to charge every project with the full cost of the platform.

In our case, the organization estimated the market value of the components provided by the platform as €5,000. Therefore, 1 % was valued €50.

3. Estimate the standard unit times of consumption of resource capacity: for each component in the platform project, we determine the time that was needed to create it using our non-invasive measurement probes (see Table 11.25):

   a. package 1: 10 %, cost-driver rate: $10 \times 50.00 = 500.00$;
   b. package 2: 20 %, cost-driver rate: $20 \times 50.00 = 1{,}000.00$;
   c. package 3: 20 %, cost-driver rate: $20 \times 50.00 = 1{,}000.00$;
   d. package 4: 20 %, cost-driver rate: $20 \times 50.00 = 1{,}000.00$; and
   e. package 5: 30 %, cost-driver rate: $30 \times 50.00 = 1{,}500.00$.

Table 11.25 Results of Time-Driven Activity-Based Costing using non-invasive measurement probes

Activity	Quantity	Unit percentage	Total percentage used (in percentages of the platform effort)	Cost-driver rate	Total cost assigned
Use of package 1	5	10	50	€500.00	€2,500.00
Use of package 2	2	20	40	€1,000.00	€2,000.00
Use of package 3	3	20	60	€1,000.00	€3,000.00
Use of package 4	1	20	20	€1,000.00	€1,000.00
Use of package 5	7	30	210	€1,500.00	€10,500.00
Total used			380		€19,000.00
Total supplied			500		€31,050.00
Unused capacity			120		€12,050.00

11.5.3 The Role of the Experience Factory in Cost Accounting

The organizational unit that is responsible for the Experience Factory can use the data coming from cost accounting to support the team as described in Chap. 8 by monitoring the ongoing processes and provide feedback based on past projects. Some possible activities are [69]:

- keep track of the costs of using certain products, tools, and processes and package this experience so that it can be reused in future projects to forecast and plan;
- keep track of how much it costs to implement certain features and package this experience to support the team in make-or-buy decisions;
- keep track of the long-term costs of certain products (long-term license costs, hardware costs, maintenance costs, etc.) and package this experience so that it can be reused in future projects to forecast and plan;
- keep track of the costs of unused resources and package this experience so that it can be reused to optimize present and future production processes;
- support management to find a price for a produced product;
- calculate cost-driver rates; or
- develop cost models.

11.5.4 Results

This chapter reports how we used non-invasive measurement to determine the costs of developing software and how to organize the collected costs according to three cost accounting standards: Total Absorption Costing, Direct Costing, and Activity-Based Costing.

To validate the detected costs, we implemented a daily feedback e-mail that summarized the effort spent the day before. This approach wanted to imitate the idea of the Personal Software Process [52] in which developers have to collect their effort data to better understand their personal software process, i.e., how they are developing software, in what they are slow, in what fast, etc. The idea was that in this case, developers, seeing how they were spending their time, would get a better understanding of how productive they are.

In this case we used the daily mail mainly to understand where our system had to be modified. For example, there were situations in which users reported wrong effort because they forgot to log off and continued to work with another user. For such cases we implemented a feature in our user interface to reassign already collected effort to another user.

11.5.5 Discussion

We described a possible implementation of Total Absorption Costing, Direct Costing, and Activity-Based Costing in the context of source code costing. To achieve this, we used non-invasive measurement.

As we pointed out already above, Full Absorption Costing has to be used carefully: the attribution of indirect costs to cost-items is—from a logical point of view—incorrect. On the other hand, it is useful to get a basic idea of which price one could ask on the long term, to understand the impact of high investments in the

platform, to determine the right balance between the investments in the platform and the direct development in the project code, etc.

A misleading message that might come from Full Absorption Costing is that teams that use the platform get punished for that as they are being charged for using it. A team might be encouraged not to use the platform but to redevelop part of it, obtaining lower costs than the costs assigned though Full Absorption Costing, and to make the overall project look more profitable. This example shows again what we explained above: Full Absorption Costing was not developed as a decision support instrument.

From a Lean perspective, this project contributed to the maximization of value and the creation of knowledge as summarized in Table 11.26.

Table 11.26 Lean aspects addressed by the second described case

Goal	Strategy	Rationale
Maximization of value	Identification and elimination of waste	We inform the team which parts of the code cost much and which less. This additional information helps to decide whether the effort was spent in a useful way or not. This can help make a better decision in future projects
Creation of knowledge	Andon	The cost information that we collect helps in decision making. Since software is invisible and software can be rewritten without any visible effects, we do not see all the effort that is behind an ingenious piece of code. Knowing the past costs of a program should help us to decide whether our intern should try to rewrite

11.6 Case 3: Developing a Lean GQM Graph

In this case study we describe in detail how we developed, together with a software development team, a GQM$^+$Strategies model to specifically monitor the progress of the team towards Lean.

11.6.1 Theoretical Framework

The starting point for this case study was an organization that wanted to improve its efficiency and effectiveness and decided to do this by adapting their software development processes towards a Lean approach.

The performance measure of Lean practices is "leanness" [109], which Bayou and de Korvin describe as: "manufacturing leanness is a strategy to incur less input to better achieve the organization's goals through producing better output [11]."

The corresponding GQM$^+$Strategies element to measure leanness was defined as follows:

- **Organizational goal**: improve leanness within software development. Using the GQM$^+$Strategies goal template, this goal can be refined as follows:
  - **Object**: the software development process
  - **Focus**: leanness
  - **Magnitude**: reduction of time-to-market, improvement of quality
  - **Time frame**: continuously, monthly evaluation of the progress
  - **Organizational scope**: software development division
- **Assumptions**:
  - Development costs are mainly determined by labor costs.
  - Software development labor costs are mainly determined by interactions with the computer.
- **Constraints**:
  - To reduce measurement costs, we favor non-invasive measurement over manual measurement.
- **Strategy**: use the principles proposed by Mary and Tom Poppendieck [92] to guide the software development teams.

Following the strategy of this GQM$^+$Strategies element, the subgoals to achieve it are the principles stated by Mary and Tom Poppendieck [92]:

1. eliminate waste to become more efficient,
2. use autonomation and standardization to build quality into the process,
3. collect, maintain, and distribute know-how to exploit experience,
4. work just in time to minimize rework and to increase agility,
5. deliver fast to maximize learning,
6. involve the developer to learn from those who do the actual job: conceive and write the software, and
7. constantly improve to become more effective and to stay competitive.

These (sub)goals will be the starting point to develop the GQM graph. The concept map of the involved parts for this case is depicted in Fig. 11.38.

The role that the different concepts in this case have is the same as in the previous one.

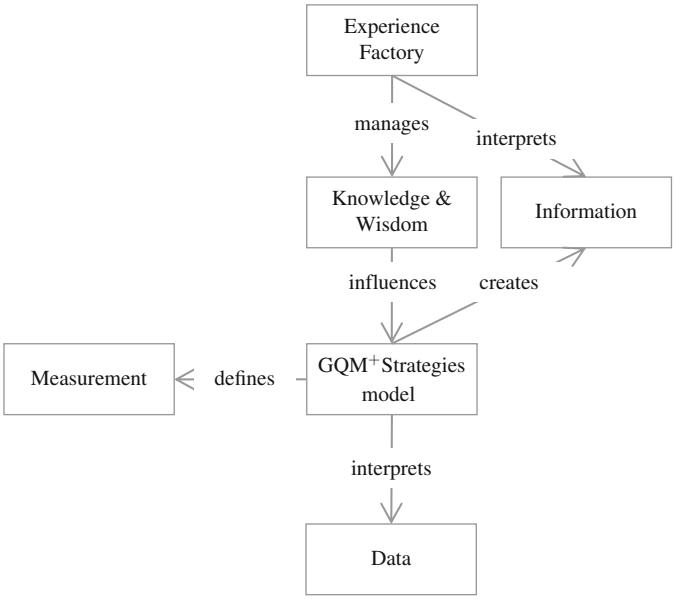


Fig. 11.38 Concept map for the third case study

11.6.2 The Study

Following the methodology illustrated in Sect. 7.4, the proposed sequence of steps was [88]:

1. identify your business goals,
2. identify what you want to know or learn,
3. identify your subgoals,
4. identify the entities and attributes related to your subgoals,
5. formalize your measurement goals,
6. identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals,
7. identify the data elements that you will collect to construct the indicators that help answer your questions,
8. define the measures to be used and make these definitions operational,
9. identify the actions that you will take to implement the measures, and
10. prepare a plan for implementing the measures.

We will explain how we went through this sequence of steps to develop a GQM graph to guide an organization towards becoming Lean.

11.6.2.1 Identify Your Business Goals

To measure "leanness," we use the seven principles of Lean software development by Mary and Tom Poppendieck (see Chap. 6):

1. eliminate waste to become more efficient,
2. use autonomation and standardization to build quality into the process,
3. collect, maintain, and distribute know-how to exploit experience,
4. work just in time to minimize rework and to increase agility,
5. deliver fast to maximize learning,
6. involve the developer to learn from those who do the actual job: conceive and write the software, and
7. constantly improve to become more effective and to stay competitive.

In fact these goals should be stated by the organization. In our case the organization was interested to explore possibilities to become more Lean, so we used the goals above as a starting point.

The business goals are not isolated; they are dependencies between them. For instance, the principle to "create knowledge" supports the principle "eliminate waste," since by applying knowledge, we do not repeat the same mistakes in the future; therefore, we will not waste time and effort [47].

The mind map [18] in Fig. 11.39 illustrates the concepts we want to study. We will refine this mind map as we refine the GQM model.

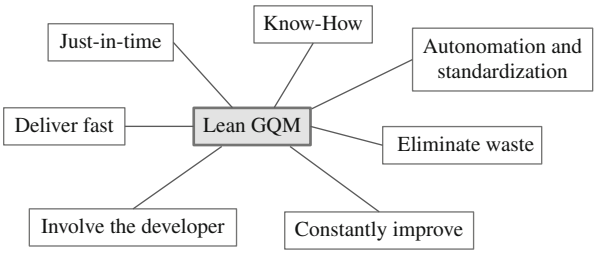


Fig. 11.39 Main branches of the mind map depicting the business goals

11.6.2.2 Identify What You Want to Know or Learn

The next step is to narrow down the specific aspects on which the organization is interested. As suggested by Park et al. [88], we specify the entities we are interested in together with the "questions that, if answered, would help you, in your role, plan and manage progress toward your goals."

We then grouped the identified entities into subgoals. To ensure that we cover as many aspects as possible relevant in an software engineering project, we use as entities the three categories proposed by Fenton and Pfleeger [33]:

- **product** (aspects about the outcome of the process: documentation, source code, etc.),
- **resource** (aspects about the resources used to produce the output: software, hardware, people, etc.), and
- **process** (the activities performed to obtain the desired output, using resources).

Tables 11.27, 11.28, 11.29, 11.30, 11.31, 11.32, and 11.33 list entities, questions, and trade-offs we identified for the seven business goals. All tables have a similar structure and consist of three elements:

- a question, which asks something about the current entity (product, resource, or process) that helps to find out something that helps to achieve the business goal;
- a rationale, which explains why we asked that question; and
- the trade-off: together with every question, we also point out the trade-off we identified. As we were formulating the questions and thinking about what we asked, we realized that there is always a trade-off to consider. For example, testing the software helps to identify defects, but if we only test, we have less time to do other things. Using the terminology presented in Chap. 5, we have to minimize the risk exposure of not testing enough and testing too much.

The here listed questions are by no means exhaustive. They are the result of our brainstorming with the company involved in this case study.

The mind map in Fig. 11.40 summarizes the identified entities and derived questions.

Table 11.27 Entities and questions derived from the business goal 1: eliminate waste to become more efficient

Entity	Question	
Product	1.1	How much modified source code is not committed? Rationale: work that has been modified but not committed is like an inventory of unfinished goods. If others base their work on the source code in the repository, it might be that their work is impacted by the uncommitted modifications. On the other hand, it is advantageous to create a system that is modular enough so that several programmers can work on different tasks without impacting each other all the time Trade-off: we have to avoid having too much rework and at the same time avoid spending too much time in developing a modular architecture
Product	1.2	How much committed source code is not tested? Rationale: the probability that untested source code contains defects is higher than in tested source code. Fixing defects later might generate more costs than testing now. On the other hand, testing requires time in which we cannot do other things Trade-off: we have to avoid testing too much and at the same time avoid testing too little. If we have too many defects, we lose our reputation and credibility
Product	1.3	How much committed source code is not documented? Rationale: undocumented source code increases the time new team members need to understand the intentions of the source code. On the other hand, documenting a statement like "a = 5" with "We assign 5 to the variable a" is useless and decreases the readability of the source code Trade-off: we have to avoid documenting too much and at the same time avoid documenting too less
Product	1.4	How much committed source code is not deployed? Rationale: similar to source code that is not committed, source code that is not deployed can contain defects that reveal itself only during deployment. Deploying late means to delay the moment in which we try out our work and understand if our assumptions were correct. On the other hand, deploying all the time might have drawbacks too: customers might be annoyed of getting updates all the time; updates on mobile devices can occur only when the device is connected to a power source, and so on. The team was using Kanban to organize their tasks. The defined Kanban limits block the team if some tasks are developed but not deployed, but there were tasks where the team needed input by the client or other organizations. In such case the team accepted that the Kanban limit was not respected. Nevertheless, it was important to regularly investigate how to complete the open tasks Trade-off: we have to avoid deploying too often and at the same time avoid deploying too rarely

(continued)

Table 11.27 (continued)

Entity	Question	
Product	1.5	How much do our customers value our products? Rationale: we should constantly invest time to understand the value we are providing with our products. It might be that we provide a product that customers do not need or that we do not provide something they need. The more we find out about how customers use our products and which problems they solve with them, the more we can improve the value we provide. On the other hand, we do not want to annoy customers too much or spy on them violating their privacy Trade-off: we have to avoid getting too much and at the same time avoid getting too less feedback from customers
Resource	1.6	Which skills do we need to acquire or improve? Rationale: having the "right" skills allows to offer valuable products. We should dedicate time to improve our skills and to learn new skills. On the other hand, we have to complete also the day-to-day work, since those activities provide value to our customers Trade-off: we have to avoid not improving and at the same time not working on the routine activities[a]
Process	1.7	What is the value of the activities we perform? Rationale: understanding how much each step contributes to the creation of value helps to identify unnecessary steps. On the other hand, studying the contribution of every little movement is not feasible Trade-off: we have to avoid having an inefficient process because we do not know what is happening and at the same time avoid spending too much time analyzing it
Process	1.8	What are the costs of the activities we perform? Rationale: understanding how much each activity costs allows us to compare these costs with the value it provides. If something costs us $1,000 but for the customer is worth $100, we should look for alternative solutions (e.g., buy it from somebody else and adapt it to our needs). On the other hand, measuring costs in too high detail can be costly, too Trade-off: we have to avoid wasting money for activities the customer does not value and at the same time avoid spending too much effort analyzing the cost-effectiveness
Process	1.9	Which mechanisms do we use to get frequent feedback from customers? (rationale and trade-off as in question 1.5)
Process	1.10	How much time do we spend in reimplementing similar features again and again? Rationale: instead of solving the same problem again and again, we can increase our effectiveness by increasing reuse. On the other hand, if we create a process that reuses everything from past projects, we risk to become inflexible and not able to solve new requirements in innovative and creative ways Trade-off: we have to avoid being inefficient because of rework and at the same time avoid losing our agility

[a] This is the content of the case study discussed in Sect. 11.4

Table 11.28 Entities and questions derived from the business goal 2: use autonomation and standardization to build quality into the process

Entity	Question	
Product	2.1	How high is the coverage of the automated unit test cases? Rationale: only automated test cases can be executed frequently and automatically. Only the source code that is covered by such test cases is tested frequently. As the complexity of a project increases, the probability of doing mistakes increases and automated test cases can help. On the other hand, it can be very difficult to automate a test case. In some cases we need to change the source code so that another program (the test case) can execute it as needed. In some other cases, quality is not needed and the effort of testing the source code is unnecessary Trade-off: we have to avoid having too low quality and at the same time avoid having too high quality
Product	2.2	How high is the coverage of the automated integration test cases? (rationale and trade-off as in question 2.1)
Product	2.3	How much of our build process is automated? Rationale: an automated build process allows developers to frequently integrate their changes with the rest of the source code base and to understand how their changes impact the whole system. Moreover, it can be implemented automatically. On the other hand, setting up and maintaining an automated build system requires discipline by all team members and a constant effort Trade-off: we have to avoid having too low quality because of so-called "bigbang" integrations at the end of an iteration and at the same time avoid spending too much time in the automated build system
Product	2.4	How often does it happen that problems that have been fixed once appear another time? Rationale: recurring problems are an indicator of too low autonomation. If a problem that might happen again is solved, it is important to cover it with a test case. If it then reappears, we can solve it right away without impacting others. Again, as in the questions before, it might be impossible or very time consuming to develop a tool that monitors some specific system Trade-off: we have to avoid loosing our reputation and trust from the client because of recurring problems and at the same time avoid spending too much time implementing autonomation
Product	2.5	What are frequent causes of defects? Rationale: if we know frequent causes of defects, we can pay particular attention to such kinds of defects or implement an autonomation solution for it Trade-off: we have to avoid doing the same mistakes over and over again because we do not study typical problems and at the same time avoid spending too much time studying our mistakes

(continued)

Table 11.28 (continued)

Entity	Question	
Product	2.6	How much source code do we have to write from scratch for every new project? Rationale: if projects have repetitive aspects, it might be worthy to develop a generic, reusable component. The drawback is that we might spend a lot of time "gold plating[a]" a generic solution. Moreover, if we then expect from everybody to reuse our predefined components, we indirectly stop developers from developing innovative ideas like trying out a new way to solve an old problem Trade-off: we have to avoid wasting effort because of rework and low quality and at the same time avoid limiting creativity
Product	2.7	How much documentation do we have to write from scratch for every new project? (rationale and trade-off as in question 2.6)
Resource	2.8	Is somebody responsible and competent and does somebody have the necessary power to implement autonomation? Rationale: if nobody is responsible, or if the responsible does not have the necessary skills or power to implement autonomation, it will not happen [95]. On the other hand, if somebody is busy improving the process, he cannot do other things Trade-off: we have to avoid having defects because we are not implementing autonomation and at the same time avoid implementing it more than we need it
Process	2.9	Do we have standardized ways to perform activities that happen frequently? Rationale: defined processes are a way to embed knowledge into manual activities. On the other hand, we cannot automate aspects that require creativity Trade-off: we have to avoid wasting effort because we do not reuse knowledge and at the same time avoid limiting creativity

[a]Gold plating means to work on something more than necessary.

