Software Configuration Management Strategies and IBM® Rational® ClearCase® Second Edit

By David E. Bellagio, Tom J. Milligan

Publisher: **Addison Wesley Professional**
Pub Date: **May 23, 2005**
Print ISBN: **0-321-20019-5**
Pages: **384**

Overview

"I w rote that the first edition of this book "communicates much of the experience, w isdom, and insight that w as acquired along the w ay to discoveri practices are and how to implement/deploy them." The second edition is full of even more practical experience! It not only refines and expands upon best practices, it also contains even more concrete "how -to" information about implementing and deploying them."Brad Appleton co-author of Softw Management Patterns: Effective Teamw ork, Practical Integration"Read this book w hen you're getting started w ith configuration management (CM); r and once more tw o years later. Softw are Configuration Management Strategies and IBM Rational ClearCase goes beyond the basics to provide a c strategies for projects of all sizes and complexities."Jennie Brow n, CM Specialist IBM Corporation"This book covers practical softw are configuratio ClearCase UCM and ClearCase use for project teams. Every project lead and SCM specialist should (re)read this book before starting a project."Rog Softw are Engineer IBM Rational Softw are"This second edition captures vital Unified Change Management (UCM) features and concepts. It brings th full circle and lays the ground w ork for a successful UCM implementation; Bellagio and Milligan continue w here the first edition left off. The inclusion concepts, single stream development, performance tuning, and other features and concepts make this the core of any old or new Unified Change M implementation.It is rare that a second edition of a book can hold your interest like the first. This publication does just that...this continues to be the s interested in Unified Change Management deployment."Adam Levensohn, Manager IBM Rational Softw areSoftw are Configuration Management (SCM largest, most complex project teams manage change, so they can deliver higher quality products faster. The First Edition of Softw are Configuration and Rational ClearCase established itself as the definitive single source for guidance on both SCM best practices and the market's leading product, I This fully updated Second Edition systematically addresses the latest ClearCase and ClearQuest® innovations, w hile offering even deeper insight in management.The authors each draw on more than 15 years of SCM experience, and the know ledge of w orking w ith IBM Rational field teams in cus w orldw ide. They systematically cover SCM planning and deployment, and SCM's use throughout the entire project lifecycle: development, integratio release deployment, and beyond. They offer practical guidance on addressing challenges that arise as projects grow in size and complexity, from distributed teams to tracking change requests.Coverage includes Understanding basic SCM concepts, and the role, value, and components of SCM Rational's Unified Change Management (UCM) modelincluding today's most effective usage models, strategies, and policy configurations Creating C establishing UCM environments, step-by-step Working as a project manager in the ClearCase environment Leveraging new ClearCase MultiSite and to full advantage Using the new ClearCase Remote Client to access centralized repositories across WANs Monitoring ClearCase and tuning it for m book is valuable for everyone concerned w ith SCM: developers w ho w antSCM to be as intuitive and convenient as possible; project managers and must efficiently manage change; tools engineers; even IT managersevaluating SCM technologies. © Copyright Pearson Education. All rights reserve

Software Configuration Management Strategies and IBM® Rational® ClearCase® Second Edition A Practical Introdu

By David E. Bellagio, Tom J. Milligan

....
Publisher: **Addison Wesley Professional**
Pub Date: **May 23, 2005**
Print ISBN: **0-321-20019-5**
Pages: **384**

Table of Contents | Index

Team LiB

PREVIOUS NEXT

Copyright

IBM Press Program Manager: Tara Woodman, Ellice Uffer

IBM Press Consulting Editor: David West

Cover design: IBM Corporation

Published by Pearson plc

Publishing as IBM Press

Library of Congress Number: 200523397

Praise for *Software Configuration Management Strategies and IBM® Rational® ClearCase®, Second Edition: A Practical Introduction*

"I wrote that the first edition of this book "communicates much of the experience, wisdom, and insight that was acquired along the way to discovering what [SCM] best practices are and how to implement/deploy them." The second edition is full of even more practical experience! It not only refines and expands upon earlier strategies and best practices, it also contains even more concrete "how-to" information about implementing and deploying them."

Brad Appleton co-author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*

"Read this book when you're getting started with configuration management (CM); read it again in six months and once more two years later. *Software Configuration Management Strategies and IBM Rational ClearCase* goes beyond the basics to provide a coherent review of CM strategies for projects of all sizes and complexities."

Jennie Brown, CM Specialist IBM Corporation

"This book covers practical software configuration management (SCM), ClearCase UCM and ClearCase use for project teams. Every project lead and SCM specialist should (re)read this book before starting a project."

Roger Jarrett, Senior Software Engineer IBM Rational Software

"This second edition captures vital Unified Change Management (UCM) features and concepts. It brings the entire UCM feature set full circle and lays the ground work for a successful UCM implementation; Bellagio and Milligan continue where the first edition left off. The inclusion of composite baseline concepts, single stream development, performance tuning, and other features and concepts make this the core of any old or new Unified Change Management implementation.

It is rare that a second edition of a book can hold your interest like the first. This publication does just that…this continues to be the standard for users interested in Unified Change Management deployment."

Adam Levensohn, Manager IBM Rational Software

IBM Press

WebSphere Books

On Demand Computing Books

Business Intelligence for the Enterprise

Biere

On Demand Computing

Fellenstein

Grid Computing

Joseph and Fellenstein

Autonomic Computing

Murch

More Books from IBM Press

DB2® Books

Acknowledgments

The authors of this second edition would like to first thank Brian White, who did the heavy lifting involved with creating the first edition of this book and who was instrumental in providing the opportunity for us to contribute to this second edition. Thanks also go to the host of co-workers and customers who over the years have contributed to our greater understanding of the principles of SCM, the functioning of the products, and what SCM strategies work and which ones don't. Most notable among these are Ralph Capasso, Stef Schurman, Ryan Sappenfield, Rob Budas, Harry Abadi, Peter Hack, and George Moberly. Our long-suffering and ever-patient editors at Addison-Wesley, Mary O'Brien and Chris Zahn, deserve many thanks as well. We would also like to thank our families for their long sufferings during this process.

Those individuals who were instrumental in our first exposure and opportunities with ClearCase also deserve many thanks: Scott Elmenhurst, Denis LeBlanc, John Leary, David Crawford, Marsha Shehan, and Doug Fierro.

Much of the material new to this edition comes directly from the experience of the Rational field teams in engagement with customers using ClearCase and UCM. Some of those folks we would like to thank for their efforts in fleshing out strategies that work with many of our larger customer needs are Samit Mehta, Mike Nellis, Ana Giordano, Kartik Kanakasabesan, Daniel Diebolt, Jennie Brown, and Bryan Miller.

Special thanks to these people who put in the time to review this book and provide comments to help make the reading experience more enjoyable: Adam Levensohn, Brad Appleton, Darryl Hahn, Dennis Brown, Jennie Brown, Jim Tykal, and Roger Jarrett.

Finally, our thanks go out to everyone at IBM, Rational Software who keep ClearCase and UCM efforts moving forward. Keep up the good work.

About the Authors

David E. Bellagio

Tom Milligan

David E. Bellagio

David has been involved in software development for the last 25 years, ever since he got addicted to it in high school. After realizing that he would not be a professional baseball player, he went on to receive his Bachelor of Science and Master of Science degrees in computer science, with honors, from Chico State University, California. David worked previously at CSC, Tandem Computers, ADP, and HP. He first began using ClearCase in 1994, and he spent the next four years deploying ClearCase to many developers while employed at ADP and HP. In 1998, he joined Rational Software as a technical field representative in the Pacific Northwest.

David currently is a Worldwide Community of Practice Leader for Enterprise Change Management at IBM, Rational Software. He works with customers and IBM teams to ensure successful deployment and adoption of Rational SCM solutions. David works with Rational field teams delivering workshops, seminars, and assessments to help improve software process and customer results. He also works onsite with customers around the world to define and manage successful deployment of Rational Software solutions.

David has presented these topics at Rational User Conferences:

1995:	"Building Software with Clearmake on Non-ClearCase Hosts"
1996:	"ClearAdminA Set of Scripts and Processes for Administrating ClearCase Sites"
2004:	"UCM Stream Strategies and Best Practices"

David has also led the UCM Hands on Workshop sessions at the 2003 and 2004 Rational User Conferences.

David currently resides in Washington with his wonderful wife and three children. When time allows, he also enjoys brewing fine ales and mead. He can be reached via e-mail at dbellagio@us.ibm.com.

Tom Milligan

Tom Milligan is a Senior SCM Technical Marketing Engineer on IBM Rational Software's Worldwide Technical Marketing team. Prior to this assignment he worked in Rational Software's Western Region Consulting Organization.

Prior to joining Atria Software in 1995, Tom worked in Electronic Design Automation, Software QA, software tools acquisition and development as well as real-time embedded software development. Tom holds a B.A (Honors College) in Computer Science from the University of Oregon (1978).

Tom has spoken at the Rational User Conference several times:

1997:	"Integrating ClearCase NT with Third-Party Applications" (specifically, integrating ClearCase with Microsoft Word)
1999:	"Integrating Requisite Pro and DDTs"
2001:	"Using Perl with the ClearCase Automation Library (CAL)"
2003:	"ClearCase Performance Analysis, Monitoring and Tuning"
2004:	"Fundamentals of Good Software Configuration Management, ClearCase Performance Analysis, Monitoring and Tuning"

Tom has also published the following articles:

The Rational Edge

November 2001:	"Using Perl with Rational ClearCase Automation Library (CAL)"
September 2002:	"Using Perl with the Rational ClearQuest API"
July 2003:	"ClearCase Performance Analysis, Monitoring and Tuning (part 1)"
September 2003:	"ClearCase Performance Analysis, Monitoring and Tuning (part 2)"

Websphere Advisor

July/August 2003: "7 Attributes of Highly Effective SCM Systems"

Rational Developer Network

"ClearCase Cheat Sheet"

Tom was also keynote speaker at the 2003 Association of Configuration and Data Management (ACDM) Conference.

Tom resides on the central coast of California with his most excellent wife and three children. In his spare time Tom enjoys playing Ultimate (also known as Ultimate Frisbee), watching his kids play volleyball and soccer, and staying up late with his telescope. He can be reached via e-mail at tmilligan@us.ibm.com.

Preface to the Second Edition

Welcome to the second edition of *Software Configuration Management Strategies and IBM Rational ClearCase!* We have updated the first edition to enhance the strategies material and to reflect developments in the evolution of IBM Rational ClearCase.

What This Book Is About

This book is about the engineering discipline of software configuration management (SCM) and how the widely used SCM product ClearCase automates and supports SCM best practices through a model called Unified Change Management (UCM). This book covers basic SCM concepts, typical SCM problems encountered as projects and software systems grow in size and complexity, and how you can apply SCM tools and processes to solve these problems. Advanced SCM topics are also discussed, including managing large geographically distributed teams and combining the disciplines of SCM and change request management (or defect tracking).

Much material in the first edition of this book discussed the issues that arise in a software-development project and how an SCM tool should be able to address those issues. That first edition material has been left intact and, in some cases, expanded simply because it represents fundamental truths about software development that have not changed and that are unlikely to change as long as humans are involved in the process.

The second edition of this book expands on the first edition by adding those features introduced to ClearCase since the first editionspecifically, the ClearCase Remote Client, UCM-enforced single-stream projects, full support for unlimited parent/child stream hierarchies, composite baselines, and expanded and more flexible UCM policy configurations. Furthermore, we have added more discussion of ClearQuest and the new ClearQuest MultiSite product. UCM has evolved in many ways since the first edition was published, and many of the additions to this edition stem from our experiences in helping customers adopt and achieve success with ClearCase and UCM in their environment.

Beyond the functional enhancements to ClearCase and ClearQuest, we have included a discussion on monitoring and tuning ClearCase performance, as well as usage models for UCM that we have seen being successfully practiced since the first edition was published. We believe this expanded information will provide significant assistance in helping the reader get the most out of the UCM environment and understand the range of UCM development models that are supported and known to work.

This book is based on the experience gained by the authors working with some incredible people in the SCM field over the last 15 years. After reading it, you should have a better understanding of software configuration management, a better idea of the software development problems solved by using SCM tools and techniques, and a clear understanding of how you can use ClearCase to solve these problems and meet your SCM requirements. The authors sincerely hope you enjoy the book and find it valuable.

What You Need to Know Before Reading This Book

The key to your success is understanding SCM, the requirements for your software project, and how to apply an SCM tool to meet a project's requirements. This book will get you started if you are new to software configuration management. However, you will get the most out of this book if you already have some SCM experience and have used basic version control tools before. This book assumes that you are familiar with the software-development process. It will also be helpful if you have a specific development project in mind while you are reading.

Who You Are and Why You Should Read This Book

This book is not about the nitty-gritty details of writing ClearCase triggers and scripting home-grown integrations with legacy tools; instead, it will give you a high-level view of some common SCM scenarios and how ClearCase can be applied. If you are new to SCM or ClearCase, read this book cover to cover. If you have used ClearCase or have a strong foundation in SCM, look through the table of contents and pick chapters and sections that are of particular interest to you.

For a Software Engineer

The biggest thing an SCM tool can do for a software engineer is to stay out of the way. SCM should perform its function, yet be as transparent as possible. The SCM tool and how it is applied should maximize your ability to make changes to the software. Poor tools or poorly designed processes can add unnecessary time and effort to your work. This book can help you identify the areas in your SCM tools and processes to streamline. It discusses some new advances in the SCM area specifically designed for streamlining development. One of these is the notion of activity-based software configuration management. The idea here is to raise the level of abstraction from files to activities. This makes working with an SCM tool, tracking your changes, and sharing changes with other software engineers more intuitive.

If you're new to SCM, read Chapter 1, "What Is Software Configuration Management?" For an overview of the objects managed by ClearCase, see Chapter 4, "A Functional Overview of ClearCase Objects." To gain an understanding of how ClearCase is used on a daily basis from a development perspective, see Chapter 8, "Development Using the ClearCase UCM Model."