Table 11.29 Entities and questions derived from the business goal 3: collect, maintain, and distribute "know-how" to exploit experience

Entity	Question	
Product	3.1	What are our strengths, weaknesses, opportunities, and threats? Rationale: knowing our strengths, weaknesses, opportunities, and threats reveals what we have to know and learn to improve and what we can exploit or what we have to avoid. On the other hand, implementing a new strategy, learning a new skill, etc., takes time. We have to carefully select the options with the best probabilities of success, while continuing to work on the projects that represent our strength. Trade-off: we have to avoid not improving and getting bankrupt and at the same time only improving and getting bankrupt
Product	3.2	Which technologies do we need to develop to meet the needs of the market better? (rationale and trade-off as in question 3.1)
Resource	3.3	Is somebody responsible and competent and does somebody have the necessary power to collect, manage, and distribute knowledge? (rationale and trade-off as in question 2.8)

(continued)

Table 11.29 (continued)

Entity	Question	
Resource	3.4	Is everybody aware of what is happening in the organization, team, or project? Rationale: we need to inform everybody about the strategy of the organization about everything that might impact the work of everybody. On the other hand, there is the risk of information overload Trade-off: we have to avoid not being informed at all and at the same time avoid being informed too much
Resource	3.5	Which skills do we need to get to meet the needs of the market better? (rationale and trade-off as in question 3.1)
Process	3.6	Do we have a process in place that supports the team in collecting knowledge? (rationale and trade-off as in question 3.1)
Process	3.7	Do we have a process in place that uses the accumulated knowledge to standardize it? (rationale and trade-off as in question 3.1)

Table 11.30 Entities and questions derived from the business goal 4: work "just in time" to minimize rework and to increase agility

Entity	Question	
Product	4.1	How much source code do we modify because of changed requirements? Rationale: if we study the impact of changed requirements, we get an idea of how much we could save implementing "just-in-time". On the other hand, doing everything "just in time" stops us from optimizing frequent activities Trade-off: we have to avoid working only improvising and at the same time avoid being inflexible
Product	4.2	What type of source code do we modify because of changed requirements? (rationale and trade-off as in question 4.1)
Product	4.3	What type of requirements require the changes in the source code? (rationale and trade-off as in question 4.1)
Resource	4.4	Which of our technologies stop us from doing something "just in time"? Rationale: some technologies were conceived with a waterfall process in mind and we spend a lot of time tweaking and tricking the technology to work in an Agile way. We should dedicate time to find out how to switch to another technology or how to modify the existing technology so that the restrictions are removed. On the other hand, if we change our tools all the time, there is no time to optimize them Trade-off: we have to avoid changing our instruments too often and at the same time avoid being inflexible
Process	4.5	Which of our methods and working habits stop us from doing something "just in time"? (rationale and trade-off as in question 4.4)
Process	4.6	How much time do we spend because of changed requirements? (rationale and trade-off as in question 4.1)
Process	4.7	What type of changes require the longest modifications in terms of time? (rationale and trade-off as in question 4.1)

Table 11.31 Entities and questions derived from the business goal 5: deliver fast to maximize learning

Entity	Question	
Resource	5.1	How much are development activities automated? Rationale: automated development steps can be adapted, extended, modified, etc. over time using the experience of the team. For example, if we find out that some specific problem can arise only on some special hardware, we can develop a test case that is executed on every nightly build. This allows us to deliver reliable updates of the software in shorter time. Manual steps have to be learned, and we lose it as soon as the person is not available. On the other hand, we cannot automate aspects that require creativity Trade-off: we have to avoid wasting effort because of manual work and at the same time avoid limiting creativity
Process	5.2	How much are deployment (installation and setup) activities automated? (rationale and trade-off as in question 2.3)
Process	5.3	How long does it take on average to solve a defect? Rationale: we have to study how fast we are to understand what is blocking us. On the other hand, we should not waste too much time studying ourselves; getting a rough idea is enough Trade-off: we have to avoid being too slow and at the same time avoid spending too much time studying why we are slow
Process	5.4	How long does it take on average to implement a new feature? (rationale and trade-off as in question 5.3)
Process	5.5	How long does it take on average to complete an iteration? (rationale and trade-off as in question 5.3)

Table 11.32 Entities and questions derived from the business goal 6: involve the developer to learn from those who do the actual job: conceive and write the software

Entity	Question	
Product	6.1	Which features would the developers suggest to implement? Rationale: as pointed out in Chap. 3, clients rarely know the technological possibilities developers do. We should listen to what developers propose to make the lives of the clients easier. On the other hand, we all prefer to work on something new than on something we know already inside out. There is the risk that we spend our time only with the new, interesting challenges, instead of fixing the older things Trade-off: we have to avoid "gold plating" the software and at the same time motivate everybody to be innovative
Product	6.2	Which features would the developers suggest to remove? (rationale and trade-off as in question 6.1)

(continued)

Table 11.32 (continued)

Entity	Question	
Product	6.3	Which are the most disliked features by developers? Rationale: developers might have not like to work on some piece of source code or project for different reasons, for example, if the source code is of poor quality. In such case, making a mistake is easy and the developer might be afraid to be blamed for things he did not intend to do. Such behavior should be taken seriously and a discussion should be started within the team why somebody does not like certain types of work and how to alleviate this. Maybe some source code can be refactored, activities automated, etc. On the other hand, some tasks **are** boring, but are required Trade-off: we have to avoid discussing all the time about every little issue and at the same time avoid not to discuss at all
Resource	6.4	Do we have a fruitful organizational culture that promotes innovation? (rationale and trade-off as in question 6.3)
Resource	6.5	Is the team responsible and competent and does it have the necessary power to contribute to the outcome of the work? (rationale and trade-off as in question 2.8)
Process	6.6	What are the most disliked activities by developers? (rationale and trade-off as in question 6.3)

Table 11.33 Entities and questions derived from the business goal 7: constantly improve to become more effective and to stay competitive

Entity	Question	
Product	7.1	How does the required effort for typical activities change over time? Rationale: as we collect more and more experience, we should become more efficient and effective. If we do not, this might be an indicator that we are not using some opportunities to improve. On the other hand, there might be other reasons, e.g., that customers are more exigent than before, that the competition is harder, etc. Trade-off: we have to avoid working in a sloppy way and at the same time avoid being inefficient because we want to be perfect[a]
Product	7.2	How often do we refactor source code? Rationale: refactoring makes it easier to implement new features or to modify existing ones. On the other hand, refactoring takes time, might introduce new defects, and requires everybody in the team to be informed about the refactoring. Sometimes we have to work with a system for some time to understand which is the best refactoring Trade-off: we have to avoid refactoring without a reason and at the same time avoid not to refactor at all

(continued)

Table 11.33 (continued)

Entity	Question	
Resource	7.3	Do we optimize resource utilization for the whole process instead of local? Rationale: local optimizations are often easier to identify and to carry out than global ones. Moreover, the complexity and risk we face tackling global optimizations are usually higher than for local ones. Additionally, there might be other aspects to consider: for example, if we drastically change the user interface to diminish the effort of the average user, we might lower the productivity of everybody at the beginning. If we change the entire API, we would need to rewrite all the systems relying on it. On the other hand, global optimizations are often able to improve the overall quality much more than local ones Trade-off: we have to avoid wasting resources optimizing locally and at the same time avoid disrupting the entire product too often
Process	7.4	What stops us from improving? (rationale and trade-off as in question 4.4)

[a]Voltaire coined the phrase: "Perfect is the enemy of good"

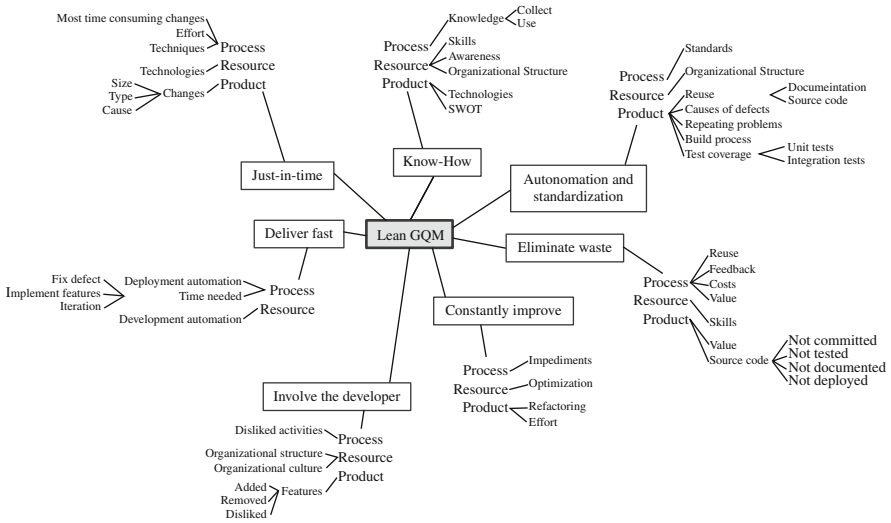


Fig. 11.40 Mind map depicting the business goals, entities, and derived questions

11.6.2.3 Identify Your Subgoals

As next step we group the aspects identified in the previous step into subgoals that state goals for activities that support the business goals.

Again, the way how we grouped the questions to subgoals reflect the goals of the organization. These subgoals are not all subgoals that can be derived from the stated questions they are those that the team evaluated as important in their context (Table 11.34).

Table 11.34 Subgoals derived from the questions in Sect. 11.6.2.2

Subgoal		Questions
1.	Keep the amount of unfinished work to a minimum Rationale: questions 1.1–1.4 deal with source code or documentation that is under development and not yet integrated with the rest of the project. This subgoal states this objective for all sorts of unfinished work. We decided to ignore measuring—in our case—too detailed aspects, e.g., how much source code was not committed, and to focus on the progress of the tasks along the Kanban board of the team, which was going from "Backlog" to "Live!" (see Fig. 11.41)	1.1, 1.2, 1.3, and 1.4
2.	Understand the value of our products Rationale: question 1.5 deals with the value of what we produce from the point of view of the client. Additionally, questions 6.1–6.3 look at the value from the point of view of developers. Questions 4.1, 4.2, 4.3, and 7.2 ask about the modifiability of our product	1.5, 4.1, 6.1, 6.2, 6.3, and 7.2
3.	Understand which skills we have to learn Rationale: questions 1.6, 1.7, 3.1, 3.5, and 7.4 all deal with the value of what we do and the skills we need. Question 6.5 asks in general what kind of abilities we have or miss	1.6, 1.7, 3.1, 3.5, 6.5, and 7.4
4.	Understand which technologies we need to develop or buy Rationale: questions 3.2, 4.4, and 6.1 ask about technologies that we need to acquire or produce	3.2, 4.4, and 6.1
5.	Study the value of past activities to learn from mistakes Rationale: questions 1.7, 1.9, 3.1, 4.5, 6.4, and 6.5 look at the value of the activities we perform and our ability to improve it. Question 4.5 looks at the activities that stop us from achieving "just-in-time"	1.7, 1.9, 3.1, 4.5, 6.4, 6.5
6.	Understand the reasons behind development costs Rationale: questions 1.8 and 7.3 ask about the costs we have and our ability to decrease them over time. Questions 1.10, 4.6, 4.7, 5.3, 5.5, and 7.1 ask about the time we need to produce. Question 6.6 wants to identify activities that should be automated	1.8, 1.10, 4.6, 4.7, 5.3, 5.4, 5.5, 6.6, 7.1, 7.3
7.	Understand which experience we should collect and package Rationale: questions 1.10, 2.4, and 2.5 deal with repeating problems and the time we waste dealing with them. Questions 2.6, 2.7, and 2.9 deal with the reuse of old solutions to new projects. Questions 3.3, 3.6, and 3.7 ask if the organization has defined processes and provided resources to collect and reuse knowledge. Question 7.1 wants to analyze if the collected knowledge is improving the processes over time	1.10, 2.4, 2.5, 2.6, 2.7, 2.9, 3.3, 3.6, 3.7, 7.1
8.	Make sure everybody has all the information needed to solve a specific task Rationale: question 3.4 asks if everybody has the right information at the right moment to solve a task. Questions 6.3 and 6.6 want to find out if developers need more information to perform their tasks easier	3.4, 6.3, 6.6

(continued)

Table 11.34 (continued)

Subgoal		Questions
9.	Embed quality into the process using autonomation Rationale: automation is part of autonomation—autonomation requires that some mechanisms checks the quality of the output automatically and informs everybody if there is a problem. Questions 2.1, 2.2, 2.3, 5.1, and 5.2 ask about automation. Question 2.8 asks about responsible team members for autonomation	2.1, 2.2, 2.3, 2.8, 5.1, 5.2
10.	Embed quality into the process using standardization Rationale: questions 2.9 and 3.7 ask about standardization and a process that supports the creation of standards	2.9, 3.7

11.6.2.4 Identify the Entities and Attributes Related to Your Subgoals

Starting from the subgoals obtained from the previous step, in this step we focus on how to reach them identifying the entities and attributes of those entities that influence their outcome.

As we were defining entities and attributes, we noticed that we needed a clear definition of what we mean by an entity. Therefore, in Table 11.35 we list all entities and how we defined them.

Table 11.35 Entities related to the subgoals of Sect. 11.6.2.3

Entity	Definition
Task	The smallest unit of work subject to management accountability. [101]. A collection of work tasks spanning a fixed duration within the schedule of a software project is called activity [101]. We further distinguish manual tasks, which require human work from automatic tasks, which can be executed by a machine
Automated verification system	Any software tool that automates part or all of the verification process [103]. An example of such a system are JUnit [62] test cases
User documentation	Documentation describing the way in which a system or component is to be used to obtain desired results [103].
Software feature	A distinguishing characteristic of a software item (e.g., performance, portability, or functionality) [103]
Guideline	An official recommendation or advice that indicates policies, standards, or procedures for how something should be accomplished [94]
Skill	The ability to use a technology or methodology to support the software development process
Source code	Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator [103]

(continued)

Table 11.35 (continued)

Entity	Definition
Resource	Skilled human resources, equipment, services, supplies, commodities, material, budgets, or funds [94]
Component	A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [104]. We use the term "component" to underline the intention of the team to reuse a given piece of software in future projects

The next table, Table 11.36, lists entities and attributes that we think help us to fulfill the subgoals listed in Table 11.34. To do this, we also looked at the questions from which the subgoals originated.

Table 11.36 Entities and attributes related to the subgoals of Sect. 11.6.2.3

Entity	Attribute	Subgoal
Task	Phase Rationale: keeping track of the current phase (e.g., Backlog, Selected, Develop, Deploy, Live!) of each task helps the team to understand when too much work is unfinished or at which phase too many tasks are stuck	1.
Software feature	Value for the client Rationale: measures the importance of a feature for the client	2., 3.
Software feature	Value for the developer Rationale: measures how important a feature is for the developer	2., 3.
Software feature	Skills that are needed to develop a feature Rationale: measures the skills needed to develop a feature. Together with the value of a feature, we can infer the skills we need to learn or improve	3.
Software feature	Technologies that are needed to develop a feature Rationale: measures the technologies needed to develop a feature. Together with the value of a feature, we can infer the technologies we need to build or buy	4.
Task	Generated value Rationale: measures how important a task is for the client	5., 7.
Task	Generated waste Rationale: measures how much waste was generated during the execution of this task	5., 7.
Task	Occurrence Rationale: measuring how often, in which sequence, at what time, etc. we need to perform a given task helps us to understand its importance	5.

(continued)

Table 11.36 (continued)

Entity	Attribute	Subgoal
Task	Effect Rationale: we need to keep track of the effects of tasks (the generated source code, documents, etc.) to optimize the process, e.g., to find a cheaper or better solution	6.
Task	Costs Rationale: the cost of a task has to match the value it has for the customer. Knowing the costs of tasks helps to understand a) which skills or technologies should be improved first to improve efficiency, and b) it helps to decide which tasks should be automated or supported by reusable solutions	3., 4., 5., 6., 7.
Task	Lifetime Rationale: if problems or tasks are not solved for a long time, there might be several reasons: the team is not aware of the task, the task is difficult, the task is not challenging, etc. We have to identify such tasks and learn how to cope with them	8.
Automated verification system	Coverage Rationale: we need to understand how much of the source code is tested automatically. We can then decide for which other source code we set up automatic testing or for which source code it would be too costly	9.
Guidelines	Usage Rationale: we need to understand for which activities we use guidelines during the development process. We can then decide for which activities we should develop guidelines, or for which activities they do not make sense	10.

The here identified entities and attributes are not complete. There are a variety of entities and attributes we could measure to learn how to achieve the goals stated in Table 11.34. To avoid wasting effort, the organization chooses the entities and attributes that it finds the most insightful to achieve their measurement goals. In our case, we choose the entities and attributes listed in Table 11.36, which we formalize in the next section.

11.6.2.5 Formalize Your Measurement Goals

We now use the template described in Chap. 7 to describe measurement goals that help us in the achievement of our subgoals and ultimately our business goals. Table 11.37 lists all formal measurement goals we stated.

Table 11.37 Formalized goals according to the GQM goal template (see Chap. 7)[a]

	Purpose			Perspective		Environment		
		For the purpose of③			From the point of view of a⑦			
No.	Analyze①	Characterization	Evaluation	With respect to⑤	Client	developer⑧	in the context of a software development team within a custom software development company⑨	Subgoal
1.	Software product②		×	The amount and causes of unfinished work⑥		×	×	1.
2.	Software product		×	The provided value	×		×	2.
3.	Software product	×		The potential provided value		×	×	2.
4.	Software features		×	The necessary skills		×	×	3.
5.	Software features		×	The necessary technologies		×	×	4.
6.	Tasks		×	The provided value	×		×	5.
7.	Tasks		×	The generated waste		×	×	5.
8.	Production costs	×		Tasks causing the costs		×	×	6.
9.	Tasks	×		The experience we can gain		×	×	7.
10.	Tasks	×		Visibility of the tasks		×	×	8.
11.	Tasks	×		The degree of autonomation		×	×	9.
12.	Tasks	×		The degree of standardization		×	×	10.

[a]This table lists all formal goal elements in a tabular form. To read the goal in form of a complete sentence, read it in the sequence depicted by the numbers ① to ⑨

11.6.2.6 Identify Quantifiable Questions and the Related Indicators That You Will Use to Help You Achieve Your Measurement Goals

For each formal measurement goal in Table 11.37 we now state quantifiable questions. As we stated in Chap. 7, we distinguish three types of questions:

1. questions that characterize the object of study with respect to the overall goal,
2. questions that characterize relevant attributes of the object of study with respect to the focus, and
3. questions that evaluate relevant characteristics of the object of study with respect to the focus.

We stated the questions in Table 11.38. We based the questions on the identified entities and attributes in Sect. 11.6.2.4.

Table 11.38 Quantifiable questions for the measurement goals in Table 11.37

No.	Goal(s)	Question	Question type
1.	1.	How many tasks are in which phase?	3.
2.	1.	How long are the tasks in their current phase?	3.
3.	1.	What are the most frequent reasons that tasks stay in an intermediate phase[a] for more than 1 iteration?	2.
4.	2., 3.	How important are the existing software features for the client?	1.
5.	2.	Which software features should we implement in the future?	1.
6.	3.	Which software features should we implement in the future?	1.
7.	4.	Which activities does a software developer perform during a project?	2.
8.	4.	How frequent are the different activity types?	3.
9.	4.	In which sequence or combination are the different activity types used?	3.
10.	5.	Which technologies does a software developer use during a project?	2.
11.	5.	How often are the different technologies used?	3.
12.	5.	In which sequence or combination are the different technologies used?	3.
15.	6.	How important is each task for the client?	3.
16.	7.	How much effort, in each software module, was spent on rework?	2.
17.	7., 8.	Which tasks generated the rework?	2.
18.	6., 8.	Which code is written because of each task?	3.
19.	3., 4., 5., 6., 7., 8.	How much does the implementation of a task cost?	3.
20.	8., 9., 10.	How long does it take for each task to finish?	3.
21.	11.	How much is autonomation used to ensure the quality of our products?	3.
22.	12.	How much is standardization used to ensure the quality of our products?	3.

[a]By "intermediate" we mean a phase that is not the first and not the last one

Now that we stated the questions, we looked for indicators that help us to answer those questions. As we pointed out in Chap. 9, an "indicator" is something that indicates and that points to the data. It helps us to infer the real data behind the indicator [8]. Park et al. [88] call indicators "a picture or display of the kind one would like to have to help answer the question." Moreover, they state that according to their experience "sketches of such pictures and displays help significantly in

identifying and defining appropriate measures." We followed this approach in this chapter, too.

To answer questions 1, 2, 3, and 20, we instrumented a software-based Kanban board that the team had developed internally (see Fig. 11.41).

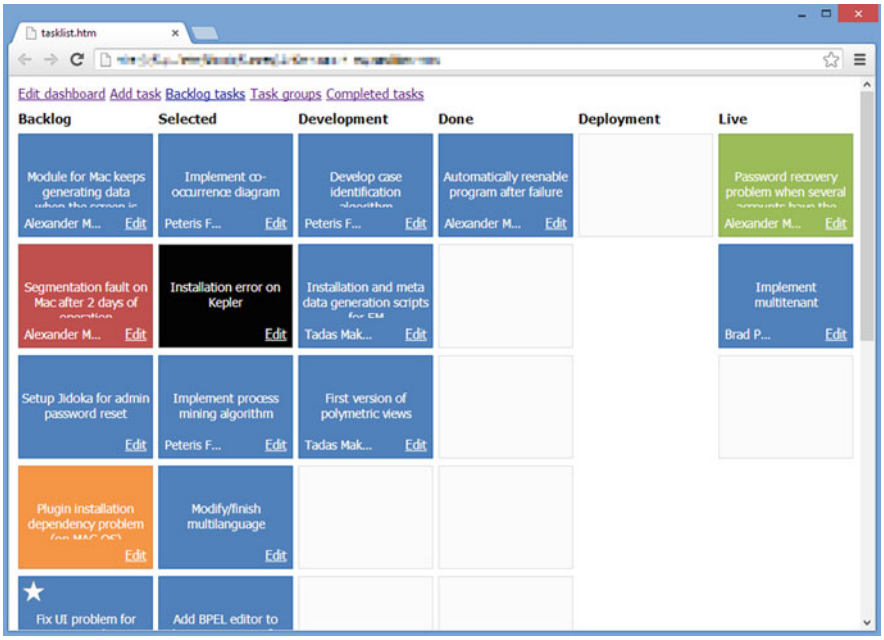


Fig. 11.41 An internally developed Kanban board

The data about the different tasks were stored in a company-internal database. This data could be extracted regularly as described in the second case study and depicted in Table 11.21.

Question 4 aimed to classify the importance of the features of the application. As we looked for indicators to measure "importance," we noticed that, in our case, the term "importance" was ambiguous. Therefore, we estimated it based on two factors:

- how often the feature was used [76] and
- the worst-case loss if the feature was not working correctly.

Both factors were estimated for each feature to obtain estimates as the examples in Table 11.39.

The final importance of a feature was then evaluated using the matrix depicted in Fig. 11.42.

Validating our estimation of the feature usage was correct; the shipped product contained a usage monitor that—with the permission of the user—collected anony-

Table 11.39 Example importance estimates

Feature	Usage	Worst-case loss	Code module
Manage invoices[a]	Medium	High	it.product.invoicing.invoice
Chat[b]	High	High	it.product.chat.*
Mail merge function to print the New Year's greetings[c]	Low	High	it.product.print.merge.*
Export function to transfer the invoice data to the accounting system[d]	Medium	High	it.product.export.*

[a]The company prints about ten invoices per day. Therefore, if the invoicing system does not work, this has not a big impact. If absolutely necessary, invoices can be written by hand and put into the system later. On the other hand, if an invoice is sent with wrong data in it (e.g., with the wrong amounts, the wrong numbering, etc.), this could result in a fine up to €2,065 per wrong invoice [84]

[b]The application has an internal chat system with which employees can use to communicate with each other. For security reasons the single computers are not connected to the Internet; therefore, the company is using this internal system. Even though this system is frequently used, it is not critical

[c]The application has a special module to print letters for the New Year's greetings. If this system does not work and cannot be repaired within weeks, this is not a problem. Nevertheless, if it is not possible to print the letters before the December 29, all envelopes have to be addressed by hand and this will cause considerable effort for the client

[d]If the organization does not register on time all the produced invoices as required by the law, this could result in a fine of 100–200 % of the taxes of the invoices that were not registered correctly [84]

mous usage data. Using this data it was possible to measure how much different features of the application were used by the customer.

Table 11.40 shows an example of the data that we collected using the usage monitor. This example shows the data of a user that clicks on the menu on "Customer," "Search," and then he enters some search criteria, clicks on "OK," and in the window that opens afterwards, he clicks on "Activities" and "Print."

Above we defined a software feature as a "distinguishing characteristic of a software item (for example, performance, portability, or functionality) [103]." Using the type of indicator described here, we limited ourselves to measure only functional aspects that the user could reach through the user interface.

Questions 5 and 6 aimed to collect proposals on how to extend the product in the future and to improve its value. The assumption was that customers are more interested in functionality that solves their current problems, while developers, aware of the technical possibilities, are able to propose new, alternative ways to solve the problems of the customer. Therefore, we allowed also developers to propose new features or extensions to existing features in the form of tasks on the Kanban board. New proposals—as the proposals of the client—were added to the "Backlog"

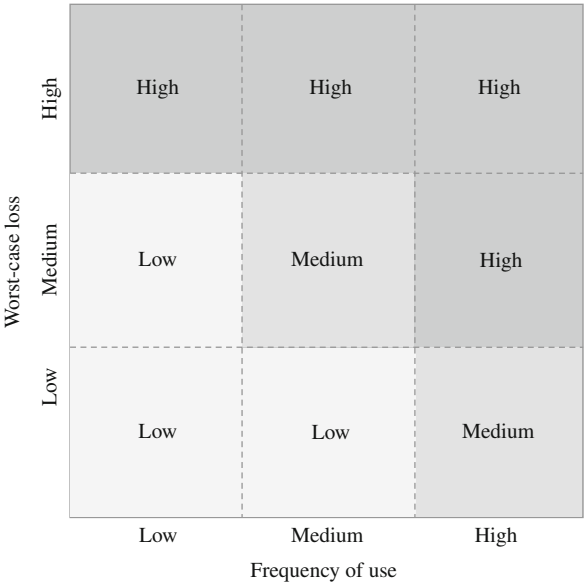


Fig. 11.42 The evaluation of the importance of a feature based on its usage and worst-case loss

Table 11.40 Example usage data of a product

Timestamp	Window class	Item type	Item caption
08.01.2013 09:36:10	it.program.Main	Menu	"Customer"
08.01.2013 09:36:15	it.program.Main	Menu	"Search"
08.01.2013 09:38:04	it.program.customer.Search	Textbox	"Name"
08.01.2013 09:41:04	it.program.customer.Search	Button	"OK"
08.01.2013 09:42:56	it.program.customer.Main	Button	"Activities"
08.01.2013 09:45:03	it.program.customer.Main	Button	"Print"

column of the Kanban board and then moved to "Selected" by the product owner if he decided that this task should be carried out.

To answer questions 7–14, we use the activity data obtained through non-invasive measurement. An example of the data that is collected is shown in Table 11.41.

To answer questions 10–12, we needed to find out the technologies that the team was using. We used the applications obtained through non-invasive measurement as in Table 11.41. In this case, we limited ourselves to track only the different software technologies the team was using instead of every possible technology.

The indicators for the tasks 15–18 will be described below, first question 19: to determine the cost of a task, we use non-invasive cost accounting as described in the previous case study (see Table 11.22).

Table 11.41 Example activity data obtained through non-invasive measurement

Machine	Application	Item	Duration	Activity
1	Eclipse	at.company.project8.Action.run()	10:00	Coding
2	Eclipse	at.company.project8.ActionTest.test1()	12:00	Coding
2	Eclipse	at.company.project8.ActionTest.test2()	7:30	Coding
1	Google Chrome	http://www.codeproject.com	2:00	Browsing
3	Google Chrome	http://stackoverflow.org	7:00	Browsing
4	Microsoft Excel	c:\projects\Comparison.xlsx	9:00	Documentation
4	Notepad	c:\projects\project8\summary.txt	3:00	Unknown
2	Microsoft Word	c:\projects\project8\manual.docx	15:00	Documentation

Question 15 wants to evaluate the importance of each task from the point of view of the client. To answer these questions, we combine two answers: the answer of question 4 and the manual classification of the importance of software modules, and question 19. For question 4 we defined the importance (based on feature usage and the worst-case loss) for each module. As in the previous case study, the team organized different modules in different namespaces. This made it possible to translate modules that the customer could understand (e.g., invoicing, chat, or New Year's greetings) into module names that corresponded to parts of the source code ("it.product.invoicing," "it.product.chat," and "it.product.mailmerge"). The non-invasive measurement probes described in the previous case study, allowed it in this one to measure in which package the developers were working during certain tasks, and to derive the importance of the task based on the importance of the code they were working on.

Our first idea to answer questions 16 and 17 was to look if certain tasks were going backwards the Kanban board, i.e., went to "Live!" but then went back to "Development" because of rework. Unfortunately, whenever some module had to be changed because of a changed requirement, the team members were not picking an old task and moving it back to "Development," they were creating a new task. Therefore, we concentrated at the source code. We answered questions 16 and 17 looking at the data collected by non-invasive cost (see Table 11.17). We were looking for parts in the code that were edited after a break of three iterations.

Question 18 identifies traceability links between the task and the written source code. This link can be also determined by the non-invasive cost accounting module.

Questions 21 and 22 evaluate the degree of autonomation and standardization. The first question can be answered automatically, calculating the coverage of the source code that is covered by the used automated verification systems. Question 22,

the coverage of the process by guidelines, has to be evaluated by hand. This estimation was based on the existing process guidelines and the estimated frequency they were used (Table 11.38).

Table 11.42 Indicators used to answer the questions of Table 11.38

No.	Indicator	Questions																				
		1	2	3	4	5	6	7	8	9	10	11	12	15	16	17	18	19	20	21	22
1.	Kanban board[a]	×	×	×		×	×												×		
2.	Feature usage[b] and worst-case loss estimation by customer[c]				×									×	×						
3.	Activity log[d]							×	×	×	×	×	×								
4.	Non-invasive cost accounting[e]													×	×	×	×	×			
5.	Coverage evaluation based on automated verification systems																			×	
6.	Manual coverage evaluation based on personal evaluation by the developers																				×

[a]See Fig. 11.41, Tables 11.21 and 11.43
[b]See Table 11.40
[c]See Table 11.39
[d]See Table 11.41
[e]See Table 11.17

The selected indicators have to be validated, i.e., we have to make sure that they represent what we want to measure. To ensure their validity, the team constantly reviews the outcomes of the measurement and questions the conclusions we make from the indicators.

As we were designing and choosing the indicators, we extended the information we needed to store in the company-internal Kanban system. In summary, we had the following data stored for each task:

- the short description of the task,
- the detailed description of the task,
- the date when the task was created,
- the author,
- the responsible,
- waits for, and
- the deadline

Table 11.43 shows some example tasks.

Table 11.43 Example tasks

Short description	Detailed description	Created	Author	Responsible	Waits for	Deadline
...	...	9.1.2014	Andrea	Joe	Joe	1.5.2014
...	...	11.1.2014	Andrea	Joe	Customer	1.7.2014
...	...	12.1.2014	Andrea	Joe	Customer	1.7.2014
...	...	15.1.2014	Andrea	Joe	Andrea	31.1.2014

Moreover, for each task a separate table was created to log all status changes together with the date when the status changed (to obtain the data as in Table 11.21 of the previous case study). The current status of the task was calculated using the data of the status changes of the tasks. Through non-invasive cost accounting, we were able to extract the source code that was edited (as in Table 11.22) because of a task, and therefore, we could obtain its severity comparing the edited source code with the defined importance of the different modules as in Table 11.39.

11.6.2.7 Identify the Data Elements That You Will Collect to Construct the Indicators That Help Answer Your Questions

In this step we now define in more detail which data exactly we are going to collect. We defined "a picture or display of the kind one would like to have to help answer the question" for every question; in this section we will explain which data we need to extract from the picture or display, i.e., the indicator.

Table 11.44 Measures extracted from the indicators of Table 11.42

No.	Measure	Indicators						Question(s)
		1	2	3	4	5	6	
1.	Number of phases	×						1.
2.	Tasks per phase	×						1.
3.	Task age	×						2.
4.	Code that was affected by the task				×			3.
5.	User that is working on the task	×						3.
6.	Feature usage estimation		×					4.
7.	Worst-case loss estimation		×					4.

(continued)

Table 11.44 (continued)

No.	Measure	Indicators						Question(s)
		1	2	3	4	5	6	
8.	Proposed extensions to the system by customers and developers	×						5., 6.
9.	Performed activities			×				7.
10.	Frequency of activities			×				8.
11.	Frequent sequences or combinations of activities			×				9.
12.	Used software technologies			×				10.
13.	Frequency of technology use			×				11.
14.	Frequent sequences or combinations of technology use			×				12.
15.	Importance of the modified module		×		×			15.
16.	Implementation effort				×			19.
17.	Implementation effort of code parts that were not modified since 2 iterations				×			16.
18.	Tasks that generated the implementation effort of code parts that were not modified since 2 iterations				×			17.
19.	Traceability links obtained through non-invasive cost accounting				×			18.
20.	Task life time	×						2.
21.	Autonomation coverage					×		21.
22.	Standardization coverage						×	22.

As we defined the data to extract from the indicators, we faced the problem that frequently we could not collect automatically what we wanted to collect. In such a situation we have several possibilities:

- we change the measurement goal,
- we change the measurement question,
- we change the indicator,
- we change the data, or
- we measure manually.

Of course, changing the goal or the question is not always a viable solution. This sounds like the story of the drunk that lost his keys and searches for them not where he lost them (in the park) but under the streetlight, because this is where there was the light. Nevertheless, sometimes we can switch indicator to measure the same data or we can change data to answer a similar question. As Park et al. put it: "constructing useful indicators is a highly creative process [88]."

In our case, it was a priority to measure automatically so that the measurement could be fed into our autonomation system and displayed on our dashboard.

Some decisions of this type were:

- To answer question 3, i.e., the most frequent reasons why tasks are stuck in intermediate phase for more than one iteration, we look at the source code that was modified because of a task (measure 5) and at the person for whom the task is waiting (measure 6). We hypothesized that we could understand from that information what kind of problem existed. In fact, there might be many other reasons that explain why a task takes longer than expected.
- To answer question 15, i.e., how important each task is for the client, we took the importance of the module in which the code is edited for that task. This is an approximation and it could be that there are tasks that have a different importance for the client than the module it belongs to.
- To answer question 10, i.e., which technologies a software developer uses during the project, we only look at the programs he is using. If the organization uses different hardware technologies and it wants to track the usage of this hardware, we need to come up with another measurement.

11.6.2.8 Define the Measures to Be Used and Make These Definitions Operational

The goal of this section is to refine the measurements identified in the previous section according to two criteria [88]:

- **Communication**: we have to make sure that others know what has been measured, how it was measured, and what has been included and excluded.
- **Repeatability**: we have to make sure that others, equipped with the definition, can repeat the measurements and get essentially the same results.

To accomplish this, we define for each measurement in Table 11.44 how it will be collected (Table 11.45).

Table 11.45 Operational measurements, i.e., measurements that also state how to determine its value

No.	Definition	Result type	Example
1.	Connect to the system that holds the data depicted in Fig. 11.41 and count the number of phases. For example, if the defined phases are Backlog, Selected, Development, Done, Deployment, and Live!, the result is 6	Number	6
2.	Connect to the system that holds the data depicted in Fig. 11.41 and count for each phase the number of tasks whose last status was in that phase	List	Backlog: 20 Selected: 10 Development: 5 ...