For a Software Project Manager or Technical Leader

As a leader for a software project, you are concerned with deciding what changes to make to a software system and then ensuring that those changes happen. Unplanned changes, made by well-meaning developers, introduce risk into the project schedule and can cause schedule delays and poor product quality. The capability to control and track change is essential to your project's success.

This book should help you gain a solid understanding of SCM, see why you need it, and learn how ClearCase can be used to solve problems you might encounter on projects. Specifically, see Chapter 6, "Project Management in ClearCase," and Chapter 7, "Managing and Organizing Your ClearCase Projects." If you are managing teams that are not all in one location, see Chapter 11, "Geographically Distributed Development," for a discussion of the issues and strategies involved.

For a Tools Engineer

The role of the tools engineer is often overlooked but is essential to success, particularly in large organizations. Your job is to figure out how to apply a given tool to the people, processes, and organization for which you work. This book gives you information about SCM and ClearCase that you can use to determine the best way to apply ClearCase to projects.

For Those Evaluating ClearCase

This book is a good starting point in evaluating ClearCase because it presents a number of common software-development scenarios, as well as more complex scenarios such as geographically distributed development. It discusses the requirements of SCM processes and tools in terms of a set of SCM best practices and shows how to apply ClearCase to support them. Included are overviews of the ClearCase out-of-the-box process Unified Change Management and ClearCase objects.

How the Book Is Laid Out

Here is a brief summary of all the chapters.

- Chapter 1, "What Is Software Configuration Management?," provides a general introduction to software configuration management and the key best practices behind it. It answers these questions: What is software configuration management? What are SCM tools? What is the SCM process?
- Chapter 2, "Growing into Your SCM Solution," discusses the growing complexity of software-development projects and proposes that as projects grow in complexity, so does their need for richer SCM support. It covers the history of SCM tool evolution using five categories of software projects, ranging from software developed by a single individual to projects with many geographically distributed project teams.
- Chapter 3, "An Overview of the Unified Change Management Model," provides an overview of the ClearCase out-of-the-box usage model, Unified Change Management, which automates and supports a particular SCM process. The material is discussed in terms of the roles and responsibilities of the various team members, such as the architect, project manager, developer, and integrator.
- Chapter 4, "A Functional Overview of ClearCase Objects," provides a high-level description of ClearCase objects and concepts. This chapter serves as a bridge between general SCM terminology and ClearCase-specific terminology.
- Chapter 5, "Establishing the Initial SCM Environment," provides information on how to configure an initial SCM environment. It discusses the basics of ClearCase architecture and how to approach performance tuning and monitoring. The chapter also covers mapping the software architecture to the physical components in the SCM tool, and briefly discusses creating the SCM repositories and importing existing software.
- Chapter 6, "Project Management in ClearCase," focuses on the role of the project manager with respect to SCM. Particular attention is paid to automation and functionality in ClearCase that specifically support the project manager, including UCM-enforced configurable project policies. It also presents an example of creating a ClearCase project.
- Chapter 7, "Managing and Organizing Your ClearCase Projects," discusses the issues of coordinating parallel work. It also covers scenarios that involve multiple teams cooperating on a common release, development of multiple releases in parallel with multiple teams, coordination of IS/IT-style projects, and coordination of documentation-oriented projects.
- Chapter 8, "Development Using the ClearCase UCM Model," provides an introduction to using ClearCase, specifically focusing on the role of the developer. It shows you how to find and join an existing project, how to make changes to files to accomplish an activity, how to deliver the changes associated with the activity, and how to update the development workspace with changes made by other developers on the project.
- Chapter 9, "Integration," focuses on the role of the integrator and discusses several approaches to both intraproject and cross-project software integration using ClearCase and ClearQuest.
- Chapter 10, "Building, Baselining, and Release Deployment," covers baselining and how baselines and composite baselines are manipulated and used in defining and automating build and release systems.
- Chapter 11, "Geographically Distributed Development," discusses the organizational,

Conventions Used

The conventions used in this book fall into two categories: those having to do with presentation of the text and those dealing with UML diagrams in figures.

Commands and Notes, Warnings, and Tips

Command-line operations are called out with a monospaced font and prompt, as in this example:

```
prompt> command -flag1 -flag2
```

Long commands are written on multiple lines, for clarity (as shown here), but should be typed on one line. A code-continuation character (`>>`) is inserted in the line to indicate that it is all one line, as in this example:

```
prompt> longcommand longobject-identifier
>> -flag1 //machine/pathname
>> -flag
```

Notes, warnings, and tips appear in the text as follows:

Note

Particular points that need to be emphasized appear in the text in this font with an arrow to alert you.

Warning

The screened warning box is used to emphasize an issue or concern that might be encountered and should be avoided.

ClearCase Pro Tip

A screened box labeled with this denotes information that is specifically useful for people who are already using ClearCase. If you have not used ClearCase, you can skip the tips.

UML Diagram Format

This book includes diagrams that use a graphical modeling language called the Unified Modeling Language, or UML. For more information on UML, see *The Unified Modeling Language User Guide*, by Grady Booch, James Rumbaugh, and Ivar Jacobson [Booch, 1999].

Here is a description of the small subset of UML used in this book: An object is shown as a box, with text that describes the object. Lines represent associations between the objects, with text that describes the association. For example, "a house has a roof:"

Chapter 1. What Is Software Configuration Management?

The title of this chapter asks such a simple question, the answer to which, one would think, ought to be known by anyone with any kind of record in software development. In reality, it seems that few are able to actually articulate what is meant by that term *software configuration management*.

To be fair, those who have a record in software development recognize that there is a need to control what is happening in the development process, and, once controlled, there is a sense that the process can be measured and directed. From that recognized need, then, comes a good working definition of software configuration management:

Software Configuration Management is how you control the evolution of a software project.

Slightly more formally, *software configuration management* (SCM) is a software-engineering discipline comprising the tools and techniques (processes or methodology) that a com pany uses to manage change to its software assets. The introduction to the IEEE "Standard for Software Configuration Management Plans" [IEEE 828-1998] says this about SCM:

SCM constitutes good engineering practice for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for identifying and controlling documentation, code, interfaces, and databases to support all life-cycle phases
- Supporting a chosen development/maintenance methodology that fits the requirements, standards, policies, organization, and management philosophy
- Producing management and product information concerning the status of baselines, change control, tests, releases, audits, etc.

Clearly, software is easy to changetoo easy. And not only is it easy to change, but it is unconstrained by the physical laws that serve as the guardrails of what is possible with hardware systems. Software is bounded only by the limits of the human imagination. Uncontrolled and undirected, imagination can quickly give rise to nightmare.

Today most software project teams understand the need for SCM to manage change to their software systems. However, even with the best of intentions, software projects continue to fail because of problems that could have been avoided through the use of an SCM tool and appropriate processes. These fail ures are reflected in poor quality, late delivery, cost overruns, and the incapability to meet customer demands.

To understand software configuration management, you might find it easier to look first at configuration management in a hardware-development environment. Hardware systems have physical characteristics that make the problems caused by the lack of sound configuration management easier to see.

For example, consider a personal computer. A computer has a processor, a mainboard, some memory, a hard drive, a monitor, and a keyboard. Each of these hardware items has an interface that connects it to other hardware items. Your mouse has a plug, and your computer has a port into which you plug your mouse, and, voilá, everything works.

If the plug on the mouse was not compatible with the port on the computer, there would be no way to connect the two pieces of hardware into a working system. Throughout the

1.1. SCM Best Practices

When implementing SCM tools and processes, you must define what practices and policies to employ to avoid common configuration problems and maximize team productivity. Many years of practical experience have shown that the following best practices are essential to successful software development:

- Identify and store artifacts in a secure repository.
- Control and audit changes to artifacts.
- Organize versioned artifacts into versioned components.
- Organize versioned components and subsystems into versioned subsystems.
- Create baselines at project milestones.
- Record and track requests for change.
- Organize and integrate consistent sets of versions using activities.
- Maintain stable and consistent workspaces.
- Support concurrent changes to artifacts and components.
- Integrate early and often.
- Ensure reproducibility of software builds.

The rest of this section explains each of these best practices.

1.1.1. Identify and Store Artifacts in a Secure Repository

To do configuration management, you must identify which artifacts should be placed under version control. These artifacts should include both those used to manage and design a system (such as project plans and design models) and those that instantiate the system design itself (such as source files, libraries, and executables and the mechanisms used to build them). IEEE calls this *configuration identification*: "an element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation" [IEEE Glossary, 1990].

In terms of an SCM tool, identification means being able to find and identify any project or system artifact quickly and easily. Anyone who has managed a development project with no SCM or poor SCM can attest to the difficulty of finding the "right" version of the "right" file when copies are floating around all over the place. Ultimately, losing or misidentifying artifact versions can lead to the failure of a project, either by hindering delivery of the system because of missing parts or by lowering the quality of the system because of incorrect parts.

Organizing artifacts and being able to locate them are not enough. You also need fault-tolerant, scalable, distributable, and replicable repositories for these critical assets. The repository is a potential central point of failure for all your assets; therefore, it must be fault-tolerant and reliable. As your organization grows, you will add data and repositories, so scalability and distributability are required to maintain high system performance.

Another means of growth in today's software market is through acquisition, which affects many companies by resulting in the geographical distribution of development groups. The SCM tool, therefore, must be capable of supporting teams that collaborate across these geographically distributed sites.

1.2. SCM Tools and SCM Process

SCM best practices are achieved by applying both processes and tools to a software-development project. This section briefly introduces both.

1.2.1. SCM Tools

SCM tools are software tools that automate and facilitate the application of the SCM best practices. As with a compiler, debugger, and editor, an SCM tool is an essential part of every software engineer's tool kit today.

It is unrealistic to try to maintain effective SCM without an SCM tool. In early SCM project environments, one or more individuals acted as the CM librarians. These librarians handed out pieces of software for people to work on, diligently recorded who had which pieces, and logged in new versions as people turned in changes. This approach is not competitive today because it is too slow, is too prone to error, and does not scale.

The goal of successful SCM is to allow as much change as possible while still maintaining control of the software. SCM tools help automate tedious, manual, and error-prone pieces of the SCM process, and can ensure that your project can support all of the SCM best practices.

1.2.2. SCM Process

A process defines the steps by which you perform a specific task or set of tasks. An SCM process is the way SCM is performed on your projectspecifically, how an SCM tool is applied to accomplish a set of tasks.

A key mistake most people make is to assume that an SCM tool will, in and of itself, solve their SCM problems or support their SCM requirements. This is wrong! The picture will not hang itself if you buy a hammer and nails. It is not the tool itself that solves a problem, but rather the application of that tool. How you apply the SCM tool to your development environment is called the usage model, or SCM process. It is this model or process that will in part determine how successfully you address your SCM issues.

1.3. Summary

This chapter helped define software configuration management in simple terms as the mechanisms used to control the evolution of a software project. An understanding of what we mean by software configuration management is crucial because if we don't know what we want to do, we have no hope of converging on a good software-development environment. To enable this understanding, we specifically discussed software-development best practices and how they are enabled by a good SCM system. We also introduced the concepts of the SCM tools and processes that are used to implement those best practices. In Chapter 2, "Growing into Your SCM Solution," we begin to explain how to use those concepts and processes efficiently and effectively.

Chapter 2. Growing into Your SCM Solution

When you understand the best practices behind software configuration management (SCM), you need to understand how to apply SCM processes and tools successfully. One of the biggest mistakes that people make is assuming that one size fits all or that one solution is the "right" solution. In reality, different software projects have different requirements and varying degrees of complexity. Thus, an SCM tool and process that works well for one project might be woefully inadequate for another. Requirements can also change over time. A software project that is just beginning likely has a very different set of requirements than those of a project that has been running for 18 months.

The key to successful SCM on any project is to allow as much change to occur to the software as possible without losing control over the software. Like a race car driver in a race, to win, you must speed up software development to the limit while still staying on the track. You must constantly tune the SCM process, as you would a race car engine or suspension, so that the overhead involved with the SCM tool meets the changing project requirements. In a nutshell, you must have an SCM tool that is highly customizable and flexible. You must be able to perform SCM tool customizations quickly and easily.

This chapter discusses the changes software projects undergo that influence the project's SCM process and tool requirements. It also discusses how SCM tools have evolved to solve SCM-related development problems in the context of five software project categories, and begins to discuss the specific characteristics in the ClearCase SCM tool that address those problems and categories.

2.1. Dealing with Changing Project Requirements

The tooling and process requirements for any software project change over time. For example, suppose you are a software developer for a small, integrated system vendor (ISV). You have developed a prototype of a new tool. A lunchtime demonstration of the prototype has caught the attention of one of the company's vice presidents, who decides that it is worth further development.

You form a team consisting of yourself, two more software developers, a technical writer, and a tester. As its first job, the team completes the first version of the product, which turns out to be very popular and a bestseller.

For the second release, you grow the team size to 12 members. The current requirements are to incorporate some complementary third-party components and expand the number of platforms on which the product will run. The team immediately begins work after finalizing the first release.

While the second release is being developed, a key customer reports some critical defects in the first release that must be fixed. A patch for the first release of the product is produced in parallel while the second release is being developed. This patch also must be incorporated into the second release.

At this point, the product catches the attention of a larger, well-established company. This company believes that the product will complement its existing product line, so it acquires the small ISV. The major development center for the new parent company is located in a different part of the country. The development team grows again to 25 members. The new team includes, among others, three more software developers located locally and five developers located remotely. Management decides to test the product using the parent company's testing group. Thus, your team needs to find a way to deliver software builds to the remote testing team.

Release 3 is full of new features and includes a number of integrations with the parent company's existing product line. As if this was not enough, a new project manager is assigned from the parent company. This manager has a different approach to change management and begins asking for more detailed reports on what changes are being made to the product and the status of those changes. And so it goes. Success breeds complexity.

Do you think that the SCM tools and processes used to implement Release 3 are the best choice to manage the development of the prototype? Do you think that the tools and processes used for Release 1 would readily support the development environment for Release 3? How do you think that your SCM tools and processes will scale as your projects and your company grow?