(continued)

Table 11.45 (continued)

No.	Definition	Result type	Example
3.	Connect to the system that holds the data shown in Table 11.21 and calculate for each task its age counting the number of days between the current date and the date when they entered their current phase	List	Task 5: 50 days Task 8: 22 days Task 9: 7 days ...
4.	Connect to the system that holds the data shown in Table 11.22 and extract all package names that were modified while a given task was active	List	it.product.module1 it.product.module2 ...
5.	Connect to the system that holds the data shown in Table 11.43 and extract all tasks together with the person because of whom the task is blocked	List	1: Andrea 2: Joe ...
6.	Connect to the system that holds the data shown in Table 11.39 and extract the feature usage estimation	List	Feature 1: high Feature 2: medium ...
7.	Connect to the system that holds the data shown in Table 11.39 and extract the worst-case loss estimation	List	Feature 1: medium Feature 2: low ...
8.	Connect to the system that holds the data depicted in Fig. 11.41 and get all tasks that are in the Backlog	List	Task 1 Task 2 ...
9.	Connect to the system that holds the data depicted in Table 11.41 and get all activities	List	Activity 1 Activity 2 ...
10.	Connect to the system that holds the data depicted in Table 11.41 and calculate the frequency of activities[a]	List	Activity 1: 2512 Activity 2: 5573 ...
11.	Connect to the system that holds the data depicted in Table 11.41 and calculate frequent sequences or combinations of activities[b]	Lists	Frequent activities: 1, 6, 12, ... Frequent combinations: (1,2), (5,2,3) ...
12.	Connect to the system that holds the data depicted in Table 11.41 and get all technologies	List	Technology 1 Technology 2 ...
13.	Connect to the system that holds the data depicted in Table 11.41 and calculate the frequency of technologies[a]	List	Technology 1: 221 Technology 2: 12 ...
14.	Connect to the system that holds the data depicted in Table 11.41 and calculate frequent sequences or combinations of technologies[b]	Lists	Frequent technologies: 2, 3, 5, ... Frequent combinations: (1,2), (5,6) ...

(continued)

Table 11.45 (continued)

No.	Definition	Result type	Example
15.	Connect to the system that holds the data shown in Table 11.39, find the modified modules among the estimates, and assign the importance of the module according to the matrix depicted in Fig. 11.42. If no importance or frequency of use is defined, assume low importance	List	it.product.module1: low it.product.module2: medium ...
16.	Connect to the system that holds the data shown in Table 11.17 and get the calculated duration	List	Class1: 600 min Method5: 120 min ...
17.	Connect to the system that holds the data shown in Table 11.17, find the code parts that were not modified since two iterations, and get the calculated duration	List	Class1: 600 min Method5: 120 min ...
18.	Connect to the system that holds the data shown in Table 11.22, find the code parts that were not modified since two iterations, and get tasks that generated that effort	List	Task1, Task2, ...
19.	Connect to the system that holds the data shown in Table 11.22, and report the tasks and the code those was modified while that tasks were active	List	Task1: class1, class2, ... Task2: method4, class3, ...
20.	Connect to the system that holds the data shown in Table 11.21, and calculate the lifetime of a task as the number of days from when it has the status "Selected" to the day it has the status "Life!"	List	Task1: 5 Task2: 16 ... ...
21.	Connect to the automated verification system and calculate the code coverage of the automated tests[c]	Value	90 %
22.	Estimation by the developers of the part of the development process that is covered by the guidelines	Value	10 %

[a] We calculated the frequency counting how often activities occurred
[b] We considered frequent activities—those activities that occurred more than 20 % of the time. To calculate frequent combinations we used association rule mining [2]
[c] We used EMMA [31], an open-source Java coverage analysis tool to calculate the coverage

11.6.2.9 Identify the Actions That You Will Take to Implement the Measures

To collect the data as we defined above, the team needs to:

- manage their tasks using the described Kanban software,
- extract the task data about the tasks as in Tables 11.21 and 11.43,
- measure the activities of the team as in Table 11.22,
- estimate the importance of features as in Table 11.39,
- measure the effort of the team as in Table 11.41,
- use an automated verification system that calculates the coverage,
- define guidelines for frequent activities, and
- estimate the part of the development process that is covered by the guidelines.

11.6.2.10 Prepare a Plan for Implementing the Measures

The plan to implement the above described measures is described in Sect. 11.3 of this book.

11.6.3 Results

In this case study we describe how we started from questions and obtained a set of measurements that can be regularly collected to support the team to with its efforts to become more and more Lean.

11.6.4 Discussion

The here identified entities and questions are not universal; they have to be identified by the organization. We started broad and as we were refining our questions to measurement goals and measurements, we understood what was possible to collect and what not. This approach is context dependent; that means that in teams different from the one in which this GQM model was developed, we might become invalid data, that is, data that does not measure what we think it measures.

From a Lean perspective, this project contributed to the maximization of value and the creation of knowledge and improvement as summarized in Table 11.46.

Table 11.46 Lean aspects addressed by the third described case

Goal	Strategy	Rationale
Maximization of value	Identification and elimination of waste	The collection of data that helps to fulfill the business goals (and nothing more) maximizes the outcome: we do not waste time and money to collect data that we do not need
Creation of knowledge	GQM model	The creation of the GQM model is a learning process itself. It helps the organization to better understand what its goals are and how they will understand whether they are progressing towards their goals or not
Improvement	Autonomation	The here described approach is a precondition for autonomation: the GQM model interprets the collected data and decides whether the team should be informed or not about something that the measurement probes detected

11.7 Summary

The three case studies that we presented in this chapter illustrate different aspects of our approach to Lean software development. The first study illustrates how a team studied one specific aspect of their development process: whether they spend more time exploring or exploiting. The second study illustrates an approach to study software development costs to understand **where** the team should improve. The third study shows the development of a GQM model to support the non-invasive measurement of how Lean a team is and where the team should improve.

Problems

11.1. Some researchers think that action research is unscientific [49]. Why do you think could somebody think of this?

11.2. Give an example of a hypothesis that can and one that cannot be examined using a laboratory experiment.

11.3. Which is the only logically correct way to attribute indirect costs to the cost-items?

11.4. How can a programmer cheat the system and use a package from the "platform" project, but avoid to get charged for it?

References

1. Adkins, T.: Five myths about time-driven activity-based costing. Sascom magazine. Online: http://www.sas.com/news/sascom/2008q2/feature_abc.html (2008). Accessed 4 Dec 2013
2. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, Washington, DC (1993)
3. Alfresco: Alfresco. Online: http://www.alfresco.com (2013). Accessed 4 Dec 2013
4. Apache Software Foundation: Apache cassandra. Online: http://cassandra.apache.org (2013). Accessed 4 Dec 2013
5. Apache Software Foundation: Apache subversion. Online: http://subversion.apache.org (2013). Accessed 4 Dec 2013
6. Astromskis, S., Janes, A., Mahdiraji, A.R.: Egidio: a non-invasive approach for synthesizing organizational models. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) Proceedings of the International Conference on Software Engineering (ICSE). IEEE, Zürich (2012)
7. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: Supporting cmmi assessment using distributed, non-invasive measurement and process mining. In: Proceedings of the International Conference on Distributed Multimedia Systems (DMS). Knowledge Systems Institute, Brighton (2013)
8. Atteslander, P.: Methoden der empirischen Sozialforschung, 10th edn. Studienbuch Series. Walter de Gruyter, Berlin (2003)
9. Avison, D.E., Lau, F., Myers, M.D., Nielsen, P.A.: Action research. Commun. ACM **42**(1), 94–97 (1999)
10. Baetge, J., Kirsch, H.J., Thiele, S. (eds.): Bilanzrecht, Handelsrecht mit Steuerrecht und den Regelungen des IASB, Kommentar. Stotax Stollfuß Medien, Berlin (2009)
11. Bayou, M., de Korvin, A.: Measuring the leanness of manufacturing systems—a case study of ford motor company and general motors. J. Eng. Technol. Manag. **25**(4), 287–304 (2008)
12. Blichfeldt, B.S., Andersen, J.R.: Research design: creating a wider audience for action research: learning from case-study research. J. Res. Pract. **2**(1) (2006). http://jrp.icaap.org/index.php/jrp/article/download/23/69
13. Boehm, B.W., Clark, B., Horowitz, E., Shelby, R., Westland, C.: An overview of the cocomo 2.0 software cost model. In: Software Technology Conference. ACM, Salt Lake City (1995)
14. Bonitasoft: Bonita. Online: http://www.bonitasoft.com (2013). Accessed 4 Dec 2013
15. Bostock, M.: Data-driven documents. Online: http://d3js.org (2013). Accessed 4 Dec 2013
16. Brown, W.J., Malveau, R.C., McCormick, H.W.S., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)
17. Bugzilla contributors: Bugzilla. Online: http://www.bugzilla.org (2013). Accessed 4 Dec 2013
18. Buzan, T.: Use Your Head. Guild Publishing, London (1984)
19. Checkland, P.B., Holwell, S.: Action research: its nature and validity. In: Kock, N. (ed.) Information Systems Action Research: An Applied View of Emerging Concepts and Methods. Springer's Integrated Series in Information Systems, vol. 13, Springer, New York (2006)
20. Cohen, M.D., March, J.G., Olsen, J.P.: A garbage can model of organizational choice. Adm. Sci. Q. **17**(1), 1–25 (1972)
21. Collins: Collins English Dictionary—Complete & Unabridged, 10th edn. HarperCollins, Glasgow. Online: http://www.collinsdictionary.com (2009). Accessed 4 Dec 2013.
22. Colombo, A., Damiani, E., Gianini, G., Scotto, M., Succi, G.: Identifying individual process patterns by means of non-invasive measurements: preliminary results. In: International Conference on Computational Cybernetics (ICCC). IEEE, Mauritius (2005)
23. Coulter, D., McGrath, G., Wall, A.: Time-driven activity-based costing. Accountancy Irel. **43**(5), 12–15 (2011)
24. Cusumano, M.A., Selby, R.W.: How microsoft builds software. Commun. ACM **40**(6), 53–62 (1997)

25. Dawson, R., Nolan, A.J.: Towards a successful software metrics programme. In: Proceedings of the International Workshop on Software Technology and Engineering Practice (STEP). IEEE Computer Society, Amsterdam (2003)
26. Deloitte Global Services: Ias 38—intangible assets. Online: http://www.iasplus.com/en/standards/ias38 (2013). Accessed 4 Dec 2013
27. Díaz-Ley, M., García, F., Piattini, M.: Implementing a software measurement program in small and medium enterprises: a suitable framework. IEEE Softw. **2**(5), 417–436 (2008)
28. Dobler, M., Kurz, G.: Aktivierungspflicht für immaterielle vermögensgegenstände in der entstehung nach dem rege eines bilmog: kritische würdigung der f&e-bilanzierung im hgb-abschluss de lege ferenda. KoR — Zeitschrift für internationale und kapitalmarktorientierte Rechnungslegung **8**(7/8), 485–493 (2008)
29. Downs, J., Plimmer, B., Hosking, J.G.: Ambient awareness of build status in collocated software teams. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) Proceedings of the International Conference on Software Engineering (ICSE). IEEE, Zürich (2012)
30. Eclipse Foundation: Eclipse ide. Online: http://www.eclipse.org (2013). Accessed 4 Dec 2013
31. EMMA contributors: Emma: a free java code coverage tool. Online: http://emma.sourceforge.net (2013). Accessed 4 Dec 2013
32. European Computer Manufacturers Association: Standard ecma-262. Online: http://www.ecma-international.org/publications/standards/Ecma-262.htm (2011). Accessed 25 May 2014
33. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, London (1998)
34. Fichman, R.G., Kemerer, C.F.: Activity based costing for component-based software development. Inf. Technol. Manag. **3**(1), 137–160 (2002)
35. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Techno. **2**(2), 115–150 (2002)
36. Fowler, M., Beck, K., john Brant, Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading (1999)
37. Fronza, I., Janes, A., Sillitti, A., Succi, G., Trebeschi, S.: Cooperation wordle using pre-attentive processing techniques. In: Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE). IEEE, San Francisco (2013)
38. GIT contributors: Git. Online: http://git-scm.com (2013). Accessed 4 Dec 2013
39. Gold, R.L.: Roles in sociological field observations. Soc. Forces **36**, 217–223 (1958)
40. Goodyear, L.E.: Principles of Accountancy. American Bookkeeping Series. Goodyear-Marshall, Cedar Rapids. Online: https://archive.org/details/principlesofacco00goodrich (1913). Accessed 4 Dec 2013
41. Gopal, A., Krishnan, M.S., Mukhopadhyay, T., Goldenson, D.R.: Measurement programs in software development: determinants of success. IEEE Trans. Softw. Eng. **28**(9), 863–875 (2002)
42. Gopal, A., Mukhopadhyay, T., Krishnan, M.S.: The impact of institutional forces on software metrics programs. IEEE Trans. Softw. Eng. **31**(8), 679–694 (2005)
43. Hall, T., Fenton, N.: Implementing effective software metrics programs. IEEE Softw. **14**(2), 55–65 (1997)
44. Hanspeter, D., Janes, A., Sillitti, A., Succi, G.: Improving the identification of traceability links between source code and requirements. In: Proceedings of the International Conference on Distributed Multimedia Systems (DMS). Knowledge Systems Institute, Miami Beach (2012)
45. Hanspeter, D., Janes, A., Sillitti, A., Succi, G.: Semi-automatic requirement tracing in modified code: an eclipse plugin. In: Proceedings of the International Conference on Distributed Multimedia Systems (DMS). Knowledge Systems Institute, Miami Beach (2012)
46. Hattori, L.P., Lanza, M., Robbes, R.: Refining code ownership with synchronous changes. Empir. Softw. Eng. **17**(4–5), 467–499 (2012)

47. Hibbs, C., Jewett, S.P., Sullivan, M.: The Art of Lean Software Development: A Practical and Incremental Approach. Theory in Practice. O'Reilly Media, Sebastopol (2009)
48. Hirsch, J.E.: An index to quantify an individual's scientific research output that takes into account the effect of multiple coauthorship. Scientometrics **85**(3), 741–754 (2010)
49. Hope, K.W., Waterman, H.A.: Praiseworthy pragmatism? validity and action research. J. Adv. Nurs. **44**(2), 120–127 (2003)
50. Hopkins, D.: A Teacher's Guide to Classroom Research, 4th edn. Open University Press, Maidenhead (2008)
51. Horsch, J.: Kostenrechnung: Klassische und neue methoden in der unternehmenspraxis. Zeitschrift für Betriebswirtschaft **80**(10), 1121–1122 (2010)
52. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Professional, Reading (1996)
53. International Financial Reporting Standards Foundation: International accounting standard 38, intangible assets, technical summary. Online: http://www.ifrs.org/IFRSs/Documents/English%20IAS%20and%20IFRS%20PDFs%202012/IAS%2038.pdf (2012). Accessed 4 Dec 2013
54. Ireland, J.: Principles of Accounting. Undergraduate study in Economics, Management, Finance and the Social Sciences. University of London, London (2005)
55. Iversen, J., Mathiassen, L.: Lessons from implementing a software metrics program. In: Proceedings of the Hawaii International Conference on System Sciences (HICSS). IEEE, Maui (2000)
56. Jalote, P.: An Integrated Approach to Software Engineering, 3 edn. Texts in Computer Science Series. Springer, New York (2005)
57. Janes, A., Succi, G.: To pull or not to pull. In: Arora, S., Leavens, G.T. (eds.) Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, Orlando (2009)
58. JDepend contributors: Jdepend. Online: http://clarkware.com/software/JDepend.html (2013). Accessed 4 Dec 2013
59. Jenkins CI contributors: Jenkins ci. Online: http://jenkins-ci.org (2013). Accessed 4 Dec 2013
60. Jersey contributors: Jersey. Online: https://jersey.java.net (2013). Accessed 4 Dec 2013
61. Anfara V.A, Jr., Mertz, N.T.: Theoretical Frameworks in Qualitative Research. Sage Publications, Thousand Oaks (2006)
62. JUnit contributors: Junit. Online: http://sourceforge.net/projects/junit (2013). Accessed 4 Dec 2013
63. Kaplan, R.S., Anderson, S.R.: Time-driven activity-based costing. Harv. Bus. Rev. **82**(11), 131–138 (2004)
64. Kaplan, R.S., Norton, D.: The balanced scorecard: measures that drive performance. Harv. Bus. Rev. **70**(1), 71–79 (1992)
65. Keele, R.: Nursing Research and Evidence-Based Practice. Jones & Bartlett Learning, Sudbury (2010)
66. Kemmis, S., McTaggart, R.: Participatory action research: communicative action and the public sphere. In: Denzin, N.K., Lincoln, Y.S. (eds.) The SAGE Handbook of Qualitative Research, 3rd edn. Sage, Thousand Oaks (2005)
67. Kilpi, T.: Implementing a software metrics program at nokia. IEEE Softw. **18**(6), 72–77 (2001)
68. Kimball, R.: The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. Wiley, New York (1996)
69. Lasser, J.K.: Handbook of Cost Accounting Methods. D. Van Nostrand Company, New York (1949)
70. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, Shanghai (2006)
71. Lehman, M.: Programs, life cycles, and laws of software evolution. Proc. IEEE **68**(9), 1060–1076 (1980)