The changes that any software project will undergo over time fall roughly into four categories:

- Increasing complexity of the software system being developed
- Increasing complexity of the project environment needed to develop the software system
- Changing requirements based on the development life cycle phase
- Changes to an organization's management processes or personnel

The first two categories of change require scalability in tools and process. The second two change over time and are not easily predicted. Therefore, they require flexibility in both tools and process. The following subsections discuss these categories in detail.

2.2. Evolution of SCM Tools

Not so many years ago, a software product typically was developed by one person, and there was little need for SCM. As software products grew in size and complexity, their development required more than a single individual. Projects remained relatively easy to manage when project teams consisted of two or three individuals all sitting close to each other. However, it was not long before project teams grew to tens and even hundreds of developers, who might not be located at the same site.

Early on, SCM processes were developed to manage change. Typically, these processes were performed manually. One or more librarians were tasked with controlling who could access the source code. To modify a file, the developer filled out a form (paper-based) and submitted it to the librarian. This form recorded which files needed to be modified and why. The librarian ensured that none of the files was being modified by someone else at the same time. If the file was free, the librarian gave a copy of it to the developer and recorded when and to whom it was checked out. The developer, when done, provided the modified copy of the file to the librarian, who recorded the new file and placed it in the appropriate archive.

Soon SCM tools were developed to help automate and assist librarians with their jobs. Usually one tool dominated on any given operating system. The basic version-control features of these tools were as follows:

- To maintain a library or repository of files
- To create and store multiple versions of files
- To provide a mechanism for locking (to enforce serialized change to any given file)
- To identify collections of file versions
- To extract/retrieve versions of files from the repository

Early SCM tools provided these basic version-control capabilities and automated the manual-librarian approach to SCM. A developer could check out a file without the intervention of a librarian. While the file was checked out, others could not modify it. When the developer was done, the file was checked in and a new version of the file was created automatically. Today the check-out/check-in model remains fundamentally unchanged.

One widely used SCM tool was the source code control system (SCCS), developed by Bell Laboratories in the early 1970s. An alternative to SCCS was the revision control system (RCS), developed by Walter Tichy at Purdue University. Both RCS and SCCS became the predominant SCM tools on the UNIX platform. Most mainframe machines at the time also had their own primary SCM tool. For example, the configuration management system (CMS) was part of the Digital Equipment Corporation (DEC) VAX/VMS operating system. (In terms of SCM tool evolution, this chapter uses UNIX as an example. Similar tool evolution was occurring in the mainframe environment. In fact, there were even some SCM features built into a few punch card systems.)

These early version-control tools usually offered a way to label or mark a particular version of each file for a set of files. This is called a *configuration* and was used to identify a specific version of the overall product (such as Release 1).

These tools greatly improved efficiency over the manual approach. They offered the classic SCM capabilities of being able to identify pieces of the system, control change to those pieces, and have an audit trail of who modified which files and when.

Of course, software-development projects continued to increase in complexity and size, thereby requiring larger teams and more coordination. The individuals supporting these

2.3. Summary

Software configuration management is an essential engineering discipline for the success of your software projects. A project's SCM requirements will change over time because of the increasing complexity of the software system being developed, the increasing complexity of the project environment needed to develop the software system, changing requirements based on the development life cycle phase, and changes to an organization's management processes or personnel.

The SCM tool that you use must be both flexible and scalable, so as to meet your changing project requirements. It is up to the project manager to take software configuration management seriously and to strike the right balance between process enforcement and configuration audit. The objective must be to maximize the rate of change (productivity) while still maintaining control, to produce a high-quality product on time with the required functionality.

ClearCase is a commercially available SCM tool that provides the capabilities described in this chapter. The remainder of this book focuses on how ClearCase supports SCM and the SCM best practices, and how it can be applied to a wide variety of software-development projects. Much of the discussion is oriented to the IBM Rational Software approach to managing change, called Unified Change Management (UCM).

Chapter 3. An Overview of the Unified Change Management Model

Although most software-development organizations recognize the value of and the need to use a software-configuration management tool to control software development, relatively few understand what effective software configuration management is. Even fewer have constructed a means for software configuration management that provides business value. This chapter explores exactly what Unified Change Management (UCM) is, along with the specific traits that allow it to provide value to a development organization. We discuss the underlying functionality of ClearCase and ClearQuest that serve as the foundations for UCM, and we explore the roles that exist in a software-development project, and how they intersect and interact with the UCM process.

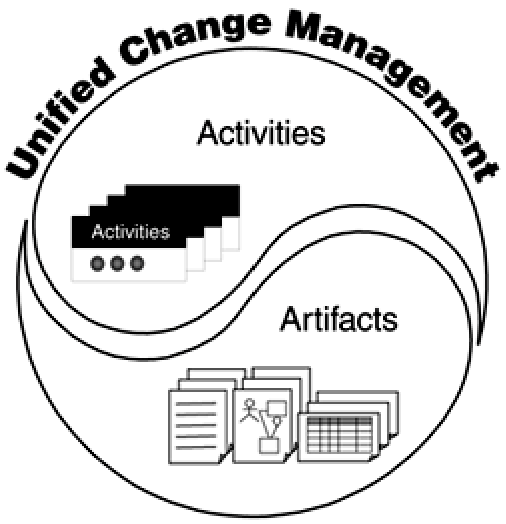
3.1. What Is UCM?

Unified Change Management is a software-configuration management process for software development that spans the development life cycle, managing change to requirements, design models, documentation, components, test cases, and source code. The UCM model reflects the 20-plus years of experience Rational Software has providing software development and SCM solutions to a broad range of customers. Rational studied the companies using the general-purpose Clear Case solution and found that many of them followed similar usage model patterns. Rational studied these models and applied their best practices to define the UCM model, generalizing the implementation for easy configuration in diverse development environments.

Fundamental to UCM is the unification of the activities used to plan and track project progress with the artifacts being changed (see Figure 3-1). Implementation of the UCM model is realized by both process and tools:

- ClearCase manages all the artifacts produced by a software project, including system artifacts and project-management artifacts.
- ClearQuest, while not required for UCM, provides enhanced management capabilities for the project's defects and requests for enhancements (referred to generically as activities), and provides the charting and reporting tools necessary to track project progress.

Figure 3-1. Unified Change Management, combining activities and artifacts.



3.2. The Value of UCM

UCM was derived from observed best practices in thousands of development organizations that demonstrated a capability to develop software in a robust, scalable, and repeatable way. By automating these best practices, UCM provides value to a development organization in many ways, but four areas are key:

- Abstraction
- Stability
- Control
- Communication

These key areas are discussed next.

3.2.1. Abstraction

Through the unification of activities and artifacts, UCM raises the level of abstraction at which software development work is done. Humans naturally work at a high level of abstractionthat is, we work best on higher-level tasks, such as "Fix bug 22" or "Implement a search function for the Web page." We are less adept at keeping track of minute details of the things we do to accomplish those higher-level tasks. UCM lets practitioners work in the abstraction space at which humans are most efficient. While that work is underway, UCM uses the underlying functionality provided by ClearCase and ClearQuest to keep track of the details of what is done to individual artifacts and information about artifacts. This means that developers are able to do their tasks faster because they aren't encumbered by the burden of tracking the changes they make for each task; they are free to concentrate on the higher-level problems that make up their assigned tasks. This also means that integrating their tasks with the overall project will be faster and less error-prone because UCM will have kept track of the work they did and will guarantee that all of their work for an activity will be delivered to the project without the prospect of accidentally forgetting to include some work.

3.2.2. Stability

Stability is important to projects as a whole and to individual project team members. Stability in a project means that the project progresses in a known, controlled way with markers placed along the way to denote intermediate stable points. UCM provides project baselines as constructs that enable this type of project stability, as well as processes to manage these project baselines. UCM uses these baselines to implement a "baseline + change" approach to workspace configuration (see Section 3.3.2, "The UCM Baseline + Change Model").

To software developers, stability means that the work areas in which developers make their changes to project artifacts are private and isolated. Not sharing a workspace with multiple team members allows a developer to work without fear that an arbitrary, unplanned change to their workspace will impede their ability to work on their assigned activities. UCM makes it easy for developers to work in stable work areas. It allows this stability first by providing personal workspaces, or "sandboxes," that are private. When a developer joins a UCM project, part of that process includes creating at least one individual, private workspace. This workspace is the area where developers modify and test their changes before committing them to the underlying repository. UCM automatically configures workspaces for the individual developer so that they conform to the development policies defined in the UCM project.

The second way UCM provides stability for developers is by providing automated facilities to allow developers to work individually or together in groups of their own choosing on their own isolated streams of development. By isolating development onto individual streams, developers

3.3. What Is ClearCase?

ClearCase is an SCM tool that provides automation and support for the SCM best practices (see the "SCM Best Practices" section in Chapter 1, "What Is Software Configuration Management?"). ClearCase provides an open architecture that is used to implement and automate a wide range of SCM solutions. ClearCase is employed in many different development environments on many types of applications, such as IS/IT systems, embedded systems, telecommunication systems, financial applications, Web site content, and other commercial and government software systems. Today companies in many diverse industries are successfully using ClearCase as the cornerstone of their SCM environments.

ClearCase solves a broad range of SCM-related problems and provides both general-purpose and specific solutions. The general-purpose solutions make very few assumptions about how ClearCase will be applied to a development environment. A certain amount of effort is required to determine how best to apply the general-purpose solution to specific SCM needs. This general-purpose SCM functionality is referred to in this book as base ClearCase. For those looking to apply SCM in the shortest period of time with the least effort, Clear Case also supports the UCM out-of-the-box usage model, which provides a specific solution. You must decide whether to invest in tailoring base ClearCase (the general solution) to your environment or to begin with the UCM model (the specific solution) and later extend it for your own requirements.

3.3.1. The ClearCase UCM Model

ClearCase UCM is based on two key concepts: activity-based SCM and component management. Both of these concepts are evident in the more advanced usage models built on top of ClearCase.

Activity-based SCM is a means of creating and manipulating a consistent change as a single named entity rather than a set of file versions. A set of versions is called a *change set* in ClearCase. The change set is an *attribute* of an activity. An *activity* is a named object that identifies the specific task, defect, feature, or requirement implemented or satisfied by the versions in the activity's change set.

Component management is a method of breaking a software system into smaller manageable pieces. Specifically, in ClearCase and SCM, component management is a means of collecting a set of files and directories into a larger entity that can be versioned and managed as a whole.

The UCM model focuses on minimizing the disruption to the software developer while maximizing the benefits gained from following sound SCM best practices. The project manager and integrator are provided with the tools necessary to manage the SCM aspects of the project in an intuitive fashion. Because the UCM model provides higher-level objects and a well-defined process, a significant number of manual steps have been automated. Therefore, for example, a developer does not need to understand the branching structure being used to manage parallel development.

Briefly, ClearCase UCM works as follows:

1. Software files and directories are organized into versioned components (ideally based on your system architecture).
2. Project managers create projects and assign project teams to work on these components.
3. Developers make changes to components, files, and directories based on assigned activities (tasks, defects, change requests).
4. New file and directory versions are collected during development and associated with

3.4. What Is ClearQuest?

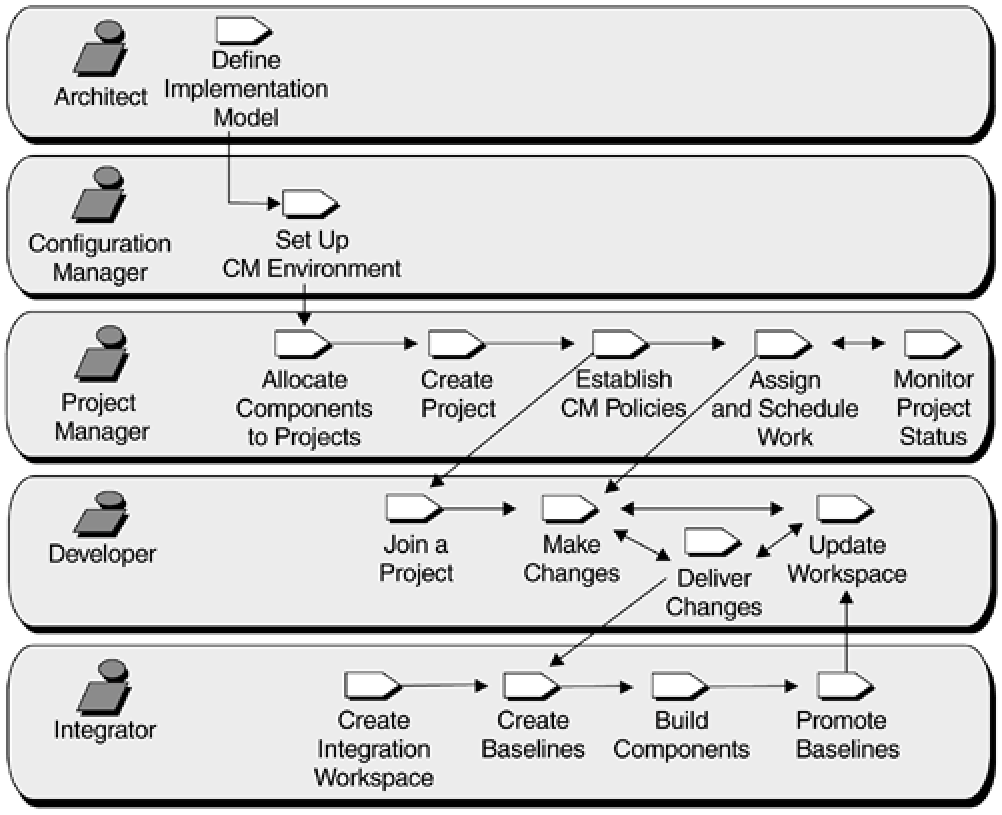
ClearQuest is an IBM Rational Software product that provides out-of-the-box support for recording, tracking, and reporting on all types of requests for change to a software system (such as defects and enhancement requests). Underlying ClearQuest is a customizable-workflow engine that enables organizations to define and enforce the process to which they want requests for change to conform.

ClearQuest has three key parts: a user interface (the client); a back-end core, which provides an interface to the data store; and a designer used to create and customize the change-request management interface and processes.

ClearQuest can be used to extend the UCM model with support for change request management. The UCM process diagram in Figure 3-5 shows two steps the project manager performs: assign and schedule work, and monitor project status. ClearQuest is the technology that supports these steps. With ClearCase alone, activities have only a name, a one-line description, and a change set. ClearCase maintains the essential configuration-management information, but not the change-request management information. To support change request management, additional data must be maintained, such as state, assigned user, a long description, priority, severity, and so on. In a UCM environment, ClearQuest provides these capabilities. If you are interested in an integrated change-management solution, you need both ClearCase and ClearQuest.

Figure 3-5. UCM process overview.

Source: **This figure is based on the configuration and change-management core workflow of the Rational Unified Process [RUP 5.5 1999].**



With ClearCase UCM alone, activities are used as the mechanism to track a change set (see the section "Organize and Integrate Consistent Sets of Versions Using Activities" in Chapter 1

3.5. ClearCase UCM Process Overview

To provide an overview of the ClearCase UCM process, this section discusses it in terms of five roles and the steps these roles perform. The roles (in order of appearance in the process) are architect, configuration manager, project manager, developer, and integrator.

Your organization might use different names, but each role is key to successful SCM. A single individual might take on more than one role, or multiple individuals might share responsibility for a single role. The role(s) an individual performs usually depends on that person's technical background, the size of the project, and the size of the system being developed. You might want to map the ClearCase UCM roles to the individuals and roles in your own organization as you work through this book.

Figure 3-5, an overview of the UCM process, lists the roles involved and the steps they perform. For example, a developer's first step is to join a project. The main team members are listed, followed by the steps they perform. Arrows indicate the typical order of the steps.

3.5.1. The Architect

The architect has a deep understanding of the system architecture. In terms of SCM, the architect is responsible for determining how the system architecture should be physically realized (that is, how to group and map the various design objects to the physical files and directories that will implement the design).

3.5.2. The Configuration Manager

The configuration manager is familiar with an organization's configuration and change-management processes and with the SCM tools being used. The configuration manager is responsible for creating and maintaining the physical in frastructure necessary to implement the design. This primarily involves creating and maintaining repositories and importing existing files and directories. (In some organizations the configuration manager is also responsible for things such as disk space allocation, network resources, and backup strategies as they relate to SCM data. The process described here allocates these activities to the system administrator.)

3.5.3. The Project Manager

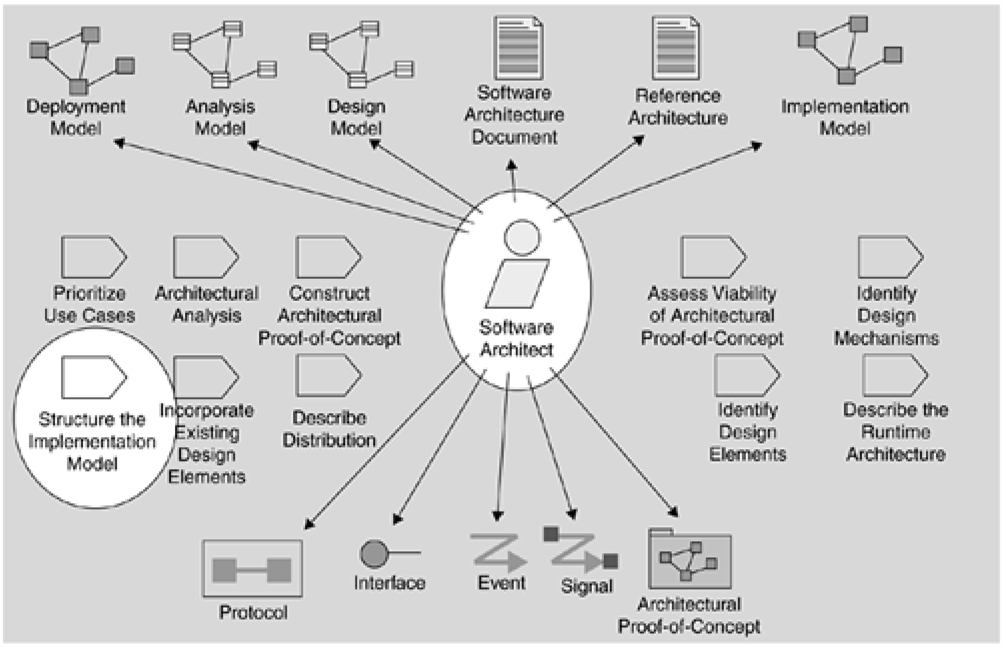
The project manager understands an organization's change-management processes and project-management policies. In terms of SCM, project managers are responsible for assigning and scheduling work activities and scoping components to project teams. When components are scoped, the project manager for a given project is responsible for creating the physical ClearCase projects and for defining the SCM policies by which the project is governed. Policies are defined in terms of both written policies and SCM tool settings that define automation and enforcement policies supported by UCM.

NOTE

In some organizations, the configuration manager is responsible for carrying out the policy decisions made by the project manager.

In larger organizations, a project manager might be two different people. The traditional division is between a management-oriented role, which allocates the work and establishes the policies, and a technical-lead role, which creates and configures the project and carries out the work. (Other key activities of the project manager are to create, monitor, and maintain a project plan. These are project-management tasks and not part of the SCM process. With

3.6. The Architect: Defining the Implementation Model


Deployment Model
Analysis Model
Design Model
Software Architecture Document
Reference Architecture
Implementation Model
Prioritize Use Cases
Architectural Analysis
Construct Architectural Proof-of-Concept
Software Architect
Assess Viability of Architectural Proof-of-Concept
Identify Design Mechanisms
Structure the Implementation Model
Incorporate Existing Design Elements
Describe Distribution
Identify Design Elements
Describe the Runtime Architecture
Protocol
Interface
Event
Signal
Architectural Proof-of-Concept

Typically, software-development requirements and budgets are established at product inception and evolve during product elaboration and development. The architect uses the requirements to create a software architecture or to modify an existing one. The software architecture serves as a logical framework to satisfy the requirements.

The concept of architecture has many facets, some that are relevant to soft ware configuration management and others that are not. The authors of *The Unified Modeling Language User Guide* [Booch 1999] and *The Rational Unified Process: An Introduction, Second Edition* [Kruchten 2000] define architecture this way:

Software architecture encompasses the following:

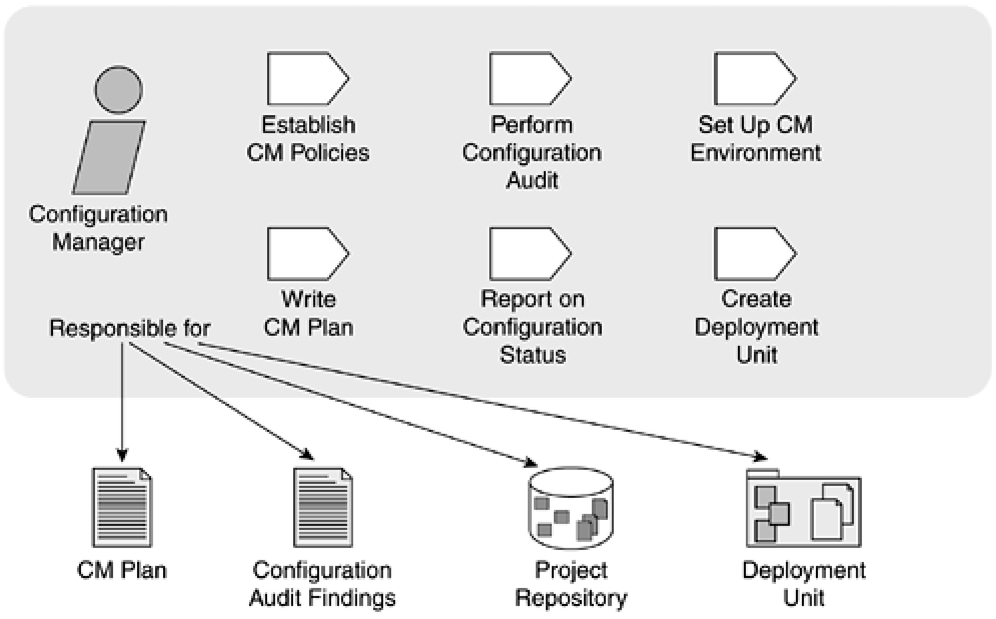
- The significant decisions about the organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements
- The composition of these elements into progressively larger subsystems; the architectural style that guides this organization, these elements, and their interfaces, their collaborations, and their composition

Software architecture is concerned with not only structure and behavior, but also usage, functionality, performance, resilience, reuse, comprehensibility, economic and technologic constraints and trade-offs, and aesthetic issues. [Kruchten 2000]

In terms of SCM, architecture is concerned with the organization, grouping, and versioning of the physical files and directories of the system, both as they are organized in the development environment and as they are deployed on the target system. Some projects use high-level design documents to describe these aspects of architecture. More recently, models are being employed to visually represent the architecture, providing different architectural views of the system. (The term *view* as used here refers to an abstraction of a system architecture. An abstraction shown from a given perspective omits entities that are not relevant to that perspective. This use of *view* is not related to the ClearCase view object.

3.7. The Configuration Manager: Setting Up the SCM Environment

Before the first developer can start making changes, the configuration manager must establish the SCM environment. Two key steps are involved: establishing the hardware environment and establishing the development environment.



To set up the hardware environment, the configuration manager works with the system administrator to assess and allocate machine resources (such as designated servers and disk space). To complete the hardware environment, the configuration manager must work with the system administrator to install and configure the necessary software tools (such as ClearCase) on both server and client machines.

The second step, establishing the development environment, involves the following:

1. The configuration manager works with the architect to finalize the implementation model (the mapping of design objects to logical packages and then to ClearCase components).

2. The configuration manager determines where the components will be stored (which SCM repositories).

3. The configuration manager creates the repositories and the components.

4. The configuration manager creates the product directory structure.

5. The configuration manager imports any existing files and directories into the repositories, to create the initial set of versioned elements.

6. The configuration manager establishes an initial baseline for all components. A component baseline records a specific version of every element (files and directories) in a component.

7. If applicable the configuration manager establishes one or more composite baselines that record a set of baselines for inclusion in projects as subsystems.

3.8. The Project Manager: Managing a Project

A ClearCase project is created by the project manager and is used by a team to produce new baselines of one or more components. A ClearCase *project* is an object that contains the configuration information needed to manage a significant development effort. Project managers configure the ClearCase project to define the scope of work for the project (the set of components) and to set policies that govern how developers access and update the set of source files. A ClearCase project also defines a shared area where development changes are integrated.

The size of the overall development initiative influences the number of projects that are created to fulfill it. Small initiatives might have one project that is producing a new release of an entire system. Large initiatives have multiple projects working together to produce a single release of a system. It is important to distinguish between the product being produced and the project or projects producing the product.

The project manager does the following:

1. Creates the project

2. Identifies the components or composite baselines (already defined by the architect) needed by the project

   [View full size image]

3. Identifies which components are to be modified (writeable) and which components are merely referenced (read-only) by the project

4. Identifies the baseline (version) of each subsystem or component from which developers will start their work

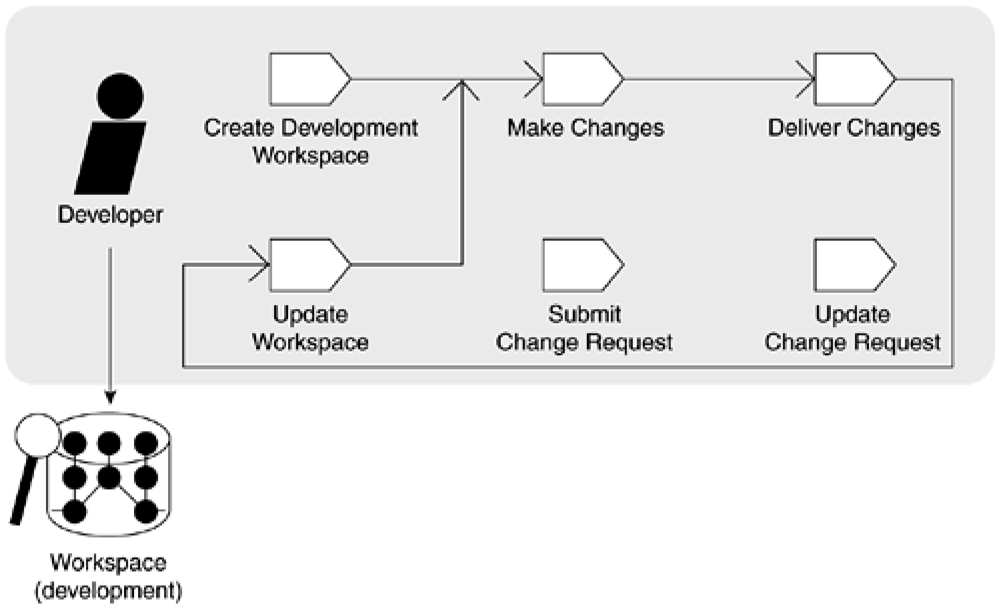
5. Defines the policies that govern how the usage model is applied to the project

Defining policies for a project means configuring how ClearCase will automate and enforce policies during development on that project, and defining other project policies, which might not be automated by ClearCase but must be documented and followed manually.

After the project manager has created the project, defined the scope of work for the project (indicating which components will be modified or referenced), and established the project's policies, developers can join the project and begin working.

3.9. The Developer: Joining a Project and Doing Development

Developers join one or more projects where they perform their work. The process of joining a project creates the developer's *workspace*. In ClearCase, a developer's workspace is comprised of a ClearCase view and a development stream. A ClearCase view provides a window into the files being managed. The stream defines the configuration for the view and determines which versions of the files will be displayed.