72. Lewin, K.: Action research and minority problems. J. Soc. Issues **2**(4), 34–46 (1946)
73. Lewin-Koh, N.: Hexagon binning: an overview. Online: http://cran.r-project.org/web/packages/hexbin/vignettes/hexagon_binning.pdf (2011). Accessed 4 Dec 2013
74. Lieberman, M.B., Montgomery, D.B.: First-mover strategies. Special Issue on Strategy Content Research. Strat. Manag. J. **9**, 41–58 (1988)
75. March, J.G.: Exploration and exploitation in organizational learning. Organ. Sci. **2**(1), 71–87 (1991)
76. Marciuska, S., Gencel, C., Abrahamsson, P.: Exploring how feature usage relates to customer perceived value: a case study in a startup company. In: Herzwurm, G., Margaria, T. (eds.) Software Business. From Physical Products to Software Services and Solutions. Lecture Notes in Business Information Processing, vol. 150. Springer, New York (2013)
77. Maskell, B.H., Baggaley, B.L.: Lean accounting: what's it all about? Association for Manufacturing Excellence's Target Magazine (2006)
78. Mertler, C.A.: Action Research: Improving Schools and Empowering Educators. Sage Publications, Thousand Oaks (2011)
79. Microsoft: Microsoft c#. Online: http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx (2013). Accessed 4 Dec 2013
80. Microsoft: Microsoft office. Online: http://office.microsoft.com (2013). Accessed 4 Dec 2013
81. Microsoft: Microsoft visual studio. Online: http://www.microsoft.com/visualstudio (2013). Accessed 4 Dec 2013
82. Microsoft: .net. Online: http://www.microsoft.com/net (2013). Accessed 4 Dec 2013
83. Mindscape: Karotz. Online: http://www.karotz.com (2013). Accessed 4 Dec 2013
84. Monfreda, N.: Le sanzioni amministrative in materia di ii.dd. ed iva. Rivista della Scuola superiore dell'economia e delle finanze a cura del Centro Ricerche Documentazione Economica e Finanziaria, vol. VII, Issue 2. Online: http://rivista.ssef.it/site.php?page=20051011123911977 (2010). Accessed 4 Dec 2013
85. Nord, R.L., Ozkaya, I., Sangwan, R.S.: Making architecture visible to improve flow management in lean software development. IEEE Softw. **29**(5), 33–39 (2012)
86. Oracle: Java. Online: http://www.java.com (2013). Accessed 4 Dec 2013
87. Oracle: Jsp. Online: http://www.oracle.com/technetwork/java/javaee/jsp/index.html (2013). Accessed 4 Dec 2013
88. Park, R.E., Goethert, W.G., Florac, W.A.: Goal-driven software measurement—a guidebook. Technical Report CMU/SEI-96-HB-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1996)
89. Parkinson, S.T., Counsell, S., Norman, M., Hierons, R.M., Lycett, M.: The precursor to an industrial software metrics program. In: Proceedings of the International Conference on Information Technology Interfaces (ITI). University of Zagreb, Cavtat (2008)
90. Pentaho contributors: Pentaho. Online: http://www.pentahobigdata.com (2013). Accessed 4 Dec 2013
91. Pólya, G.: How to Solve It: A New Aspect of Mathematical Method. Science Study Series. Doubleday Anchor Books, Garden City (1957)
92. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, Upper Saddle River (2006)
93. Popper, K.: The Logic of Scientific Discovery. Routledge, London (2002)
94. Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 5th edn. Project Management Institute, Newtown Square (2013)
95. Reiß, M.: Das kongruenzprinzip der organisation. Wirtschaftswissenschaftliches Studium **11**, 75–78 (1982)
96. Robbes, R., Röthlisberger, D.: Using developer interaction data to compare expertise metrics. In: Working Conference on Mining Software Repositories (MSR). IEEE, San Francisco (2013)
97. Santos, P.S.M.d., Travassos, G.H.: Action research use in software engineering: an initial survey. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Computer Society, Lake Buena Vista (2009)

98. Astromskis, S., Janes, A., Sillitti, A., Succi, G.: An approach to non-invasive cost accounting. In: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Verona, Italy (2014)
99. Seicht, G.: Moderne Kosten- und Leistungsrechnung: Grundlagen und praktische Gestaltung, 8th edn. Linde, Wien (1995)
100. Sidiropoulos, A., Katsaros, D., Manolopoulos, Y.: Generalized hirsch h-index for disclosing latent facts in citation networks. Scientometrics **72**(2), 253–280 (2007)
101. Software Engineering Standards Committee of the IEEE Computer Society: IEEE standard for software project management plans. IEEE Std 1058-1998 (1998)
102. Sommerville, I.: Software Engineering, 9th edn. Addison-Wesley, Reading (2010)
103. Standards Coordinating Committee of the Computer Society of the IEEE: IEEE standard glossary of software engineering terminology. IEEE 610.12-1990 (1990)
104. Szyperski, C.: Component Software: Beyond Object Oriented Programming. Addison-Wesley Professional, Reading (2002)
105. The Linux Foundation: Linux. Online: http://www.linux.org (2013). Accessed 4 Dec 2013
106. Thompson, J.L.: Strategic Management: Awareness and Change. Chapman & Hall, London (1993)
107. Trac contributors: Trac. Online: http://trac.edgewall.org (2013). Accessed 4 Dec 2013
108. Venkatesh, V., Bala, H.: Technology acceptance model 3 and a research agenda on interventions. Decis. Sci. **39**(2), 273-315 (2008)
109. Vinodh, S., Chintha, S.K.: Leanness assessment using multi-grade fuzzy approach. Int. J. Prod. Res. **49**(2), 431–445 (2011)
110. Wegmann, G.: The activity-based costing method: development and applications. IUP J. Account. Res. Audit Pract. **8**(1), 7–22 (2008)
111. Whetten, D.A.: What constitutes a theoretical contribution? Acad. Manag. Rev. **14**(4), 490–495 (1989)
112. Yin, R.K.: Case Study research, Design and Methods, 3rd edn. Applied Social Research Methods Series. Sage Publications, Thousand Oaks (2009)

Chapter 12
Conclusion

Jeder, der sich die Fähigkeit erhält, Schönes zu erkennen, wird nie alt werden.
(Anyone who keeps the ability to see beauty never grows old.)

Franz Kafka

12.1 Introduction

As an applied researcher, we prefer to confront our ideas and models earlier and more often with the reality than our theoretical colleagues. Doing so increases the relevance of our research and forces us to keep a holistic view towards our research, since it is not always possible to apply research just to a part of the world. This is—according to our opinion—the most important lesson of Lean Thinking: optimize the whole.

The foreword of Taiichi Ono's book titled "Toyota Production System" contains a "simple but brilliant [4]" description of the goal of Lean Manufacturing, which is depicted in Fig. 12.1. Taiichi Ohno explains this figure saying: "All we are doing is looking at the time line, from the moment the customer gives us an order to the point when we collect the cash. And we are reducing that time line by removing the non-value-added wastes [4]."

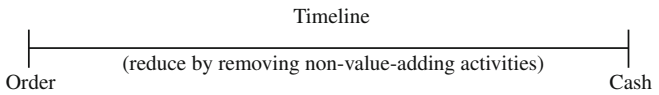


Fig. 12.1 A birds-eye view on the Toyota Production System [4]

The approach described in this book aims to make "go and see" easier, a method that in Lean is called "Gemba." Gemba represents the search for knowledge, i.e., the search for understanding the execution of processes, why certain activities take the time they take, where we have to improve, and so on. Gemba promotes a culture of measurement, i.e., a culture that wants to understand the best strategy to achieve a given goal. This book describes how practitioners can use non-invasive

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9_12

measurement to implement autonomation and get feedback about their progress towards getting more and more Lean.

Still we have to be careful: measurement and transparency do not work everywhere; it depends on the organizational culture. Organizational culture "is the basic pattern of shared assumptions, values, and beliefs considered to be the correct way of thinking about and acting on problems and opportunities facing the organization [3]." Deal and Kennedy put it like this: "the way things get done around here [2]."

If the culture promotes "the strongest survives," nobody will be interested in sharing data about his own performance since he might be afraid that somebody could find out a weakness. On the other hand, if the culture supports collaboration and learning, the individual team members will agree to measure their activities and their software development process outcome to improve.

12.1.1 Lessons Learned

We would like to point out some lessons we learned as we supported organizations becoming Lean using a measurement framework as described in this book:

- *Do not expect that you get told what to do.* We cannot go to a manager and ask: "OK, so what should we measure to help you become Lean?" He (probably) will not be able to give you an answer because he does not know how measurement contributes to becoming Lean. This is similar to what we said in Chap. 3 that we need to "convert business objectives into requirements that take into consideration the technical possibilities and their costs." The manager is not aware of the technical possibilities and their costs. It is our task to study the business needs of the organization and to propose opportunities to achieve them. This is why we stressed the use of the GQM$^+$Strategies approach so much.
- *Explain what you are trying to achieve.* Lean Thinking itself requires the involvement of the person that is actually working in the production process, because this person has the most knowledge about how to improve it. With manual measurement, developers are aware of *what* is collected, because they have to collect and provide the required data. Since our approach is based on non-invasive measurement, the developers (and the other stakeholders) are not "naturally" aware of the data that is now available; that means that they cannot help us with new, innovative ways to analyze or to interpret it. As a consequence, we have to spend more time (compared to manual measurement) explaining what we are collecting, how we are processing the data, what information we gain from it, etc. to obtain the acceptance and involvement of the team.
- *Measure only what you need.* A typical mistake at the beginning is to measure as much as you can. It seems that the more data we have, the better decisions we can make. We did this mistake, and—besides understanding that this is not true—we

realized that we spent a lot of time and money to build a system that we did not need.
- *Proceed incrementally.* Begin with some few measurement goals and extend them later. As the benefits of having non-invasive measurement and Andon in place become visible, you can decide if it is worthy to extend the framework to more measurements, more information, and more knowledge. We pointed out already in Chap. 9 that data comes with a cost.
- *Ensure privacy and confidentiality.* Privacy (the possibility for the individual to control one's private life or personal affairs [1]) and Confidentiality (the possibility for the individual to control one's data) have to be ensured. Everybody that is part of the measurement process should have the possibility to control which data are collected about himself and who has access to it.
- *Verify the validity of the collected data.* The correct functioning of the measurement system should be constantly verified, possibly using a Jidoka approach. Some examples of such problems are:
  - a measurement probe stops working;
  - a computer gets disconnected from the network and the collected data are not uploaded anymore;
  - the assumptions under which a measurement probe was developed are not valid anymore, and the collected data do not measure anymore what they should.

To give every interested reader a hint of how to design such a framework, Appendix B describes the architecture of our implementation.

Problems

12.1. Design a prototype architecture of a system to keep everybody informed about the number of tasks with a high severity that are present in the backlog and if the last nightly build failed.

References

1. Collins: Collins English Dictionary — Complete & Unabridged, 10th edn. HarperCollins (2009). Online: http://www.collinsdictionary.com. Accessed 4 Dec 2013
2. Deal, T.E., Kennedy, A.A.: Corporate cultures: the rites and rituals of corporate life. Addison-Wesley, Reading (1982)
3. McShane, S., Glinow, M.V.: Organizational Behavior. McGraw-Hill, New York (2000)
4. Ōno, T.: Toyota Production System: Beyond Large-Scale Production. Productivity Press, Cambridge (1988)

Appendix A
If Architects Had to Work Like Software Developers[1]

Dear Mr. Architect,

Please design and build me a house. I am not quite sure of what I need, so you should use your discretion.

My house should have between two and forty-five bedrooms. Just make sure the plans are such that the bedrooms can be easily added or deleted. When you bring the blueprints to me, I will make the final decision of what I want. Also, bring me the cost breakdown for each configuration so that I can arbitrarily pick one.

Keep in mind that the house I ultimately choose must cost less than the one I am currently living in. Make sure, however, that you correct all the deficiencies that exist in my current house (the floor of my kitchen vibrates when I walk across it, and the walls don't have nearly enough insulation in them).

As you design, also keep in mind that I want to keep yearly maintenance costs as low as possible. This should mean the incorporation of extra-cost features like aluminum, vinyl, or composite siding. (If you choose not to specify aluminum, be prepared to explain your decision in detail.)

Please take care that modern design practices and the latest materials are used in the construction of the house, as I want it to be a showplace for the most up-to-date ideas and methods. Be alerted, however, that the kitchen should be designed to accommodate, among other things, my 1952 Gibson refrigerator.

To ensure that you are building the correct house for our entire family, make certain that you contact each of our children and also our in-laws. My mother-in-law will have very strong feelings about how the house should be designed, since she visits us at least once a year.

Make sure that you weigh all of these options carefully and come to the right decision. I, however, retain the right to overrule any choices that you make.

[1] The author of this letter is unknown.

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9

Please don't bother me with small details right now. Your job is to develop the overall plans for the house: get the big picture. At this time, for example, it is not appropriate to be choosing the color of the carpet. However, keep in mind that my wife likes blue.

Also, do not worry at this time about acquiring the resources to build the house itself. Your first priority is to develop detailed plans and specifications. Once I approve these plans, however, I would expect the house to be under roof within 48 h.

While you are designing this house specifically for me, keep in mind that sooner or later I will have to sell it to someone else. It therefore should have appeal to a wide variety of potential buyers. Please make sure before you finalize the plans that there is a consensus of the population in my area that they like the features this house has.

I advise you to run up and look at my neighbor's house that he constructed last year. We like it a great deal. It has many features that we would also like in our new home, particularly the 25 m swimming pool. With careful engineering, I believe that you can design this into our new home without impacting the final cost.

Please prepare a complete set of blueprints. It is not necessary at this time to do the real design, since they will be used only for construction bids. Be advised, however, that you will be held accountable for any increase of construction costs as a result of later design changes.

You must be thrilled to be working on an interesting project as this! To be able to use the latest techniques and materials and to be given such freedom in your designs is something that can't happen very often.

Contact me as soon as possible with your complete ideas and plans.

Sincerely yours,

Dr. Joe Plumber

P.S.: My wife has just told me that she disagrees with many of the instructions that I've given you in this letter. As architect, it is your responsibility to resolve these differences. I have tried in the past and have been unable to accomplish this. If you can't handle this responsibility, I will have to find another architect.

P.P.S.: Perhaps what I need is not a house at all, but a travel trailer. Please advise me as soon as possible if this is the case.

Appendix B
A Possible Architecture for a Measurement Framework

This appendix presents the architecture of a measurement and visualization framework to support Lean software development as described in this book. We follow the 4 + 1 view model of architecture proposed by Philippe B. Kruchten [18]. This model is based on the idea that an architecture cannot be described precisely only from one point of view since it accomplishes the requirements of different stakeholders, e.g., users, programmers, or system operators. The 4 + 1 model does not try to visualize all aspects of an architecture in one diagram, but proposes different views, each view with a different objective. The five views are the logical, process, physical, development view, and the scenarios. We describe each view in Table B.1.

Table B.1 The views of the 4 + 1 view model of architecture

Name	Content
Logical view	Describes the logical entities of the architecture, i.e., the services the system provides to its end users
Process view	Describes the execution of operations by the entities described in the logical view. It describes the runtime behavior of the architecture
Physical view	Describes how the software is deployed on hardware
Development view	Describes how the code is organized within the development environment
Scenarios	Describes usage scenarios or, in other words, test cases of the architecture

Scenarios describe what to expect from the architecture. They act as test cases, specifying what the architecture should be capable of, given a certain input. We use the schema proposed by Bass et al. [2] to describe a scenario. It consists of:

- the **source of stimulus**: the description of the entity that generated a stimulus, i.e., something that requires a reaction from the architecture,

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9

- the **stimulus**: the stimulus to which the system has to react,
- the **environment**: the external conditions under which the stimulus arrived,
- the **artifact**: the target of the stimulus,
- the **response**: the expected response of the architecture, and
- the **response measure**: a description how we will measure that the expected reaction took place.

The single scenarios are implemented through architectural decisions. For example, the choice to use an NoSQL database such as Apache Cassandra [1] instead of a relational database has impacts on the quality of the architecture. It increases, for example, scalability, because Apache Cassandra is constructed to allow adding nodes (i.e., hardware) to a database cluster incrementally. On the other hand, Apache Cassandra does not support referential integrity, transactions, the locking of records, object-relational mapping, etc., which decreases the usability for developers.

This is an example of typical trade-offs one has to accept when designing an architecture: in the example of Apache Cassandra, we have to give up usability to obtain scalability. In fact, to design an architecture often means to balance the trade-offs between different solutions [2].

B.1 Scenarios

The most important scenarios that guided us in the development of the here described architecture are as follows (we use the terminology recommended in [4]):

1. Modifiability: The measurement framework must support the collection of measurements about employed resources, the performed processes, and the produced artifacts. It must be possible to add, modify, and remove by the end user on runtime (a) properties of the described resources, processes, and artifacts; (b) measurement transformations (analyses that have to be performed before the data can be visualized); and (c) measurement visualizations.

   - the **source of stimulus**: System administrator;
   - the **stimulus**: Add, modify, or remove a property, transformation, or visualization;
   - the **environment**: At runtime;
   - the **artifact**: The database and all systems that interact with data, transformations, or visualizations;
   - the **response**: The changed property, transformation, or visualization configuration is stored in the database and available in the rest of the system;
   - the **response measure**: The modification should have instant effect, without downtime for the end users.

2. Usability: The measurement framework must support the collection of measurements in a distributed way, i.e., coming from different computers. If the server is

not reachable, the data must be collected on the client side and sent to the server as it is reachable again.

- the **source of stimulus**: End user;
- the **stimulus**: Wants to upload data;
- the **environment**: At runtime;
- the **artifact**: Measurement system;
- the **response**: The measurement system caches the data locally and, as a connection to the server is available, uploads it;
- the **response measure**: The data are uploaded with the minimal performance impact to the system of the end user.

3. Usability: The measurement framework must support the visualization of measurements or the result of an analysis based on measurements on the web. No software should be installed on the client machines to visualize data, and the visualizations itself must be small enough to be transferred over the Internet.

   - the **source of stimulus**: End user;
   - the **stimulus**: Wants to visualize measurements or the result of an analysis based on measurements;
   - the **environment**: At runtime;
   - the **artifact**: Measurement system;
   - the **response**: The measurement system retrieves the required visualization from the server and visualizes it on a web page;
   - the **response measure**: The measurements or visualizations are displayed immediately (or with a minimal delay) after the request.

4. Usability: The measurement framework must support the organization of measurements in the form of dashboards as described in Chap. 10.

   - the **source of stimulus**: End user;
   - the **stimulus**: Wants to organize measurements in the form of dashboards;
   - the **environment**: At runtime;
   - the **artifact**: Measurement system;
   - the **response**: The measurement system supports the user in the definition of dashboards as described in Chap. 10;
   - the **response measure**: A dashboard can be created picking goals, questions, and indicators. The time to accomplish this depends only on the size and complexity of the dashboard.

5. Usability: The measurement framework must collect the measurements with a minimal impact to the performance of the system of the end user.

   - the **source of stimulus**: End user;
   - the **stimulus**: Wants to collect measurements;
   - the **environment**: At runtime;
   - the **artifact**: Measurement system, operating system, and applications running on the system of the end user;

- the **response**: The measurement system collects the measurements and submits them to the server;
- the **response measure**: The user does not notice any performance loss of the operating system or the applications he is using.

B.2 Logical View

Before we explain the structure of the database, we want to illustrate the data flow, i.e., where data are gathered, how they are processed, where they are stored, and how they are extracted to be visualized to the end user.

Figure B.1 illustrates the data flow diagram using the Yourdon/DeMarco notation.

The Yourdon/DeMarco Notation in Short. The Yourdon/DeMarco notation uses the following elements to describe a data flow:

- rectangles to describe the start and end points,
- two horizontal lines to describe data containers (called data stores) from which one can read and write,
- circles to describe functions that transform data, and
- arrows that describe the actual flow of data from one point to another.