When developers join a project, they can work directly in their view or through any number of integrated development environments (IDEsfor example, IBM WebSphere Studio Application Developer, or Microsoft Visual Studio). In views, developers check out, edit, build, unit-test, debug, and check in file and directory elements as needed to accomplish their tasks. All modifications are associated with an activity, forming a change set.

Activities represent the tasks that developers work on. Examples include "implement feature A," "fix defect 109," or "redesign the search algorithm for orders." In UCM, the project manager can plan and assign activities if a change-request management tool such as ClearQuest is being used (see Chapter 12, "Change Request Management and ClearQuest"). Developers can also define activities when they start doing work if a project is using an informal change-management process. The developers need only track the change sets.

When developers complete an activity, they deliver their changes to the project's integration stream, making them available to the integrator for integration and inclusion in the next component baseline(s). Developers must also keep their own development stream current with changes being made by other developers on the project. ClearCase supports an explicit operation called *rebase* (short for *rebaseline*). Rebase incorporates new baselines created by the integrator into the development stream. After a rebase, developers will see new baselines that include changes that other developers have delivered and that the integrator has integrated.

After developers have created, tested, and delivered changes, it is time for the integrator to build and test those changes.

3.10. The Integrator: Integration, Build, and Release

The job of the integrator is to take changes that developers deliver, produce new baselines of components, and promote those baselines for internal project consumption or external use.

Each project has one integration stream into which developers deliver their activities. The integrator uses an integration view attached to the integration stream to create a working environment for performing the first step of the integration: creating a new baseline. The integrator creates an integration view, configured by the integration stream to select the previous project baseline plus all the new versions identified by the delivered (but as yet unincorporated) activities.

The integrator freezes the integration stream to ensure that the integration view will select a fixed set of versions and then builds the system. If the build succeeds, the integrator creates new baselines of the components that have changed.

If the system passes a certain level of testing, the integrator can promote the baselines. *Promotion* is a means of marking baselines as having either passed or failed a certain level of testing. For example, an integrator could promote a baseline to Built, Tested, or Rejected. These promotion levels are defined by the project manager as part of defining project policies and are largely used by external consumers of the baseline. By promoting project baselines to a certain level of quality, the integrator implicitly selects the baselines as the project's recommended baselines. The recommended baselines are the default baselines presented to developers when they join a UCM project or rebase (update) their development streams.

3.10.1. Releasing a Component

To release a component, the integrator places all project deliverables, which are artifacts, under version control. The product deliverables can reside in one of the development components or in a separate component reserved for deliverables. Deliverable components are components that contain elements that have been built and are shared with other teams. These elements are included in the build process (for example, statically linked libraries), are included in the final system's runtime environment (for example, executables, dynamically linked libraries), or make up the deliverable (for example, documentation).

The elements of a deliverable component are also included in a baseline of that component. Other development teams can select and share deliverable or source elements by referring to the appropriate component and its baseline.

3.10.2. System Integration

For large systems, there is likely to be more than one project producing modifications to many components, generating a need for system integration. System integration happens when component and subsystem (composite) baselines are assembled into a final system.

With the UCM, system assembly is done by creating a system project and selecting the baselines that are being produced by the various subprojects. In other words, all components for the system project are reference (read-only) components. The complete system can then be built and tested as a whole.

3.11. Summary

UCM provides the software-configuration management tools and processes to achieve effective, efficient software development. It does this by building upon a model that unifies the activities and artifacts upon which projects are built and progress. This model serves to raise the level of abstraction at which development is done; to provide stability for projects and individual contributors; to manage, track, and control the flow of changes in a project; and, finally, to provide facilities for automated project metrics as well as real-time communication about a project's activities and artifacts. UCM is built upon the foundations of ClearCase and ClearQuest to manage the project's artifacts and activities, respectively. You have seen that the UCM process can be described in terms of five roles: the architect, the configuration manager, the project manager, the developer, and the integrator.

Chapter 4. A Functional Overview of ClearCase Objects

This chapter provides an overview of ClearCase objects and concepts. It serves as a bridge between general SCM terminology and ClearCase-specific terminology.

4.1. The Repository: Versioned Object Base

At the heart of any SCM system is the object repository. ClearCase repositories are called *versioned object bases*, or VOBs. The ClearCase online help reads, "A VOB is the permanent data repository in which you store files, directories, and metadata." Anything that can be represented as a file or directory can be managed in a ClearCase VOB.

ClearCase VOBs support all of the characteristics listed in the first SCM best practice: The SCM repository must be scalable, fault tolerant, distributable, and replicable (see Chapter 1, "What Is Software Configuration Management?"):

- **Scalable** ClearCase VOBs can grow from hundreds of files and directories to many thousands of files and directories. Files and directories in a ClearCase VOB can be moved between VOBs if a VOB becomes too large. VOBs can be split and joined.
- **Fault tolerant** ClearCase VOBs have an internal database that does not require additional database management. The VOB database and ClearCase architecture ensure that transactions such as check-out on a file are atomic and so ensure that the permanent datastore does not get corrupted.
- **Distributable** ClearCase VOBs can be distributed across different servers in the network transparently to the end user. Because VOBs can be moved, you can distribute the load as needed to meet project demands.
- **Replicable** ClearCase VOBs can be replicated in two or more sites. Replication means full copies are made and kept up-to-date (in sync) with each other at different geographic sites. This is critical for geographically distributed development, where the network between sites is often not as reliable or often does not have as high a bandwidth as the local area network.

ClearCase VOB technology has an interesting twist not found in traditional SCM systems. In systems such as the revision control system (RCS), the repository and the file system are two separate entities. With ClearCase, you can combine the two.

Instead of being a black-box repository from which things need to be inserted and extracted, the ClearCase repository displays its contents as files in a file system, which can be operated on in the same way as you would files in the native file system. In the same way you add disks to a file system, you can create as many ClearCase VOBs as required. The VOBs can be distributed across many machines but referenced as if they were one single repository. Figure 4-1 shows two servers and three VOBs, and how the VOBs plug into the native file system. A scalable solution such as this one is particularly critical for large organizations.

Figure 4-1. Distributed VOB architecture.

4.2. Workspaces: Snapshot and Dynamic Views

One of the essential functions of an SCM tool is to establish and manage the developer's working environment, often referred to as a *workspace*, or "sandbox." In ClearCase, workspaces are called *views*. The primary purpose of the view is to provide developers with a stable and consistent set of software, where they can make changes and perform unit testing. ClearCase views select the appropriate versions of the files and directories, as defined by a set of configuration rules, from all available versions in a VOB. For example, a view might select all versions that were used to build Release 2 of a system.

The view makes the files and directories available for browsing, modifying, and building. The versions selected in a view should be stable (the versions of the files and directories you are working on should not change without warning) and consistent (the versions of multiple elements selected should be complementary). For example, if you are fixing a bug in Release 1, you don't want to select some element versions from Release 1, some from Release 2, and some of the latest element versions. Instead, you want all the Release 1 element versions plus your own modifications.

Every ClearCase view contains a set of rules that define the configuration for that view (that is, that define which versions should be seen). These rules are called a *configuration specification*, or config spec. For projects that use the UCM model, ClearCase automatically generates these config specs (for a discussion of streams, see the section "Project Management: Projects, Streams, and Activities," later in this chapter). For projects that do not use UCM, you can use the default config spec, generate config specs by hand, or automate config spec generation using scripts.

ClearCase provides two main types of views, *snapshot* and *dynamic*. A third type of view is really a subtype of a snapshot view. This third type of view, the *Web view*, is used by the ClearCase Web Client and the ClearCase Remote Client. Each type has its own advantages, and you should expect to use both snapshot views and dynamic views to maximize the benefits of ClearCase. Snapshot views have copies of the files loaded into them from the VOB. Although they take longer to set up and require more disk space than dynamic views, build performance is better because there is no dependency on the network to access the files. Web views are like a snapshot view, maintaining copies of files loaded from the VOB. However, instead of locating the database portion of the view on the ClearCase Web client, where the files are loaded, this database is located on the ClearCase Web server. Dynamic views reference files directly out of the VOB. No copies are made, so setting up a dynamic view is fast and takes very little disk space. The following sections go into more detail on each type of view.

4.2.1. Snapshot Views

Snapshot views are similar in approach to traditional development sandboxes (that is, they are uncontrolled copies of a software system's files and directories). When you create a snapshot view, you specify a local directory into which the files and directories you want to work on are copied. This is referred to as a `Get` operation or a read-only check-out by some SCM tools. ClearCase refers to this as loading a snapshot view.

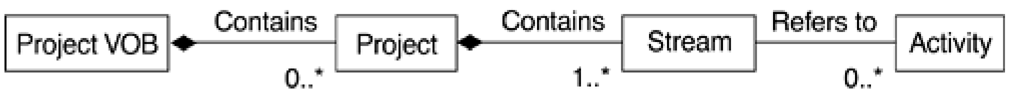
The use of ClearCase snapshot views differs from the traditional approach of local file copies in a number of ways. Snapshot views have a database that keeps track of what versions have been loaded into the working directory. Snap shot views mirror the directory structure as part of a load/update operation. They keep track of when a developer changes the read-only bit on a file and modifies it without issuing a check-out command. This is called hijacking the file. During update operations, hijacked files are not overwritten and can be turned into check-outs.

Another difference from traditional sandboxes is that snapshot views can be updated in a

4.3. Project Management: Projects, Streams, and Activities

Because of the complexity and size of software-development efforts, project managers need automation and tools to help organize and manage large software projects. ClearCase UCM has objects and automation that assist in the management and organization of software projects: projects, streams, and activities (see Figure 4-5).

Figure 4-5. Projects, streams, and activities.



4.3.1. Projects

The Project Management Institute defines a project as follows:

Projects are performed by people, constrained by limited resources, and planned, executed, and controlled. A project is a temporary endeavor undertaken to create a unique product or service. Temporary means that every project has a definite beginning and a definite end. Unique means that the product or service is different in some distinguishing way from all similar products and services. Projects are undertaken at all levels of the organization. They may involve a single person or many thousands. They may require less than 100 hours to complete or over 10,000,000. Projects may involve a single unit of one organization or may cross organizational boundaries as in joint ventures and partnering. Projects are often critical components of the performing organization's business strategy. [PMI, 1996, p. 4]

A ClearCase *project* directly maps to this definition. A ClearCase UCM project represents a group of individuals collaborating to produce new baselines of one or more components of a system (or perhaps the entire system). A ClearCase project is an object whose attributes define the scope of work for that project (which components are being worked on), the policies that govern the work for that project, the workspaces (streams and views) used on that project, and which activities are being worked on by the team members.

Project objects are created in PVOBs and can be organized into folders. A Project Creation Wizard is used to create a new project, and a Project Explorer is used to browse and modify projects.

4.3.2. Streams

A workspace is a logical concept in ClearCase UCM that is implemented with two objects: a *stream* and a view. A stream defines the working configuration for the view (or views) associated with it. It contains the information needed to automatically generate a configuration specification for the view. Unlike base ClearCase users, ClearCase UCM users do not have to create or modify the config specs. Streams logically define configurations in terms of baselines and activities.

Figure 4-6 illustrates how this works. The VOB shown on the left contains many elements and many versions. In the middle, the triangle represents a stream. The base of the triangle lists one or more foundation baselines for the stream. The upper portion of the triangle lists a set of activities (along with their change sets). The view (shown on the right) uses the stream's activities and baselines to display particular versions of elements stored in the VOB.

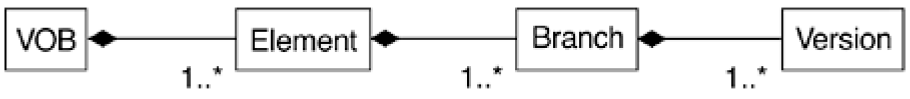
Figure 4-6. Stream/view relationship.

4.4. Versioned Objects: Elements, Branches, and Versions

The atomic object put under version control in ClearCase is referred to as an *element*. Elements are file system objects: files and directories. Every element records versions of the file or directory it represents. So, when a user checks in a file, a new version is created for that element. These element versions are organized into branches. A *branch* is an object that specifies a linear sequence of element versions. They are used for many purposes, such as doing parallel development and maintaining variants of the system.

Each element starts life with a main branch and a null zero version that does not have any content. This is represented in ClearCase as `/main/0`. The first new version checked in creates Version 1 on the receiving branch or stream. In a UCM environment, this first version of an element is created on the UCM stream associated with the ClearCase view in which the user is working. The organization of versions into branches provides a time-ordered representation of each element's history. The relationship between the repository (VOB), the elements it contains, and each element's branches and versions can be seen in UML notation in Figure 4-15.

Figure 4-15. VOB, element, branch, and version relationships.



ClearCase provides both a command-line way and a graphical way to view an element's branches and versions, both called a *version tree*. A graphical example of a version tree display is shown in Figure 4-16. Boxes indicate the branches. Circles indicate time-ordered versions as they are checked in. The arrows indicate merges of changes from one branch to another. The text next to a version indicates a label that has been applied to that version.

Figure 4-16. Graphical version tree.

4.5. Component Management: Components and Baselines

UCM provides the capability to group together files, directories, and folders that should be controlled and versioned as a unit into UCM components. UCM also provides the capability to define versions of individual components as UCM baselines. UCM further provides the capability to group baselines into logical groups called composite baselines. These UCM constructs and how to use them are discussed in this section.

4.5.1. Components

A ClearCase *component* groups files and directories that should be developed, integrated, baselined, and released together. The files and directories grouped into a ClearCase component usually implement a reusable piece of the system architecture (although this is not enforced).

Components are defined by identifying a root directory. That directory and all files and subdirectories are considered to be part of that component. For more details, see "The Architect: Defining the Implementation Model," in Chapter 3.

4.5.2. Baselines

A version of a component is a *baseline*. A component baseline identifies zero or one version of each element that is contained in that component. Component baselines are used to configure a stream and ultimately to provide the right information to the view to determine what versions of the files and directories should be displayed.

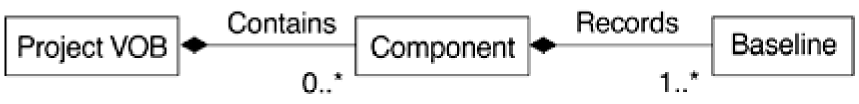
NOTE

Components have baselines the way elements have versions. When you change an element, you create a new version of that element. When you change elements in a component, you create a new baseline of that component. A stream groups together a set of component baselines. When you perform a baseline operation on the project's integration stream, you are creating a new set of component baselines, one for each component that has been modified. Composite baselines provide the functionality of a project-wide baseline by defining a single baseline that encompasses all the relevant individual component baselines required to define a stable project configuration.

Each baseline has a user-defined quality or promotion level. That is, with ClearCase, a company can define different levels of testing and can mark baselines indicating the level of test ing that each has passed. This makes it easier to perform reuse because other project teams can determine what level of quality any given component baseline has reached.

Figure 4-17 shows the relationship among the project VOB, components, and baselines.

Figure 4-17. Project VOB, component, and baseline relationships.



Composite Baselines

4.6. Process: Labels, Attributes, Hyperlinks, Triggers

ClearCase provides a number of additional objects that are useful for a wide variety of purposes. In many cases, these are used by ClearCase itself to implement specific functionality, but they are also available to you when you want to automate or enforce project processes and/or record additional data and relationships between objects in the system. The major objects, which are often referred to as *metadata*, are labels, attributes, hyperlinks, and triggers. These objects have been used to implement specific project policies as part of UCM.

4.6.1. Labels

A *label* is an instance of a label type and is attached to a version of an element. A *label type* is a named tag that can be used to identify a consistent set of element versions. For example, you could create a label type called `RELEASE1` and attach a label of that type to all the versions of the elements that make up Release 1. In and of themselves, labels do not represent any semantics between elements other than those your organization defines.

In general, UCM users will be using components and baselines, and should not need to create and manipulate labels themselves (this is handled by UCM automation). However, if a project using UCM needs to share code and interact with a project using base ClearCase, labels would be used. The UCM project would identify the label associated with a given component baseline for sharing outside that project. For sharing internally, a ClearCase label can be imported as a UCM baseline. Figure 4-11 shows how labels are displayed in the Windows version tree browser.

4.6.2. Attributes

Attribute types form a name/value data set that can have *attribute* instances attached to almost any ClearCase object. For example, you might create an attribute type called `review_status` that has one of the values of `passed`, `failed`, or `pending`. An instance of this type can be attached to each version of an element, indicating the code review status of that version.

Basically, attributes can be used to associate arbitrary data with objects in the system. Attributes can be attached to elements, baselines, branches, ver sions, hyperlinks, projects, components, and activities.

Attributes are defined to contain one of these data types:

- **Integer** Integer values
- **Real** Floating-point values
- **Time** Date/time values
- **String** Character string values
- **Opaque** Arbitrary byte sequence values

Any of these can also have an enumerated list to restrict the legal values. For example, a string attribute type named `priority` could be defined as a string, enumerated type `HIGH`, `MEDIUM`, and `LOW`.

4.6.3. Hyperlinks

Hyperlinks define relationships between objects. For example, a predefined hyperlink type

4.7. Building: Clearmake, Derived Objects, Configuration Records

One aspect of configuration management that is often overlooked is build management. ClearCase provides significant functionality aimed at supporting reproducible builds, object sharing between developers, parallel builds, and distributed builds. ClearCase supports these best when you are using make technology as your build engine. The key build features are described in the following subsection.

4.7.1. Build Audit

Typically, when a software system is built, many object files, libraries, and executables are produced. ClearCase calls these *derived objects*. Often the traceability between the derived objects and the versions of the source used to produce them is lost, making it impossible to reproduce builds. This can lead to an inability to maintain, debug, and patch releases being used by your customers.

ClearCase provides a make-compatible build tool called *clearmake*. When clearmake is used to build derived objects, a record is kept of who built each derived object, when it was built, on what platform it was built, and, most important, what files and what versions of those files were referenced during the build. This information is called a *configuration record*. All the configuration records for all derived objects that are part of a build make up the *build audit*. ClearCase provides the capability to generate a bill of materials for any build (that is, what versions of what files were referenced for the entire build).

When some other type of build tools, such as Ant, CruiseControl, and customized programmatic build scripts are used, ClearCase provides the `buildaudit` command. The `buildaudit` command is used in conjunction with these other types of build tools to produce a configuration record and can generate a comprehensive bill of materials for any build.

NOTE

For further information on using ClearCase with Ant, you can review the materials at http://ant.apache.org/manual/OptionalTasks/clearcase.html.

Information on using ClearCase with the CruiseControl build tool can be found at http://www.buildmeister.com/articles/cruisecontrol-clearcase.php.

The build audit can be used to compare builds, enabling you to see what versions of what files are different between two builds or even what compiler options might have changed. A frequent problem in debugging is having an error show up in code that everyone is convinced was not changed. One of the big advantages of being able to compare two builds is the ability to quickly determine which files or build options changed from one build to the next.

4.7.2. Object Sharing

By keeping a configuration record for each derived object, it becomes possible to automate sharing of derived objects. This means that instead of rebuilding the same object, library, or executable over and over, it can be shared between developers. This happens automatically when using clearmake because clearmake can determine that the derived object you are about to build uses the same versions, same compiler, same compiler switches, and so on as one that already exists. This derived object sharing can significantly reduce both the development build times and the amount of disk space required for each developer's workspace.

Team LiB ◀ PREVIOUS NEXT ▶

4.8. Summary

This chapter explored the ClearCase objects that make up a ClearCase UCM environment and the relationships between them. From the versioned object base at the highest level to individual streams, baselines, and activities, you have also seen how these constructs can be used to provide a project with structure, control and tracking, and management of the assets under development.

Team LiB ◀ PREVIOUS NEXT ▶

Team LiB ◂ PREVIOUS NEXT ▸

Chapter 5. Establishing the Initial SCM Environment

The quickest way to get started using ClearCase is by creating one project VOB (PVOB) and one source VOB, and importing your existing source code. However, this is effective for only small projects and small systems. For most projects, some advanced planning is advised. This chapter covers the information you need to know to do this planning and the steps you must take to set up the initial SCM environment. We cover the basics of ClearCase architecture, discuss the hardware resource requirements for ClearCase, suggest some guidelines for taking a system from its logical design to its physical implementation, cover the creation of ClearCase VOBs, and discuss setting up component baseline promotion levels.

ClearCase installation and other administrative details are not covered in this book. It is expected that you will consult the ClearCase documentation set for installation and administration. Another resource that is full of practical advice on how to successfully deploy and implement a ClearCase environment is *The Art of ClearCase Deployment* [Buckley and Pulsipher, 2004].

Team LiB ◂ PREVIOUS NEXT ▸

5.1. ClearCase Architecture Basics

The first step in creating any SCM environment is to obtain or allocate the necessary hardware and install and configure the SCM tool. This section gives you a high-level overview of the ClearCase architecture and makes some hardware configuration recommendations. Admittedly, the information provided here is a simplified view of the actual ClearCase architecture. (The ClearCase administration manual provides much greater detail.) However, you should find it sufficient for basic planning purposes.

ClearCase is a multiserver, distributed SCM tool that allows for a great deal of flexibility and scalability in hardware configuration. To understand the hardware resource requirements for ClearCase, you should start with a basic understanding of the ClearCase architecture. ClearCase is specifically designed to spread the workload across multiple machines. During evaluations, it is certainly possible to install and configure one machine to use ClearCase, but typically a ClearCase environment consists of at least one server and several clients.

When determining your hardware environment for ClearCase, you must find a home for six types of processes and the multiversion file system (MVFS). The processes are as follows:

- License server
- Registry server
- VOB server (In fact, three types of servers are associated with a VOB. The actual number of server processes running depends on the load being placed on the VOB. ClearCase automatically starts additional server processes when needed and stops these processes when demand drops.)
- View server
- ALBD server
- Client processes (such as cleartool)

The multiversion file system is a client-side requirement and is used on platforms that support dynamic views.

5.1.1. The License Server and Registry Server

The two administrative processes are the *license server* and the *registry server*. The purpose of the license server is to manage license keys and ensure that the licensing constraints are not violated. (ClearCase includes its own license manager. ClearCase licensing is truly per user. A single user can be using ClearCase at his or her desk on a Windows machine and in a test lab on two UNIX machines, and consume only one ClearCase license. ClearCase licenses are floating and have a default timeout period of 1 hour.) The purpose of the registry server is to maintain the directory and machine locations on which all view and VOB data is being stored. In this way, any ClearCase client can locate any ClearCase data, regardless of where in the network this data is stored. The license server and registry server require very little in the way of server resources. The primary consideration is that the machine on which these servers reside should be very reliable. Often the license and registry servers are placed on the same machine as the primary VOB server processes.

5.1.2. The VOB Server and View Server

ClearCase VOBs store all the files and directories being managed by ClearCase. As such, they are a global resource that most clients will need to access. Each VOB has a set of VOB server processes associated with it that handle read and write traffic to the VOB. For small projects

5.2. ClearCase Hardware Resource Requirements

One of the questions most often asked is what kind/size/make/model/type of machine should be used for a VOB server. The answer invariably is, "It depends." Many factors contribute to what type of machine will be best for your company (including amount of data, type of data, number of concurrent users, and build strategy). The information in this section and the following section on the client should be viewed as a starting point in your quest for the perfect environment.

The key to a well-performing ClearCase environment is a well-performing server machine. How do you ensure that your server machine (or machines) performs well? Easy: Buy the most loaded and most powerful machine you can. Sooner or later, you will grow into (or out of) it, and you never know whether your budget will get cut next year. However, if your company is like most, hardware resources are not always easy to acquire, and you might need to start with what you already have. So, it is useful to understand where to apply your dollars with respect to ClearCase.

Typically, the VOB server processes make the most demands on system hardware. The VOB server machine is also the most critical piece of a ClearCase network as far as performance monitoring and tuning are concerned. A VOB server machine is a machine that is dedicated to running VOB server processes. There might be more than one VOB on a single VOB server machine, and there might be more than one VOB server machine in the network. The key resources to consider when sizing a VOB server are memory, disk I/O, network bandwidth, and CPU. These are listed in order of priority to ClearCase performance, with memory being the most important and CPU being the least.

5.2.1. Memory Requirements

VOB server processes, view server processes, and the MVFS do a lot of caching and background data writing. The best thing you can do to make sure you have a well-tuned ClearCase environment is to ensure that your server and client machines have enough main memory. When a machine starts paging ClearCase data to disk, you can guarantee that performance will suffer. Memory is the cheapest way to improve ClearCase performance. Lack of sufficient memory is the most common cause of poor ClearCase performance.

VOB server machines should have a minimum of 1GB. (As of this writing, this pertains to ClearCase v2003.06.14.) The rule of thumb is that you add all the database space consumed by VOBs stored on a machine and divide by 2. This is the minimum amount of main memory you should have on a dedicated VOB server machine (see the "Other Recommendations" section, later in this chapter, for more on what "dedicated" means). You can use cleartool space or the ClearCase administration console to determine this amount. So, in short, for every 2MB of database space you have, you should have 1MB of main memory. The assumption here is that half of the data in a ClearCase VOB will be actively accessed at any given time. A more conservative approach, and one recommended if you have the resources, is to have 1MB of memory for every 1MB of VOB database size. This allows the entire set of the VOB data to be available in memory at any given time. Because VOBs will continue to grow over time, it is also good practice to have extra memory on a VOB server machine.

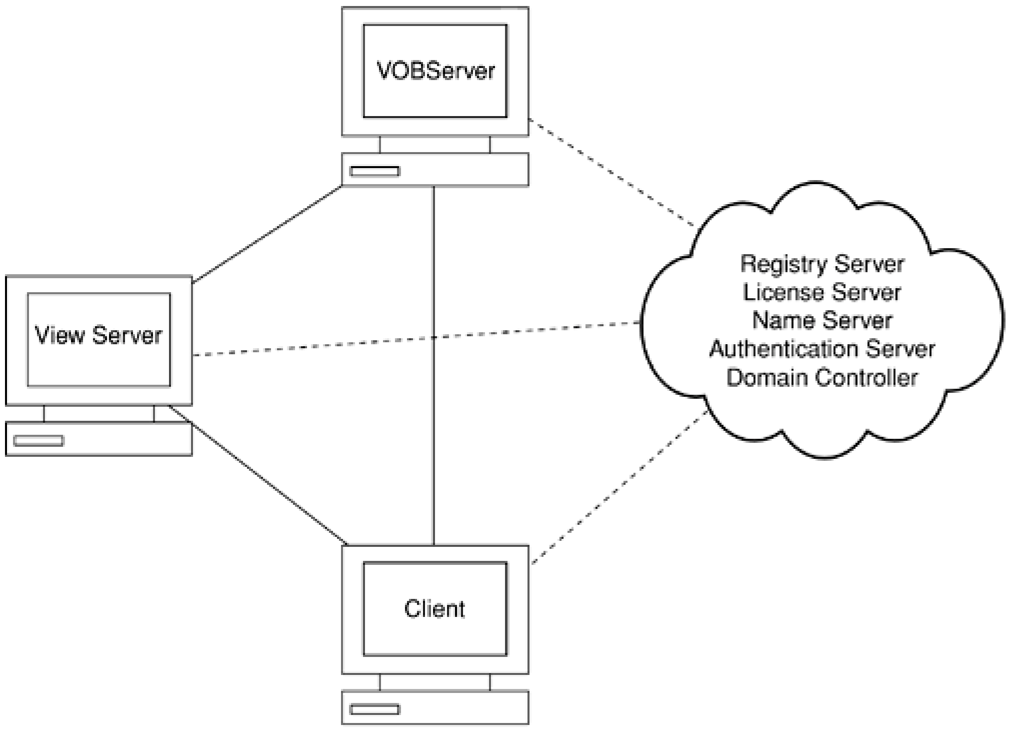
For client machines, a minimum of 512MB of main memory is recommended. Many people might feel this is too high, but this recommendation covers a minimum amount of memory for a software developer's client machine rather than specifically for ClearCase. There is clearly a trend toward Windows machines on the desktop, and often developers need many applications running at the same time. For example, a developer might have a defect-tracking tool such as ClearQuest, a design tool such as Rose, and an IDE such as Microsoft Visual Studio all open at the same time. If you add one or two ClearCase views and some ClearCase client processes such as the difference tool or version tree browser, the memory requirements quickly add up. To keep ClearCase views from paging their caches to disk, you need to ensure that sufficient

5.3. Monitoring and Tuning for ClearCase Performance

Many interdependent pieces in a development environment play a role in ClearCase performance. Additionally, exactly how these pieces interact depends on how an organization uses ClearCase. This can lead to some confusion about how to even approach the problem of monitoring and tuning to optimize ClearCase performance. Here are three simple principles to keep in mind when approaching ClearCase performance monitoring and tuning.