A detailed explanation of the notation can be found in [29].

This data flow follows the data flow for a typical data warehouse [17]. The data are extracted by the measurement probes and sent to the message queue. The message queue is a data store that has two functions: (a) to minimize the time that the client has to wait for the server to process the data and (b) to perform an initial consistency check of the data. Moreover, it is used for debugging purposes to see the incoming data before it is stored in the data warehouse. If the incoming data are correct, a process reads the data waiting in the message queue and uploads in into our Apache Cassandra data warehouse, i.e., our central repository. The tables present in this repository are shown in Fig. B.2.

The data warehouse contains all the history of the data, but that is rarely needed. Therefore, a process that extracts only the data that is needed for a particular visualization extracts a part of the data and stores it in a separate database (called "Project data mart" in the diagram"). From there, a visualization can read the data and prepare the output that we submit to the browser of the user.

All three processes (the three circles in the diagram) in our architectures are based on a plugin architecture. That means that a developer can:

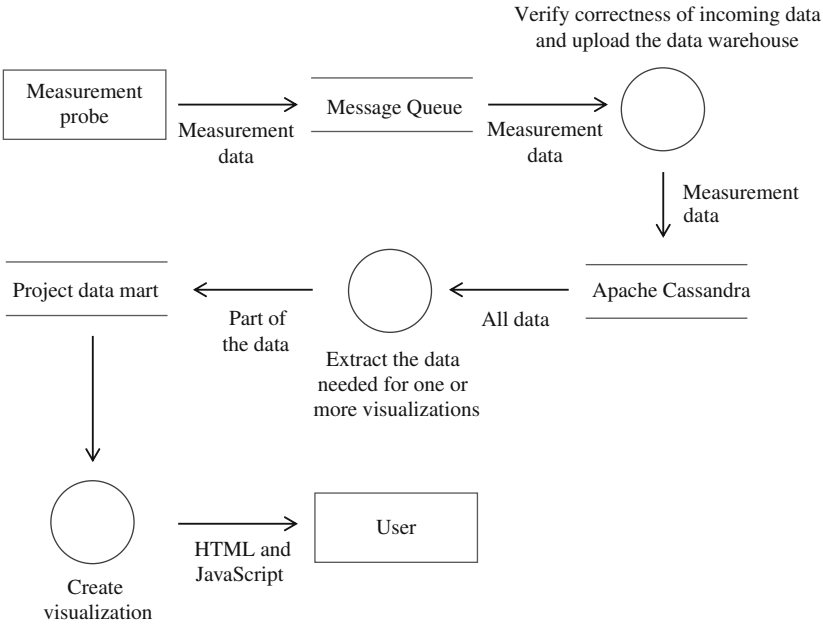


Fig. B.1 Data flow in the measurement framework

- add his own message queue plugin that handles the upload of a part of the data that enters the message queue into the data warehouse,
- add his own data mart plugin that handles the extraction of a part of the data to prepare it for a visualization, and
- add his own visualization plugin that creates a visualization.

This approach allows us to extend the system with new types of data or new visualizations without changing the core of the system (see scenario 1). In the database, these plugins are called ETL plugins (ETL stands for “Extract Transform Load" and is used in the data warehouse terminology to describe processes that extract data from a source, transform it, and load it into a target data store).

In the following we will explain the purpose of the different tables present in the database. The tables in Fig. B.2 represent all the data that we collect. In our implementation these tables are stored in an Apache Cassandra database [1].

We defined a generic data structure for product and process measurements. The three tables handling the data for product measurements are:

- The table “product_metrics_item” stores data about products, i.e., software projects. Product measurements describe a given point in time. This point in time is stored in “product_metrics_run.” We store product metrics together with pointers to the piece of code they describe. Some example records of the “product_metrics_item” table for the run number 5 are shown in Table B.2.

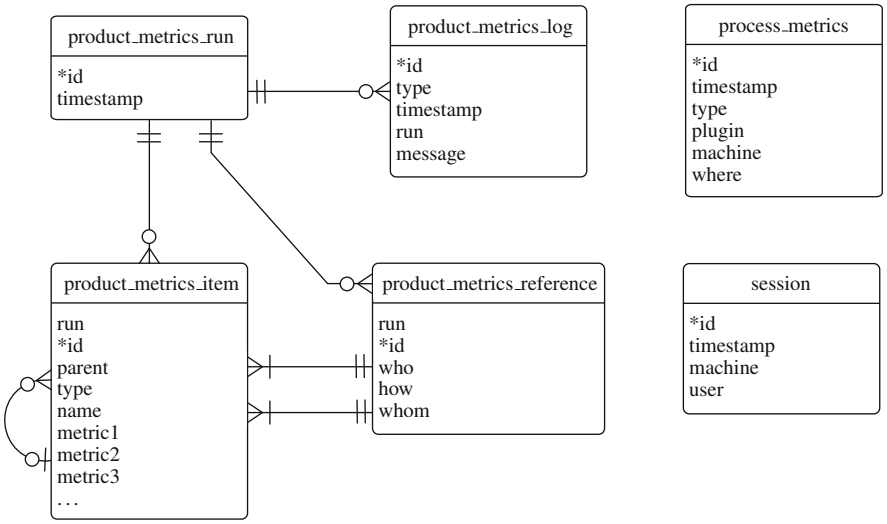


Fig. B.2 Database tables storing data about product measurements, process measurements, and user sessions

- The table "product_metrics_reference" stores references (or links) between items in the "product_metrics_item" table. The column "how" describes the type of relationship. For example, we store if one class extends another class or if a class is stored within a file, and so on.
- The table "product_metrics_log" stores warnings or error messages that arise during the data upload.

The table "process_metrics" stores all the events that we collect about the software development process. For each event we store the type of event, e.g., effort, which plugin reported the measurement, from which machine we obtained the measurement and where, and in which application or piece of code the event was triggered.

To identify each machine, we assign a GUID [20] to each machine during installation. As the machine sends data to the server, it also submits its GUID. Knowing from which machine data are coming from is useful for debugging purposes.

To describe where the event was triggered, we use a JSON [8] object. This allows us to have the necessary flexibility to add new measurements without having to change the database structure (see scenario 1).

Examples of what we store in the "where" column are shown in Listings B.1 and B.2. The location described in Listing B.1 refers to a method "setResult" with one parameter of type "double" within the class "AggregationResult."

```
{
  "file_name":"src/jobs/AggregationResult.java",
  "id":[
    {
      "type":"namespace",
      "name":"jobs"
    },
    {
      "type":"class",
      "name":"AggregationResult"
    },
    {
      "type":"method",
      "name":"setResult(double)"
    }
  ]
}
```

Listing B.1 Example event 1

The location described in Listing B.2 refers to an application that had the process name "Inkscape" and had the caption "/data/figures/139.svg - Inkscape."

```
{
  "process":"inkscape",
  "caption":"/data/figures/139.svg - Inkscape"
}
```

Listing B.2 Example event 2

In the first version of our measurement framework, we had also the user column in the table "process_metrics." Unfortunately it sometimes happened that the user forgot to log off and recorded effort in the name of another user. To avoid updating thousands (or millions) of records after such a case, we separated the user column from the process metrics and stored it in the table "session." Table B.3 shows some example records of the session table.

From these example records, we see the user "6aac0291-919e-4443-…" (also a random GUID assigned upon installation) logged on machine "c14545ee-c5e2-41e5-…" at 9:00. Then, at 10:00, user "8e09c244-02ef-4899-…" logged in on that machine, and so on. Since we know the machines from which the events are reported and we know the users that work on each machine, we can assign the events to the users. If now a user notices that he forgot to log off or that data are by mistake assigned to him, he can update the "sessions" table with the corrected data.

Table B.2 Example records for the "product_metrics_item" table extracted using the Eclipse [9] parser [15, 25]

Run	Id	Parent	Type	Name	LOC	CC[a]	WMC[b]
5	1		Namespace	It	2,000		
5	2	1	Namespace	Unibz	2,000		
5	3	2	Namespace	Measurement	2,000		
5	4	3	Namespace	Product	2,000		
5	5	4	Class	Task	200		10
5	6	4	Class	Tenant	300		20
5	7	4	Class	Owner	500		25
5	8	7	Method	Add(int id)	500	5	
	9	7	Method	Remove(boolean force)	500	8	

[a] Cyclomatic Complexity [23]
[b] Weighted Methods per Class, the sum of the complexities of all the methods in a class [7]

Table B.3 Example records for the "session" table

Id	Timestamp	Machine	User
1	08.01.2014 09:00	c14545ee-c5e2-41e5-...	6aac0291-919e-4443-...
2	08.01.2014 10:00	c14545ee-c5e2-41e5-...	8e09c244-02ef-4899-...
3	08.01.2014 11:00	3d68b8d0-b5df-4728-...	6aac0291-919e-4443-...
4	08.01.2014 12:00	6acbe0a3-ec85-4d5a-...	3c569127-51f5-44df-...
5	08.01.2014 15:00	6acbe0a3-ec85-4d5a-...	b5dd6b5b-abcf-44e4-...

Figure B.3 describes the database tables storing data about the analysis of data to prepare it for the visualizations. The table "etl_plugin" stores the locations of the jar files, i.e., the plugins that we installed in our measurement system. We mentioned for Fig. B.1 that any function in the data flow can be defined as a plugin.

If the measurement system wants to execute a plugin, it obtains the location of the jar file that contains the plugin from the table "etl_plugin," adds the corresponding jar file that is stored in a known folder on the server to the class path, instantiates it, and invokes it. The table "etl_plugin_settings" stores optional settings that a plugin might require, and "etl_plugin_log" is a table in which a plugin can put debugging messages.

When a data mart plugin extracts a part of the data from the data warehouse, we call that an analysis run. After one analysis run, one record is added to table "analysis_run." The result of the run is:

- a list of items that are described (stored in the table "analysis_run_item"),
- data about the reported items (stored in the table "analysis_run_data"), and
- relations between the items (stored in the table "analysis_run_reference").

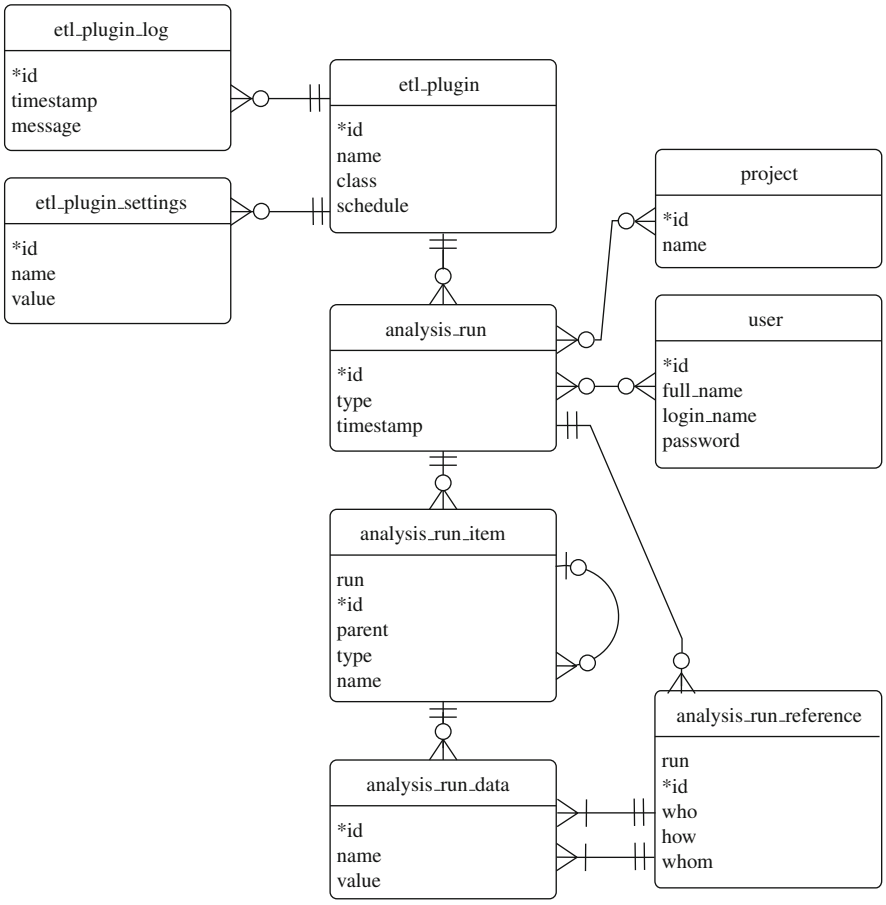


Fig. B.3 Database tables storing data about plugins, analyses, projects, and users

The result of an analysis run is entirely defined by the visualization plugin that extracts the data from the data warehouse and prepares it for a visualization. It can be data that describes one or more projects and one or more users. Therefore, a visualization plugin has to specify for which user(s) and for which project(s) the data are relevant. This is accomplished by adding references to the tables "user" and "project."

The tables of Fig. B.3 that start with "analysis_run" are stored in a specific project database (the project data mart). All other tables are stored in the database "configuration," a database that is common to all projects in our system.

The third part of the database schema of our implementation is depicted in Fig. B.4. It shows the tables we use to implement role-based access control [10]. We define user roles in the table "role" and link entries in the table "users" to roles. Then, we define which roles a user has for each specific project in the table "project_role_user."

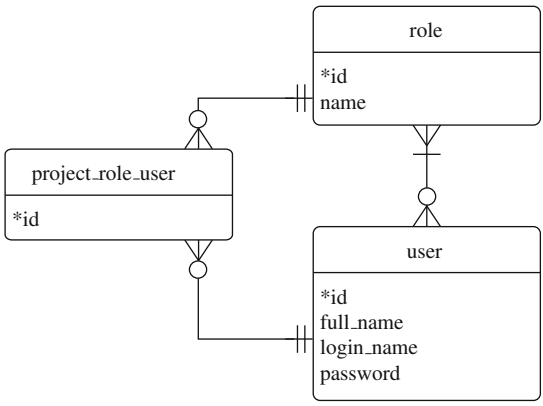


Fig. B.4 Database tables storing data about users and projects

The fourth part of the database schema of our implementation is depicted in Fig. B.5, and it shows the tables we use to store the dashboards (see Chap. 10). We link dashboards to a project, which means that they are visible for the users of that project. This decision is made by the designer of the dashboard. He can decide to share a dashboard with one or more projects.

In our approach, the decision of which measurements to consider and how to visualize them is handled by a visualization plugin. By choosing a specific visualization plugin, we choose the measurements it uses and we choose how these measurements are visualized. If we look at the dashboard from a GQ(I)M point of view, a plugin represents the indicator. This is why, the table "gqim_indicator" is linked to a "etl_plugin."

The first maybe intuitive alternative we considered when designing the system was to allow the user to define all elements of the GQ(I)M interactively. The user should then be able to pick a set of measurements and define how the system should build an indicator out of these measurements. Then he could link the indicator to one or more questions and those questions to goals. This would have meant for us that we had to implement some sort of chart wizard as it can be found in spreadsheet applications. This chart wizard would have to allow the user to define how the measurements have to be combined to draw the chart.

We decided that—considering our skills—the effort to develop a chart wizard as described above was much higher than developing a plugin-based system. Moreover, we were confident that a plugin that could be written in Java that defines an indicator gives to the author of an indicator a much higher flexibility than a chart wizard created by us (see scenario 4).

The three tables "gqim_goal," "gqim_question," and "gqim_indicator" are tables that are not linked with each other but just to one or more projects. This is why they do not contain the actual GQ(I)M for a dashboard, but they are templates for goals, questions, and indicators that can be used for GQ(I)Ms in various dashboards.

This means that if a user defines an indicator "Source code complexity of a project," he can share this indicator (together with its settings) with other projects, where users can reuse it in their GQ(I)Ms.

When a user defines a dashboard, he picks from the available goals, questions, and indicators and saves this definition in the tables "gqim_definition_goal," "gqim_definition_question," and "gqim_definition_indicator."

All tables of Figs. B.4 and B.5 are stored in the database "configuration," a database that is common to all projects in our system. This database is modifiable through the administrative interface of the system.

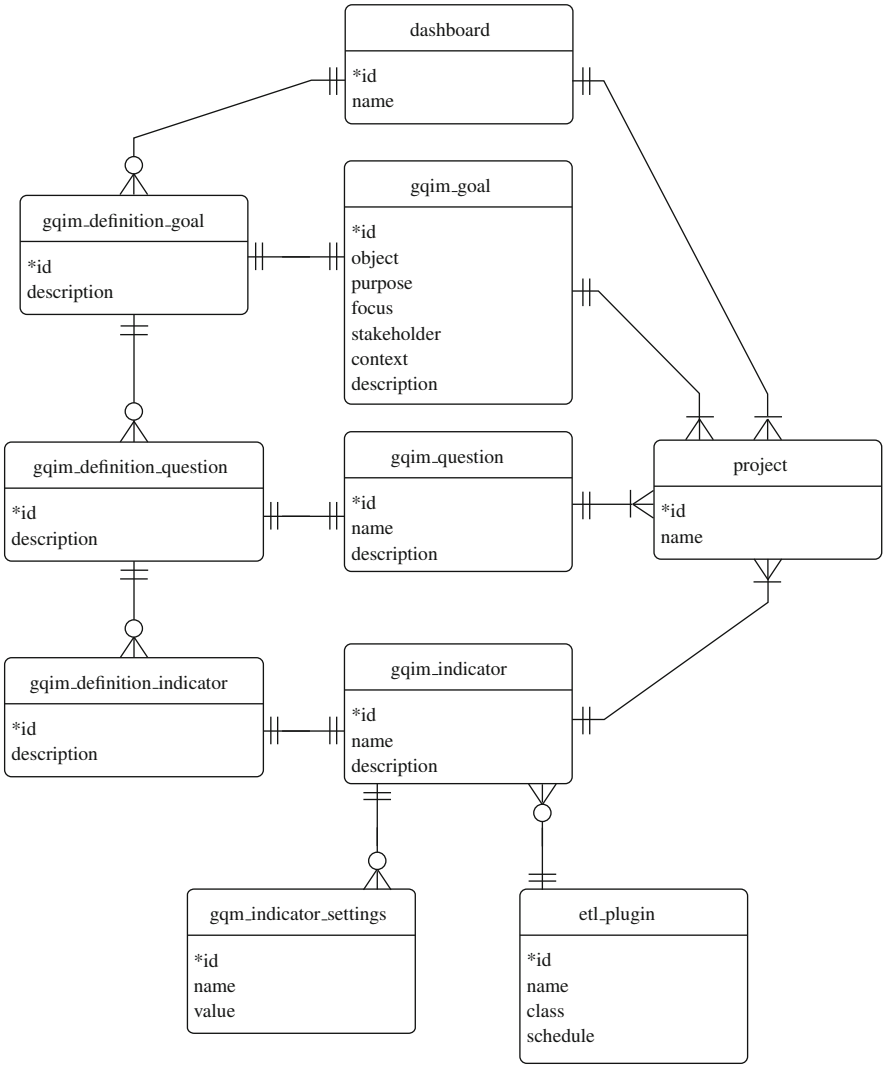


Fig. B.5 Database tables storing data about the defined GQM, the stored data in the experience factory, and visualizations

B.3 Physical View

The UML deployment diagram in Fig. B.6 summarizes how our system is deployed on the different machines.