1. ClearCase performance depends on the interoperation of several different server roles. Figure 5-8 provides an illustration of this interoperation.

   Figure 5-8. ClearCase server roles for performance monitoring and tuning.

   In this figure, the ClearCase VOB server, the view server, and the ClearCase client form the vertices of a triangle. The edges of the triangle represent the connectivity between the ClearCase roles. When monitoring and tuning ClearCase environments, you need to pay attention to each vertex on this triangle as well as each edge. Furthermore, you need to measure the performance characteristics of the connections to a set of commonly used network resources; the ClearCase registry server, the ClearCase license server, network name servers, and network authentication. Often one machine will be configured to fill two or more roles. For example, a ClearCase client is often also a ClearCase view server, or a VOB server often also fills the roles of the ClearCase registry server and the ClearCase license server.

   Performance on each individual server depends on the interoperation of the memory, disk, network, and processor subsystems on that server.

2. Partition your analysis on each server and client into manageable pieces. One successful partitioning scheme is illustrated in Figure 5-9.

   Figure 5-9. The ClearCase performance stack for partitioning monitoring and tuning tasks.

5.4. Defining the Implementation Model

Defining the implementation model involves going from the logical design to the physical implementation of the system. Logical design elements, such as classes, are grouped into physical files and organized into physical directories. The Rational Unified Process calls this structuring the implementation model, whose purpose is the following [RUP 2003.06.13 2004]:

- To establish the structure of the implementation model
- To adapt the structure of the model to reflect team organization or implementation language constraints
- To define dependencies between subsystems
- To add test artifacts to the implementation model
- To update the implementation view of the software architecture document

Moving from logical design to physical implementation is a very important step and, if done incorrectly, can cause problems. In *Large-Scale C++ Software Design*, John Lakos discusses this issue in a section entitled "Physical Design Concepts":

Developing a large-scale software system in C++ requires more than just a sound under standing of logical design issues. Logical entities, such as classes and functions, are like the flesh and skin of a system. The logical entities that make up large C++ systems are distributed across many physical entities, such as files and directories. The physical architecture is the skeleton of the systemif it is malformed, there is no cosmetic remedy for alleviating its unpleasant symptoms. [Lakos, 1996, p. 97]

This instruction applies to any large-scale software system, not just one implemented using C++. When defining the implementation model, it is important to take SCM into account. This is done by having a clear mapping between the high-level design entities and the high-level SCM entities used to manage groups of files.

When using ClearCase, you must establish a mapping between the architecture and ClearCase components. In UML terms, this means establishing a mapping between the logical design packages (implementation subsystems) and the ClearCase components that contain the files that will implement those packages. (The Rational Unified Process defines this as the implementation view of the architecture. See [Kruchten 2000].) The mapping should be at a high level in the architecture, usually the subsystem level; as noted in the RUP formulation, it should include test artifacts. For small systems, there might be only one ClearCase component and no further decomposition. Systems with hundreds of components should not be mapped directly, one to one, to ClearCase components. Instead, you should identify a higher level in the architecture by grouping the hundreds of components into a smaller set of subsystems (refer to Figure 3-3).

For both SCM purposes and good architecture, there must be a high degree of cohesion between internal elements of a ClearCase component and a low degree of coupling between ClearCase components. This is key for successfully mapping architecture to versioned components. David Whitgift explains, "Each item within the hierarchy should be cohesive: It should possess a single defining characteristic that relates its [elements]. The coupling between items in different parts of the hierarchy should be weak; in CM terms, this means that dependencies should be minimized" [Whitgift 1991]. Dependencies here refer primarily to build-time dependencies.

The implementation subsystems, managed in ClearCase components, should have clearly defined interfaces with other parts of the system and should be independently buildable and testable, which allows for independent and parallel development of major parts of the software

5.5. Creating the VOBs

When you have determined how many ClearCase components you will need, you are ready to actually create the project VOB, source VOBs, and the components. This section provides an overview of creating ClearCase VOBs and the use of administration VOBs. It is not a replacement for the ClearCase administration manuals. When actually creating VOBs, refer to these manuals for more detailed information. The purpose of including this information here is to give you a better idea of how ClearCase really works, which will aid in understanding later chapters.

As discussed in the section "The Repository: Versioned Object Base" in Chapter 4, there are two types of VOBs: standard and project. Any VOB can also act as an administration VOB, which stores metadata (for example, branch types or label types) that is shared among a set of linked VOBs. This section provides an overview of how VOBs are created on different platforms and discusses the use of administration VOBs.

5.5.1. Creating the PVOB Using the Command-Line Interface

In this example, you will create a PVOB using the command-line interface. This example uses UNIX syntax, but the command-line interface is also available on the Windows platform. The first step in creating a PVOB is determining where it will be stored. This means both which machine the PVOB server will run on and which disk partition it will physically be stored in. Second, you must determine where in the UNIX file system the VOBs mount point will be created. Unlike Windows, the mount point for a VOB on UNIX can be anywhere in the file system. Typically, VOBs are mounted under a common mount point, such as `/vobs`.

For an example, let's say that you will create a PVOB mounted under `/vobs` with the name projects. The PVOB storage will be on a machine called ccserver1 in a directory that has been exported as `/ccserver1_store1`.

To create the project VOB, you would log onto the machine ccserver1 and issue the following command (all on one line):

```
prompt> cleartool mkvob
  -tag /vobs/projects
  -ucmproject
  -c "My New PVOB"
  -public /ccserver1_store1/projects.pvb
```

Let's break this down:

- `cleartool mkvob` is the ClearCase command.
- `-tag /vobs/projects` is the flag that determines the mount point for the PVOB. That is the point in the UNIX directory structure where end users will go to access the data in the PVOB.
- `-ucmproject` is the switch that indicates that this VOB should be a UCM project VOB instead of a regular VOB that contains only file and directory elements.
- `-c "My New PVOB"` is the comment you enter that will be stored and displayed as the description for this PVOB.
- `-public` is the flag indicating that this PVOB will be used by a group of users on many machines. When using this flag, you are required to enter the public VOB password set by the administrator when ClearCase was installed. This allows the VOB to be mounted

5.6. Baseline Promotion Levels

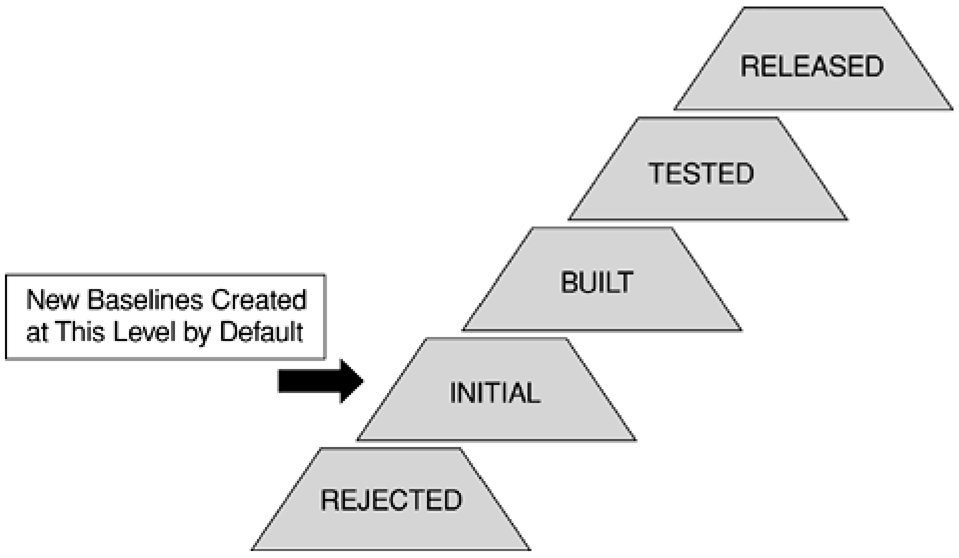
When using ClearCase UCM, you must perform one final administrative setup task: defining the promotion levels for component baselines. A baseline is a single version of a component. The quality or status of that baseline is indicated by a baseline promotion level. All components stored in a project VOB share a common set of legal promotion levels. This is to ensure a consistent definition of promotion level across multiple projects.

NOTE

If different projects cannot agree on a set of promotion levels, they must be stored in separate project VOBs. However, this is not recommended practice if these projects will be sharing components. A common understanding of baseline promotion levels is key to effective project communication.

ClearCase UCM predefines a set of component baseline promotion levels, illustrated in Figure 5-20. Baseline promotion levels are linear. You can move a baseline upward or downward. You must define the promotion level where a new baseline will begin its life. Within the out-of-the-box promotion levels, this is named INITIAL.

Figure 5-20. Predefined baseline promotion levels.



From the command line, you redefine the promotion levels using the command `cleartool setplevel`. See the command-line reference page in the ClearCase manual for details, including the steps on how to rename an existing promotion level.

From the ClearCase Project Explorer, you can redefine the promotion levels by selecting the project VOB top-level folder and selecting the Define Promotion Level menu, as shown in Figure 5-21.

Figure 5-21. ClearCase Project Explorer: defining promotion levels.

Team LiB ◀ PREVIOUS NEXT ▶

5.7. Summary

This chapter has reviewed the fundamental architecture of a ClearCase environment. We discussed each of the server roles upon which ClearCase depends, as well as some possible ways to organize these roles in several kinds of environments. We considered the hardware requirements for these server machines, and we discussed how to approach monitoring and tuning an environment for optimal ClearCase performance. Finally, we showed how to create VOBs and PVOBs and how to manipulate baseline promotion levels. Now a project can be created and development can begin.

Team LiB ◀ PREVIOUS NEXT ▶

Chapter 6. Project Management in ClearCase

This chapter discusses creating, configuring, and managing a ClearCase project and how the SCM project relates to an overall project for teams of various sizes. The ClearCase project policies are also discussed, including when and why you use each of them.

6.1. What Is a ClearCase Project?

Every organization has its own definition of a project. So, what is a ClearCase project conceptually? A ClearCase project consists of one or more people who are working together on activities to produce one or more artifacts. Typically, a ClearCase project represents a large development effort such as a major release or major iteration of a set of components or of an entire software system.

A ClearCase project defines the following:

- Who is making changes
- What is being changed
- How changes are made
- How changes flow from a developer's workspace into a release

6.1.1. Who Is Making Changes?

A ClearCase project organizes a set of people who are collaborating to produce new software component baselines. Depending on the size of the overall development effort, there might be one or more ClearCase projects.

ClearCase project use can be discussed using the five project categories defined earlier (see "Five Project Team Categories" in Chapter 2, "Growing into Your SCM Solution"):

- **Individual** The ClearCase UCM parallel development process might not be appropriate for an individual developer. In this case, the developer can opt to use base ClearCase functionality or can use UCM and work on only one stream. The advantages that ClearCase UCM provides to the individual developer lie in the area of tracking the developer's changes and associating those changes with specific UCM activities. This tracking and association provides specific detailed information that enables developers to determine their progress against their goal and to ensure that the functionality they set out to build is actually implemented. This auditability can be an important advantage if the software is being developed for another organization.

  Working on a single stream in ClearCase UCM is trivial. First, you can do it by conventionthat is, create a UCM project and then choose to work only on the project's integration stream. This mode of work still allows development streams to be created as child streams of the integration stream if that level of project organization becomes necessary. Another way to work on only a single stream is to let UCM enforce that policy. A UCM project can be created as a single-stream project. If a project is so designated, ClearCase UCM will create the project-integration stream and will enforce the policy that no other stream can be created in the project. In either scenario, the developer gets all the advantages that UCM activity tracking provides, but without the overhead of delivering and rebasing activities.

- **Small** A small project team would have one ClearCase UCM project created for each major release of the product it is producing. For teams like this, it is important not to confuse "project" and "product." For example, a project might be Release 3 of product X. In ClearCase, the project Release 3 identifies a group of individuals working together to produce this major release. The team is working on product X, which is physically realized in one or more components. Very small teams, such as individual developers, might determine that a single-stream development model would serve them best. In this case, they can choose to work on only the project's integration stream, leaving the door open to a later expansion of the project into parallel development, or they can define their project as a single-stream project and then let UCM enforce the

6.2. Creating a ClearCase Project

A Project Creation Wizard walks you through the creation of a new ClearCase project. This is shown in an example in the section "Creating Your Project," later in this chapter. However, before you run the wizard, you should understand the decisions you will be asked to make when setting up a new project. The steps you will take are as follows:

1. Identify the project manager for the project.
2. Identify the components and baselines that form your project's foundation.
3. Determine the policies that govern your project's work.
4. Determine the UCM project properties that structure your project.
5. Determine where the physical project object will be located.
6. Create your project.

6.2.1. Identifying Your Project Manager

Someone must create the project and be responsible for it. (The role of the project manager is described in "The Project Manager: Managing a Project," in Chapter 3, "An Overview of the Unified Change Management Model.") For ClearCase projects, the project manager should be a technical individual who understands SCM and the UCM model. Some organizations might refer to this person as a technical lead, project integrator, or configuration manager.

In any case, you must identify the person who is responsible for creating and maintaining the project. This person will make decisions about how the project will be run and what parts of the system will be worked on. This person also will actually create the ClearCase project. (It is possible to have an administrator or tools engineer create the project for another individual. However, after project creation, a number of ClearCase objects must have their ownership changed. Therefore, it is more straightforward for project managers to create projects themselves.)