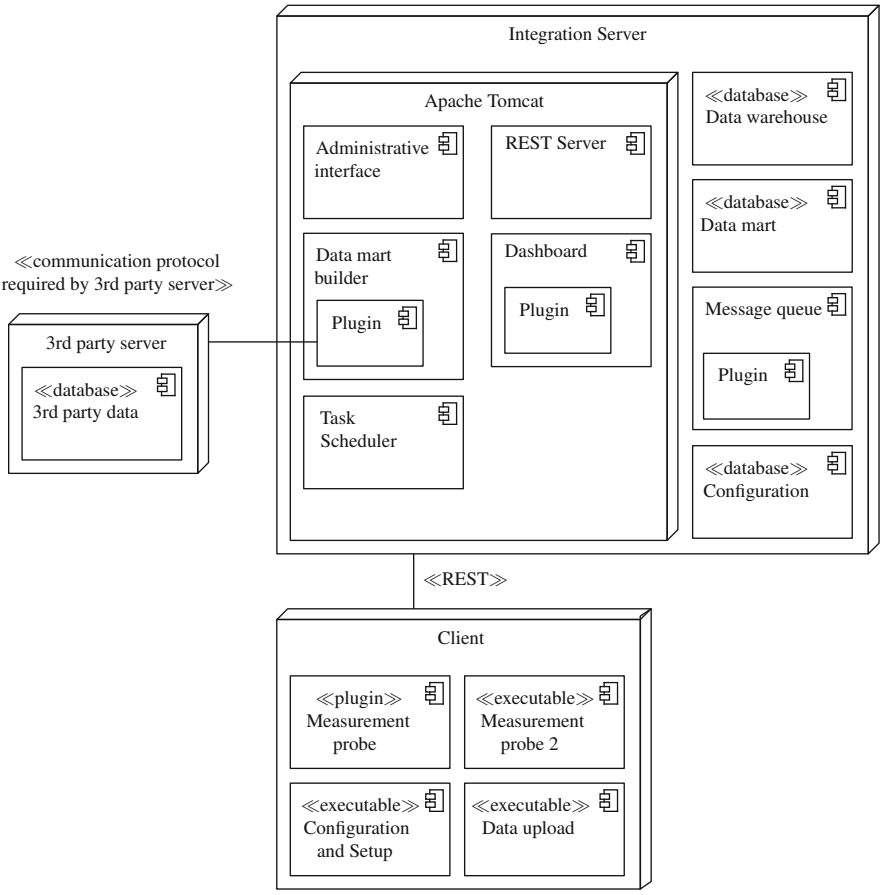


Fig. B.6 UML deployment diagram of the deployed components in our measurement system

The components on the "Integration server" are:

- the three databases: the data warehouse, the project data mart, and the configuration database,
- the message queue, together with the message plugins that handle the upload of the incoming data into the data warehouse,
- the administrative interface,
- a REST [11] service to interface with the server,

- the "Data mart builder" that updated the data mart on regular time intervals using the data mart plugins to extract parts of the data and stores it into the data mart. As we see in the figure, the data mart plugins could also interact with servers other than ours (what we call "3rd party servers") to extract data and store it into the data mart.
- the dashboard component, together with the visualization plugins that read data from the project data mart to visualize it, and
- a task scheduler that runs the different plugins at predefined time intervals.

On the client side, we install:

- the measurement probes,
- a tool to configure data collection properties, and
- the component that uploads the data to the server.

B.4 Process View

The processes that are executed by the here described architecture are:

- the collection of measurements, defined by the measurement probes,
- the upload of the measurements to the data warehouse, defined by the message queue plugins,
- the extraction of data to prepare it for a visualization, defined by the data mart plugins, and
- the creation of a visualization, defined by the visualization plugins.

The precise steps that each one of these plugins or probes executes depends on the specific data that has to be collected and visualized. The case studies in Chap. 11 illustrate some of the elaboration steps we performed through plugins.

B.5 Development View

We developed our measurement framework using Eclipse [9]. The hierarchical structure of our directories is organized as the components depicted in Fig. B.6.

Solutions

Problems of Chapter 1

1.1 Software development certainly requires creativity. It requires it to match the requirements of the client with the available technologies, to combine them into one solution, and to apply the result to the real world so that the problem is solved.

This process requires the ability to pick several technologies, combine them, imagine how a possible combination will work, which users will work with it, how they will work with it, on which hardware it will run, and many things more. This makes it similar to art, in which the artist combines different materials and skills to create what he has in his mind, but this is not different from craftsmanship. The difference between craftsmanship and art is that the result of craftsmanship has to have a function, while art does not.

The need to fulfill a function guides and limits the software developer in his choices. He cannot do whatever he wants to realize the idea he has. In fact, as the software developer writes software that does not have a function anymore but just represents an idea, he becomes indeed a software artist, for example, the piece of code below: it compiles but it does not do anything useful except to convey an idea. It is a poem taken from a collection of poems written in code, written in C++ by Daniel Bezerra [3]:

```
1 class love {};
2
3 void main()
4 {
5   throw love();
6 }
```

Listing A.1 "Unhandled love," a poem written in C++ by Daniel Bezerra

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9

All three, art, craftsmanship, and software development, can be learned to a certain extent, and for all three, talent certainly helps.

In summary, if we take "the need to fulfill a function" and "the presence of rules that we have to follow to succeed" as criteria to decide whether software development is an art or a craft, we think that it qualifies as a craft.

1.2 Venkatesh and Bala [28] studied "how and why employees make a decision about the adoption and use of information technologies." They see the intention to use a technology influenced by mainly two factors: the perceived usefulness and the perceived ease of use. This explains why we can observe that a software that is perceived as useful but not easy to use is still used. On the other hand, this does not mean that the customers are happy. As soon as they have an alternative that they perceive as more useful or more easy to use, they will switch.

To survive in a competitive market, it is therefore important to constantly adapt the solution to what the user needs. (In Chap. 3 we will point out that this is what "quality" actually means: the conformance to user requirements.)

Problems of Chapter 2

2.1 The roundabout is not the best analogy to Lean, but it has some aspects that are interesting to compare. Traffic lights increase safety and throughput. Image a crossing as in Fig. A.1. The traffic on all four roads has to give way to the traffic coming from the right. Therefore, everybody stops at the crossing to look if there is a car coming from the right side. Using traffic lamps, only a part of the traffic has to stop. Moreover, in a street without traffic lights, if all lanes are occupied, a "deadlock" can occur: everybody is waiting for the next car on the right side.

The disadvantage of the traffic lights is that they do not adapt to the traffic flow as good as the roundabout. Everybody that had to stop at a red traffic light on an empty street knows what we mean.

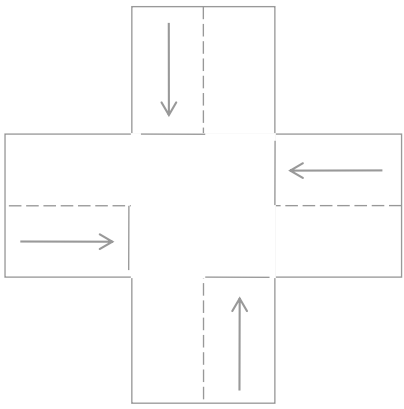


Fig. A.1 A street crossing without traffic lights

The roundabout is an interesting solution: with a simple trick, we changed the initial problem (which can become quite complex; see, e.g., Fig. A.2) into a much simpler problem.

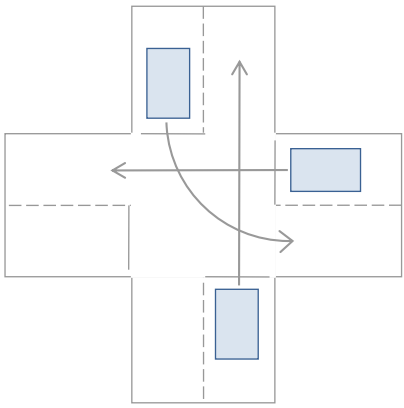


Fig. A.2 How do the three cars pass this crossing correctly?

Similar to this example, the ideas of Lean Thinking sometimes appear like simple tricks, but on the other hand, they reduce the complexity of the problem significantly.

2.2 The idea of Lean is to increase the surface of the rectangle. We want to increase our productivity by:

- organizing our work in such a way to reduce waste,
- increasing our knowledge about the development process, and
- increasing the value we provide to the customer.

This means that the resources that we have should remain the same; they should just be used in a more effective and efficient way.

Problems of Chapter 3

3.1 There is no unique, right way to classify the practices whether they contribute to identify what has value and what not, to increase knowledge, or to improvement. The practices frequently contribute to multiple aspects of the software development process.

Our proposal is listed in Table A.1

Table A.1 Classification of CMM practices whether they contribute to identify what has value and what not, to increase knowledge, or to improvement

Practice	Value	Knowledge	Improvement	Rationale
Causal Analysis and Resolution			×	Causal Analysis and Resolution aims to analyze past mistakes, to develop a solution, and to avoid the mistake in the future
Configuration Management	×			Configuration Management aims to organize work products so that all team members can work productively on them
Decision Analysis and Resolution		×		Decision Analysis and Resolution aims to collect all the information that is needed to avoid rework
Integrated Project Management	×			Integrated Project Management aims to ensure to provide value to all stakeholders
Measurement and Analysis		×		Measurement and Analysis aims to collect all necessary information to manage the process
Organizational Performance Management			×	Organizational Performance Management aims to select and deploy ways to improve the organizational performance
Organizational Process Definition	×			Organizational Process Definition aims to minimize the waste because of bad organization of the available resources
Organizational Process Focus			×	Organizational Process Focus aims to determine, plan, and implement process improvements
Organizational Process Performance		×		Organizational Process Performance aims to understand the performance of the process
Organizational Training		×		Organizational Training aims to provide the necessary knowledge and wisdom to all employees
Product Integration	×			Product Integration aims to ensure the final product which corresponds to the requirements
Project Monitoring and Control		×		Project Monitoring and Control
Project Planning	×			Project Planning aims to optimize the utilization of the resources to achieve the desired goal. The goal is to optimize the process to maximize the provided value
Process and Product Quality Assurance			×	Process and Product Quality Assurance
Quantitative Project Management		×		Quantitative Project Management aims to collect all the data that is necessary to manage the project
Requirements Development	×			Requirements Development aims to, based on the requirements of the client, determine, analyze, and validate the product requirements
Requirements Management	×			Requirements Management aims to identify what is important for the client and to align the output to the requirements

(continued)

Table A.1 (continued)

Practice	Value	Knowledge	Improvement	Rationale
Risk Management	×			Risk Management aims to prepare the organization to all eventualities and to maximize the expected value
Supplier Agreement Management	×			Supplier Agreement Management aims to coordinate with suppliers and agree on minimum quality standards to maximize the produced value
Technical Solution	×			Technical Solution aims to choose the right technologies and to design the right product to maximize the value for the client
Validation	×			Validation aims to make sure the product provides value to the client
Verification	×			Verification aims to make sure the product was built according to the requirements

3.2 The two cycles Define-Measure-Analyze-Improve-Control and Plan-Do-Study-Act are very similar. The first step "Define" and "Plan" are identical. The "Do" step of Plan-Do-Study-Act is missing in Define-Measure-Analyze-Improve-Control; it is implied. The two steps "Measure" and "Analyze" correspond to "Study"; the final steps "Improve" and "Control" correspond to "Act."

Problems of Chapter 4

4.1 One assumption of the traditional cost of change curve is that once the software is deployed, the costs to change something are the highest. These costs are the actual costs of changing the code and the costs to redeploy the new solution. The functionality to update over the Internet lowers these costs. Instead of sending to all clients an update in the form of a floppy disk or a CD, now the new version can be downloaded from the Internet. All major operating systems use this technology, as well as many applications. Some applications update automatically without asking (e.g., Google Chrome [12]). Also mobile operating systems frequently use this functionality, then called "Update over the air."

Not all costs can be avoided in this way: the costs of changing the code remain. Since the amount of source code and the complexity tend to grow during the development, changing the code later in the process has a higher probability to affect other parts of the system then changing the code earlier in the process.

Therefore, another technique to lower the costs of change is to use Component-Based Programming[27] in which we combine components, i.e., as a unit of composition with contractually specified interfaces and explicitly stated context dependencies only.

A technique to get rid of the update problem at all is Software as a Service. Since all clients interact with the software over a server, it is much easier to provide to all clients the update: we just have to update the server side.

4.2 The outcome of the change from a waterfall process to an Agile one depends on the willingness of all involved developers and clients to reevaluate their previous ways of working. If the developers just stop collecting all requirements upfront but are not willing to adopt the Agile practices to lower the cost of change curve, the costs **will** be as predicted by the traditional cost of change curve.

The problems we foresee are of practical nature: suddenly developers have to change methods that worked for many years. This will lower their productivity for a while, which will influence their perceived achievement and their motivation [13]. Therefore, we recommend to begin using Agile in a small project that is not critical so that developers can experiment and get familiar with the new practices and build up confidence in using them.

Also for the customer a switch to Agile can be frustrating. We have seen that many customers prefer to state all their requirements at the beginning of a project and then would like not to hear anything anymore from the programmers. Some others perceive it as annoying that they are asked for feedback regularly.

This shows that customers have to be involved in the decisions what "kind of Agility" is most valuable for them. Agile assumes that we can maximize the value for the customer getting frequent feedback on our work. Instead of investing time (and therefore money) giving us feedback, it might be more valuable for a customer to pay more for the product, in exchange for being less involved in the requirements gathering process. This would mean that it would be the responsibility of the software company to identify the requirements.

Not all software companies would accept such responsibility, but, for example, management consulting companies such as PriceWaterhouseCoopers, Ernst & Young, etc. would accept it. They would—for a different price than the software company—analyze the situation of the customer and carry out the implementation of an IT solution that is adequate to solve the problems of the customer [19].

Problems of Chapter 5

5.1 We expect the stakeholder influence, risk, uncertainty, and the cost of changes as in Fig. A.3 during an Agile project. The stakeholder influence remains high to allow the maximization of value. Therefore, the Agile team uses a development strategy that keeps the cost of changes low.

We update the typical cost and staffing levels across the Agile project life cycle as in Fig. A.4. We expect that the first two preparatory phases ("Starting the project"

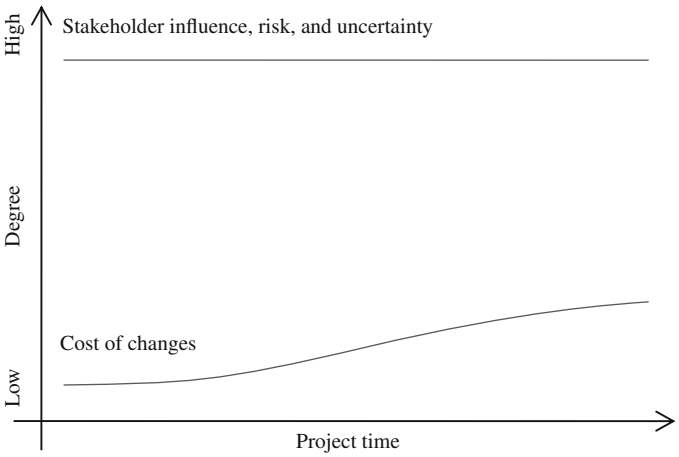


Fig. A.3 Stakeholder influence, risk, uncertainty, and the cost of changes during an Agile project

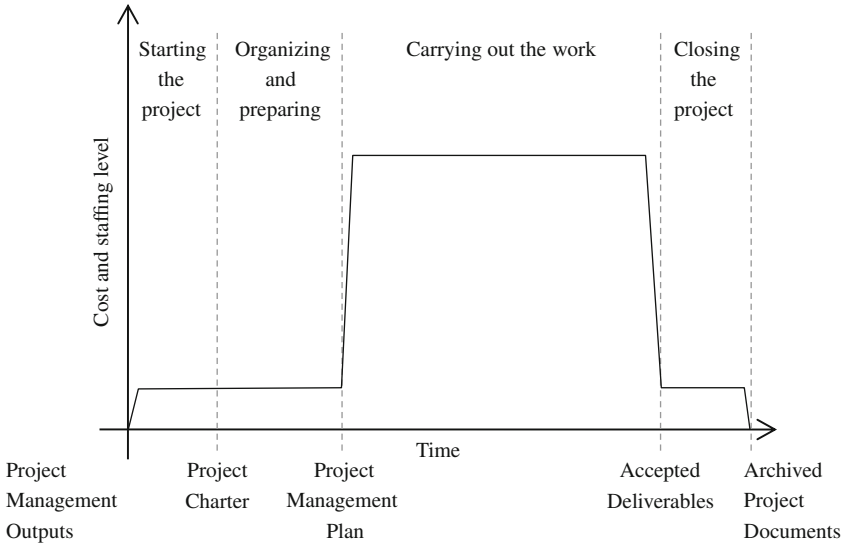


Fig. A.4 Typical cost and staffing levels across the Agile project life cycle

and "Organizing and preparing") are much shorter; therefore, we do not expect that the staffing level rises considerably during this phase. During the main phase ("Carrying out the work"), we do not expect that the team (typically) changes in size since in software engineering, Brooks' law applies, which states that "adding manpower to a late software project makes it later [5]." Moreover, we expect that Agile teams are small in comparison to other projects; this is why we draw the line less continuous as in Fig. 5.10.

Finally, we expect, as in the initial figure, that the closing phase of the project is handled only by a part of the team or assigned to a maintenance team.

5.2 The problem is not to follow the recommendations of the guru. His recommendations might indeed be very wise. The problem is that we do not know the rationale behind his recommendations. Practitioners should try out the recommendations in projects themselves and gain insight why certain mechanisms work and which need to be adapted to the context of the project, the organization, and the team.

In addition, we would recommend to look already now at specific issues that stop the team from being Agile. These issues are not mentioned in the books of the gurus; they might be very specific to the business in which the organization operates. It is therefore important to develop the skill to identify and to replace practices and technologies that work against Agility.

Problems of Chapter 6

6.1 Our proposal to classify the Lean software development practices suggested by Mary and Tom Poppendieck is listed in Table A.2.

Table A.2 Classification of the Lean software development practices suggested by Mary and Tom Poppendieck whether they contribute to identify what has value and what not, to increase knowledge, or to improvement

Practice	Value	Knowledge	Improvement	Rationale
Eliminate waste	×			Eliminate waste aims to maximize the value provided to the client
Build quality in			×	Build quality aims to change the software development process so that it is automatically enacted
Create knowledge		×		Create knowledge aims to collect knowledge about how to ensure the quality of the product
Defer commitment	×			Defer commitment aims to minimize rework
Deliver fast		×		Deliver fast aims to maximize the understanding of what is valuable or not
Respect people		×		Respect people aims to collect knowledge from all stakeholders
Optimize the whole			×	Optimize the whole aims to optimize to overall outcome, not single steps in the process

6.2 Martinsons et al. [22] implemented a balance scorecard for an IT organization. They chose the following perspectives:

- **Business value**: aims to understand the value that the business is producing.
- **User orientation**: aims to understand if the products and services provided by the organization fulfill the needs of the user.

- **Future readiness**: aims to understand if the organization is improving its products and services and preparing for future changes and challenges.
- **Internal process**: aims to understand if internal process is efficient and effective.

Possible goals for each perspective are:

- **Business value**:
  - ensure that the software projects provide value to the client and
  - minimize the software development costs.
- **User orientation**:
  - understand which technologies help the user to achieve his goals and
  - understand the goals of the user.
- **Future readiness**:
  - learn about new technologies that could be useful for the client and
  - adapt to new hardware and software trends of the market.
- **Internal process**:
  - minimize defects discovered after deployment and
  - maximize reuse.

Again, this is one of the possible solutions: the "right" one depends on the organization that will use it. It is important to understand the idea behind the Balanced Scorecard: to look at the organization from different perspectives, each perspective with its goals and measures.

Problems of Chapter 7

7.1 To evaluate the readability for the source code of an application, we can begin defining the measurement goal using the GQM goal template presented in Chap. 7:

- **Object of study**: the source code committed in our source code repository;
- **Purpose**: evaluate;
- **Focus**: readability;
- **Stakeholder**: programmer;
- **Context factors**: the Java programming language.

To evaluate this measurement goal, we could ask the following questions:

1. What is the readability for a specific file, package, class, method, or project?
2. Which files, packages, classes, or methods have the highest readability?
3. Which files, packages, classes, or methods have the lowest readability?
4. How does the readability develop over time in the code base?

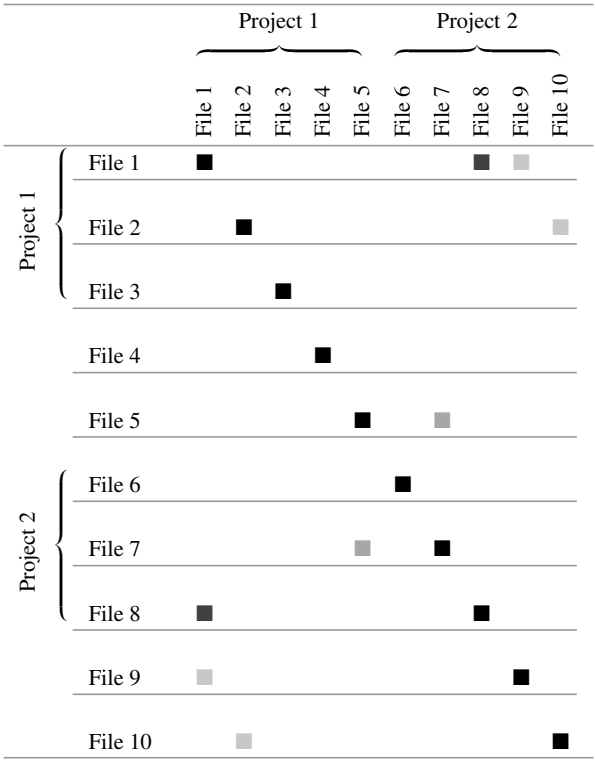
All these questions still need a defined way to evaluate the readability. We could calculate a readability index based on the following measurements, which we can calculate for each file, package, class, method, or project:

- the extensiveness of comments (the number of commented lines ÷ total number of lines),
- the extensiveness of blank lines (the number of blank lines ÷ total number of lines),
- the average length of the variable names, and
- the average length of the method names.

A remark: we once measured readability for a project using the measurements above. We then noticed that programmers used to comment out code that was not needed anymore, but they thought it might be useful in the future. Another reason was that the commented-out code was used in testing scenarios. While we think this is a bad practice, we could not ask the team to remove all the commented code; therefore, we had to find a way to distinguish commented text from commented code and ignore the commented code.

7.2 One graphical indicator could be a co-occurrence matrix as in Table A.3. The darker the marker, the higher the similarity between the two files.

Table A.3 Co-occurrence matrix example



According to this matrix, it seems that a large part of file 1 was copied into files 8 and 9 in the second project. Similarly, a part of file 2 was copied into file 10. Files 3 and 4 were not used in the second project. Parts of file 5 are present in file 7 of the second project.

This indicator can now be used to define a similarity index between two files or two projects.

Problems of Chapter 8

8.1 Some examples of wisdom relevant for Lean Software Development are the answers to the questions in Table A.4.

Table A.4 Examples of wisdom, relevant for Lean Software Development

	Organizational learning	Project learning	
Question	↓	↓	Rationale
It took our team 3 months to complete the last project. Why did it take that long?	×		It is important to know how long it takes for us to perform certain activities and why we chose to do those activities. In many projects a lot of time passes because we wait for input from the customer. It is important to find out why it takes the customer so long; we might be able to help him and gain time
We usually deploy a new system on New Year's Eve; in this way it is easier to migrate all the data from accounting. Why is this the best way?	×		On New Year's Eve, many organizations close their current accounting period and start a new one. Introducing a new system that interacts with accounting on this date saves the team from migrating a large part of the old data into the new system. On the other hand, this strategy limits the team in its decisions: a new system can only be installed at a specific date during the year. We need to verify if this is the best way to be valuable for the client, i.e., if it is cheaper to maintain this constraint, or if we should work on a sophisticated import function that allows the team to deploy a new system also during the year
We implemented feature X, which could be useful for other projects. How can we make a component out of it so that it can be reused?	×		If source code is likely to be reused in the future, we should package it to make it easy for others to use it
We are now working at project Y; which technologies can we use to avoid committing to a database technology?		×	This step looks for package experience that can be used to avoid committing on a technology early and then wasting effort if this decision reveals as wrong
We are currently facing many defects in project Z; how can we improve the defect rate?		×	This step tries to solve a problem in a specific project using previously collected experience

8.2 This question does not have one valid answer; some possible suggestions for the team are below. The answers we chose consider that the team might be skeptical about the introduction of a new software development methodology.

1. We have to get a clear picture of the pressure from the competition to innovate and lower the prices. There has to be a shared understanding about this and we have to evaluate means to achieve this goal. Lean in this case is not a hype but a viable method to study the own product costs and to decide how to reduce them.
2. We have to study the market and understand which strategy is more likely to succeed: upfront or iterative requirements gathering. With the high technical skills, we can look for solutions that allow to reuse a previous solution or to change the functionality of some implemented piece of code also later in the process.
3. We have to demonstrate that a defined software development process is followed only to the extent to improve flexibility and quality. Rules and regulations on how to code, which coding standards to follow, how to develop test cases, etc. have to be justified with the contribution they make to become more Lean.
4. The team has low planning skills. The team should decide whether it wants to take the opportunity of being involved in tailed software projects. For this purpose, it should keep the release cycles short so that the estimation errors do not have a big impact on the entire project.

Problems of Chapter 9

9.1 If I think Theory X is true, I will focus on behavior control (see Chap. 4). I want to make sure everybody is behaving correctly. For example, some measurement probes would be:

- monitor the activities of a developer and warn the management if the sequence of working steps is the recommended one;
- monitor the activities of a developer and warn the management if a test is written before the actual code; and so on.

If I think Theory Y is true, I will focus on measurement probes that focus on output and clan control. I have to make sure we produce the right result and that everybody knows what the customer expects from us. For example, some measurement probes would be:

- monitor the activities of a developer and warn the developer if some parts of the code have a too high complexity,
- monitor the activities of a developer and warn the developer if some source code has a too low testing coverage, etc.

9.2 We need to develop a measurement probe in batch mode:

- if the data we collect describes a given point in time (e.g., we check out the complete source code of a project on April 2nd) and
- if the data we collect describes a time interval and the system supports the extraction of the data we are interested in (e.g., we read the entries of a log file that describes events in time).

We need to develop a measurement probe in background mode if the data we collect describes a time interval and we cannot extract the data we are interested in after the fact. Then we need to develop a probe that logs what is happening as the events are taking place.

Problems of Chapter 10

10.1 These are some examples of data on which the different concepts focus:

- Activities: work steps, for which we know which resources every step used, which artifacts were modified, when it started, and when it ended;
- Resources: skilled human resources, equipment, services, supplies, commodities, material, budgets, or funds [26];
- Artifacts: source code, user documentation;
- Measurement: instruction on how to obtain nominal, ordinal, interval, and ratio scaled data;
- Non-invasive measurement: measurement probes, measurement schedules, instructions on where from to extract data, instructions on how to transform the data, instructions on where to submit the data;
- Data: nominal, ordinal, interval, and ratio scaled data;
- GQM$^+$Strategies model: organizational goals, measurement goals, questions, and measurements;
- Lean Thinking: principles;
- Knowledge and Wisdom: know-how and know-why (see Chap. 5);
- Information: know-what (see Chap. 5);
- Experience Factory: knowledge and wisdom; and
- Andon: visualizations.

10.2 There are several possibilities: one might be that your team members do not consider it useful. Some reasons for this could be (these points are based on the Technology Acceptance Model 3 [28]):

- the displayed data has no relevance for their job,
- the displayed data are hard to understand,
- the displayed data are hard to believe,
- it is not "cool" to use the displayed data; one cannot profile himself using it or it is considered bad to use it,
- our past experience does not suggest us to use the displayed data, and
- nobody forces them to use the displayed data.

Problems of Chapter 11

11.1 There exist different points of view of what researchers consider research. One possible classification is given by Petersen and Gencel [24]:

- **Positivist**: they believe that an objective reality exists. We can clarify a hypothesis based on objective observations and measurements.
- **Interpretivist**: they believe that there are different "realities" that we can observe. We can clarify a hypothesis based on observations and measurements in their context and understanding how its actors see and interpret the world.
- **Participatory**: they believe that we need to interact with what we are studying to understand it. We can clarify a hypothesis based on our interaction with the object of study.
- **Pragmatist**: they do not believe that we can create a model of the reality in our mind; therefore, we study it looking at what effects it has [14]. We clarify a hypothesis by identifying its practical consequences [14].

According to our experience, researchers that think that action research is unscientific are probably positivists.

11.2 A hypothesis that is difficult to be evaluated in a laboratory (using a controlled experiment) is, for example: "Anxiety is unrelated to drinking coffee." There are many factors that can influence the state of anxiety and it will be difficult to exclude them all in the laboratory. In this case it would be more promising to study the presence of anxiety in a group of people that consumed a moderate amount of caffeine over a longer period [21].

A hypothesis that can be evaluated in a laboratory is one where we have control over the factor of which we are studying the effects. An example of a hypothesis that can be evaluated in the laboratory is: "Bacterial growth is not affected by temperature." We are able to exclude other factors than the temperature of which we think that they might have an influence. Moreover, we are able to control the temperature as we want.

11.3 There is no logically correct way, the only way is to calculate the costs according to Direct Costing and to monitor the contribution margin.

11.4 If we assume that our measurement tool looks at the package imports at the beginning of a file, he can refer to the class directly (without package import), e.g., as in Listing A.2.

```
1 public void hiddenUseOfPlatformMethod() {
2   // I want to use the class "Fancy" from the package
3   // com.cool.platform, but do not want to be charged for it.
4   com.cool.platform.Fancy c = new com.cool.platform.Fancy();
5   c.serveYourself();
6 }
```

Listing A.2 How to cheat the cost accounting system

Since our tool only parses the import statements at the beginning of Java source code files, this use of a "platform" class is not detected.

This example shows that measurement should not be used to control people (see Sect. 9.5) but to gather knowledge that can be used to improve.

Problems of Chapter 12

12.1 To set up such a system, we need two measurement probes: one that connects to issue tracking system (e.g., Bugzilla [6]) and one that connects to the automatic build and test system (e.g., Jenkins [16]).

Both systems, Bugzilla and Jenkins, offer a REST [11] application programming interface. In this case we could collect the data just to visualize it, for example, we saw a system like this that just displayed the collected numbers on a screen attached on the wall as in Fig. A.5.

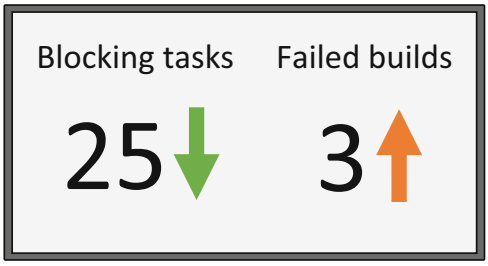


Fig. A.5 Monitor displaying measurements

The data could be collected regularly by a script or a Java application on a server and visualized on a website on the monitor.

A more sophisticated system would save all the collected data to find the system that fails the most or collects the causes why systems fail or why blocking tasks are reported.

References

1. Apache Software Foundation: Apache cassandra (2013). Online: http://cassandra.apache.org. Accessed 4 Dec 2013
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Longman, Boston (2003)
3. Bertran, I.: code {poems}. In: Allen, J., Boshears, P. (eds.) Continent., 2.2. Continent (2012). Online: http://www.continentcontinent.com/index.php/continent/index. Accessed 4 Dec 2013
4. Bradner, S.: Key words for use in rfcs to indicate requirement levels (1997). Online: http://www.ietf.org/rfc/rfc2119.txt. Accessed 4 Dec 2013
5. Brooks, F.P., Jr.: The Mythical Man-Month (Anniversary edn.). Addison-Wesley Longman, Boston (1995)

6. Bugzilla Contributors: Bugzilla (2013). Online: http://www.bugzilla.org,. Accessed 4 Dec 2013
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
8. Crockford, D.: Json: the fat-free alternative to xml (2006). Online: http://www.json.org/fatfree.html. Accessed 4 Dec 2013
9. Eclipse Foundation: Eclipse ide (2013). Online: http://www.eclipse.org. Accessed 4 Dec 2013
10. Ferraiolo, D., Kuhn, R.: Role-based access control. In: Proceedings of the NIST-National Computer Security Conference (NCSC). National Institute of Standards and Technology, Baltimore (1992)
11. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. **2**(2), 115–150 (2002)
12. Google: Google chrome (2013). Online: http://www.google.it/chrome/browser/. Accessed 25 May 2014
13. Herzberg, F.: One more time: How do you motivate employees? Harv. Bus. Rev. **46**(1), 53–62 (1968)
14. Hookway, C.: Pragmatism. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy, Spring 2010 edn. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Stanford (2010). Online: http://plato.stanford.edu/archives/spr2010/entries/pragmatism. Accessed 4 Dec 2013
15. Janes, A., Piatov, D., Sillitti, A., Succi, G.: How to calculate software metrics for multiple languages using open source parsers. In: Petrinja, E., Succi, G., Ioini, N., Sillitti, A. (eds.) Proceedings of the International Conference on Open Source Software: Quality Verification (OSS). IFIP Advances in Information and Communication Technology, vol. 404. Springer, Koper (2013)
16. Jenkins CI Contributors: Jenkins ci (2013). Online: http://jenkins-ci.org. Accessed 4 Dec 2013
17. Kimball, R.: The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. Wiley, New York (1996)
18. Kruchten, P.: The 4 + 1 view model of architecture. IEEE Softw. **12**(6), 42–50 (1995)
19. Lauesen, S.: Software Requirements: Styles and Techniques. Addison-Wesley, Harlow (2002)
20. Leach, P.J., Mealling, M., Salz, R.: Rfc 4122: a universally unique identifier (uuid) urn namespace (2005). Online: http://www.ietf.org/rfc/rfc4122.txt. Accessed 4 Dec 2013
21. Lee, M.A., Cameron, O.G., Greden, J.F.: Anxiety and caffeine consumption in people with anxiety disorders. Psychiatry Res. **15**(3), 211–217 (1985)
22. Martinsons, M., Davison, R., Tse, D.: The balanced scorecard: a foundation for the strategic management of information systems. Decis. Support Syst. **25**(1), 71–88 (1999)
23. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
24. Petersen, K., Gencel, C.: Worldviews, research methods, and their relationship to validity in empirical software engineering research. In: Proceedings of the Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-Mensura). IEEE, Ankara (2013)
25. Piatov, D., Janes, A., Sillitti, A., Succi, G.: Using the eclipse c/c++ development tooling as a robust, fully functional, actively maintained, open source c++ parser. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W. (eds.) Proceedings of the International Conference on Open Source Systems: Long-Term Sustainability (OSS). IFIP Advances in Information and Communication Technology, vol. 378. Springer, Hammamet (2012)
26. Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 5th edn. Project Management Institute, Newtown Square (2013)
27. Szyperski, C.: Component Software: Beyond Object Oriented Programming. Addison-Wesley Professional, Reading (2002)
28. Venkatesh, V., Bala, H.: Technology acceptance model 3 and a research agenda on interventions. Decis. Sci. **39**(2), 273–315 (2008)
29. Westfall, L.: The Certified Software Quality Engineer Handbook. ASQ Quality Press, Milwaukee (2008)

Index

A. Janes, G. Succi, *Lean Software Development in Action*,
DOI 10.1007/978-3-642-00503-9

T

V

W