6.2.2. Identifying Your Components and Baselines

One of the challenges of managing software projects is ensuring that team members are working on the "right" thingswhat pieces of the software system should be modified and what version of those pieces should be used as the starting point for those modifications. As was discussed in Chapter 3, one of the key benefits of UCM is that it uses a baseline + change model. This model provides a framework that ensures that only consistent changes move through the system.

When you start a new ClearCase project, you must decide which components will be needed to perform the activities planned for the project and which versions of those components will be used as the starting point (see "Component Management: Components and Baselines" in Chapter 4, "A Functional Overview of ClearCase Objects," for more information on components).

The components and component baselines define the scope of work for your project, or what components that project must reference and modify. The baselines of the components you select are referred to as the foundation base lines for your project. Specifically, the *foundation baselines* define the initial configuration for your project's integration stream.

6.3. Summary

This chapter discussed UCM projects and all the things you need to think about when you create onethings such as the size of the project team and how that will affect the dynamics of making changes, how those changes are to be made and integrated together, and UCM project policies, such as which components should be modifiable, which your project should simply use in a read-only form, how you want to recommend baselines for development, and much more. Finally, you learned how the UCM Project Creation Wizard guides you easily through establishing the initial properties and policies for your project. Subsequent chapters explore specific project scenarios and workflows, and how they work in a UCM environment.

Chapter 7. Managing and Organizing Your ClearCase Projects

To be competitive in today's market, you need to be prepared to effectively manage your software-development projects. If you don't, you might find your project encountering cost overruns, missed deadlines, low quality, and an incapability to respond to changing market conditions. Perhaps you will even face your project being cancelled. If you manage a small software-development project team (fewer than 5 people), it is fairly easy to keep on top of what the team is doing and how it is progressing toward the project's goals. However, as your software-development project team grows in size, it is harder to coordinate changes among the team, or sets of teams. And in today's world of software development, your software-development team might not only be large, but also geographically distributed across the world. If you can leverage the power of large software-development teams, manage to keep your project on track, and produce high-quality software that meets your customers' needs, you will win the game.

If you cannot keep track of a locally organized software-development project, then surely you will not be able to handle a geographically distributed development project. This chapter discusses large, multiproject development scenarios and other interesting project situations, including IS/IT application development, documentation-oriented project teams, and small project teams. It explores different strategies for organizing these teams and how to manage their integration activities. Chapter 11, "Geographically Distributed Development," extends these concepts to cover geographically distributed development in detail.

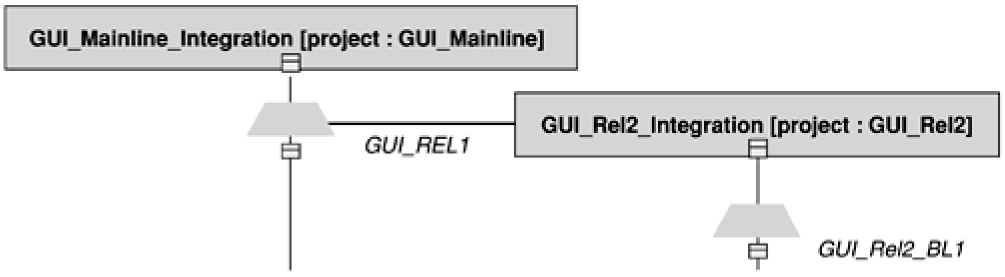
7.1. Coordinating Multiple Parallel Releases

As discussed in "Parallel Development Support" in Chapter 2, "Growing into Your SCM Solution," it is not often that you have the luxury of finishing one release before you begin the next. A project that starts from an existing ongoing project is called a follow-on project. Follow-on projects are supported in ClearCase UCM by creating a new project based on an existing baseline of a selected project's integration stream. To keep changes in sync between these projects going forward, you can rebase or deliver changes between the projects, depending on which direction you want the changes to flow. It is important to understand direction of change flow. For instance, you would not want to accidentally deliver future product enhancements back to a previous release's maintenance project. When you have multiple releasesand, therefore, multiple projects underway at the same time, it is good practice to establish a mainline project that is used to synchronize the source code. A mainline project is a project that serves as an integration and release point for multiple subprojects. Let's take a look at these types of projects.

7.1.1. The Follow-On Project

This section presents an example of creating a follow-on project. Suppose your development team needs to start working on the GUI_Rel2 project (note that this naming convention would indicate that this project is modifying the GUI component for Rel2 work). From the ClearCase Project Explorer, you would create a new project called GUI_Rel2. When asked in the Project Creation Wizard, you would answer yes to seeding the new project from the recommended baseline of an existing project. By selecting the baseline GUI_REL1 (from the GUI_Mainline project), you copy the GUI_REL1 component/baseline configuration to the GUI_Rel2 project. Work can now proceed on the GUI_Rel2 project (see Figure 7-1).

Figure 7-1. Follow-on project initial component baseline graph.


GUI_Mainline_Integration [project : GUI_Mainline]
GUI_REL1
GUI_Rel2_Integration [project : GUI_Rel2]
GUI_Rel2_BL1

7.1.2. The Mainline Project

Regardless of team organization, you want to have a well-understood process of where to start an SCM project. Your teams should also have an idea of how to complete a project. With no guidance in this area, your teams will do whatever they think is right at the time they need to do it. This will lead to some serious problems over time. It is critical to develop a "rules of the road" document to agree on how your organization will behave while performing these operations. One of the ways teams tend to start a new project is by following on an existing project. This section discusses the problems with the cascading follow-on approach and how the use of a mainline project solves these problems. The primary problems with exclusively using the follow-on approach are losing changes, permanent divergence, and SCM tool failures because of infinite cascading branches. When you are managing many parallel releases, patch releases, and many operating system variants of a software system, it can be easy to lose changes or to allow parts of your system to permanently diverge. Most SCM tools use a mechanism to represent the structure of element versions. If you continue to branch off other branches, you set up an infinite cascading scenario. Most SCM tools other than ClearCase will reach some limit in branch naming as branches continue to cascade. Also, some development

Team LiB

◂ PREVIOUS NEXT ▸

7.2. Organizing Large Multiproject Development Efforts

In large development efforts (major and extensive), multiple projects will be working together toward one single release of a software system. In very broad terms, there are two approaches to organizing your project teams: by architecture and by feature. Another pattern of organization in today's world is to get as much utilization as possible out of your software-development resources. This leads to many developers working on multiple projects and releases at the same time. Developers might be pulled off one project to help another, and so on. The organizational approach you take impacts how you do SCM on your project. Generally, over the life of a software system, a hybrid approach is employed.

NOTE

As related to SCM, the differences between architecture-oriented and feature-oriented project teams are significant only when a change affects multiple components.

7.2.1. Architecture-Oriented Project Teams

Architecture-oriented teams are organized along the same lines as the system architecture. There is no substitute for having good system architecture. You can avoid many SCM challenges with good architecture. If you don't have good architecture, it will be hard to organize your teams around architecture. You typically must logically break up your monolithic code base into components as a start, but you will find that your projects still have to change all components. You will most likely organize by feature if you fall into this category. So, in the simple example in Chapter 6, "Project Management in ClearCase," we have one component each for the database, the core, and the GUIs. For a large system, we would assign one team to work on the database component, another to work on the core component, and a third to work on the GUI component. The relationship between components and projects is not always one to one. In more complex systems, a single team might be responsible for more than one component. For example, each GUI interface in an application might be stored in its own component, but a single GUI team might be responsible for changes to any of the GUI components.

Architecture-oriented teams have the following characteristics and effects on SCM processes:

- The teams generally remain the same from release to release (barring major architectural changes).
- Team member knowledge becomes specialized the longer team members work in their product area.
- The integrity of the architecture is easier to maintain than with feature- oriented teams.
- The definitions and stability of component interfaces becomes very important to interteam coordination.
- Iteration planning is more complex because each iteration usually must demonstrate some feature. To demonstrate a feature that spans components, teams must produce baselines that implement pieces of the feature in their components. These components, when integrated, must implement an overall system-level feature.
- Integration is simplified because it involves assembling various component baselines rather than merging changes to common/shared code. This becomes even more

7.3. Coordinating Cooperating Projects: Independent Components

In many circumstances, you might have more than one project team cooperating on producing a final product release. For project teams that are organized architecturally (see the section "Architecture-Oriented Project Teams," earlier in this chapter), it is easy to define producer/consumer component relationships between projects because only one project team is making changes to any given component for any given release.

7.3.1. Project Creation

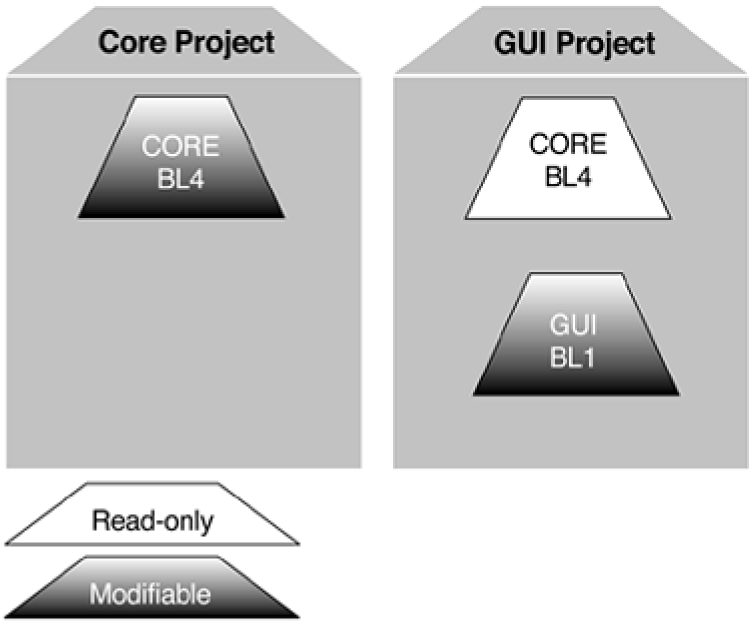
Here is an example. Let's say you have a product with a core component and a GUI component. You are working on producing Release 2 of this product. You have organized your teams architecturally so that you have a core team and a GUI team. For the core team, create a project that will modify the core component. Choose a baseline of the core component you want that project to work onfor example, Baseline 4.

CLEARCASE PRO TIP

When creating the project, be sure to seed it with baselines from the mainline project's integration stream for this component. You will be presented with a list of baselines. You should then pick the baseline from the list that you want to start this project from.

Allow the core component to be modified. For the GUI team, create another project (again, from the mainline project) and include two components: the GUI component and the core component. Pick the same baseline (Baseline 4) for the core component and a baseline for the GUI component. Mark the core component as read-only. You now have two projects whose component lists are illustrated in Figure 7-7.

Figure 7-7. Initial GUI and core project component lists.



7.4. Coordinating Cooperating Projects: Shared Components

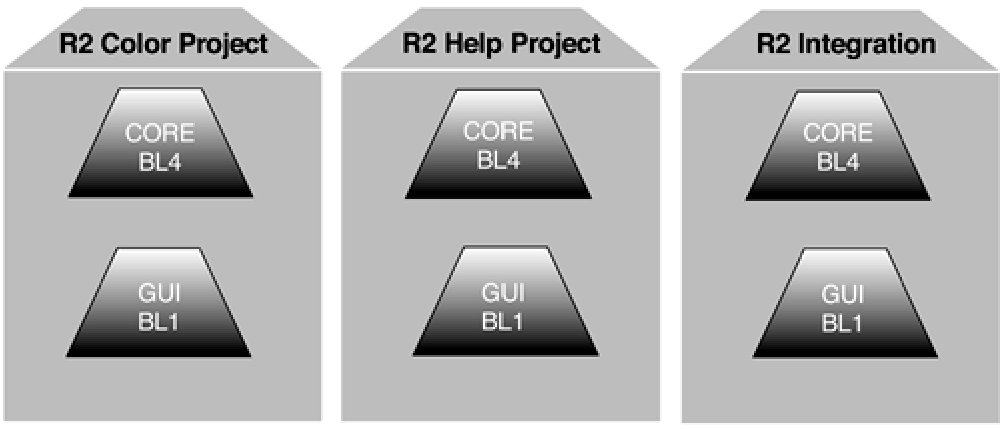
In many circumstances, you might have more than one project team cooperating on producing a final product release. Project teams that are organized around major features (see the section "Feature-Oriented Project Teams," earlier in this chapter) often share components, which leads to the projects modifying the same components in parallel. Creating these projects is easy, but integrating changes is more complicated than in the producer/consumer case discussed in the previous section, "Coordinating Cooperating Projects: Independent Components." It is important to integrate changes early and often. The last thing you want nearing the end of a release cycle is a major integration task among multiple projects.

7.4.1. Project Creation

We'll use the same example as in the previous section. Let's say that you have a product with a core component and a GUI component. You are working on producing Release 2 of a drawing application. Two major features are to be implemented for Release 2: new color-manipulation routines and interfaces, and an online help system.

You decide to organize your teams around features, so you create two projects: Release 2 Color and Release 2 Help. You also create a third project, Release 2 Integration, where changes will be integrated. You now have three projects; their component lists are shown in Figure 7-11.

Figure 7-11. Feature-oriented project component lists.



CLEARCASE PRO TIP

If your feature-based teams are relatively small (three developers or less), you can model this development effort as a shared child stream of the integration project and avoid the overhead of creating and maintaining a new project. Also, you can model subintegration streams, useful for large feature integration, through multiple levels of child streams.

7.4.2. Iteration Planning

Let's look at iteration planning. As part of the work on Release 2 in this example, you must

7.5. Coordinating IS/IT Development Projects

ClearCase UCM is particularly well suited for typical IS/IT development projects, which are often projects being developed by an internal group, where the system being developed is deployed for in-house or internal use. A good example is a trading system for a financial firm. Because the software system is mission-critical for the financial firm, it is developed in-house. It provides a competitive advantage for the company's traders. It is not sold or shipped to other customers. The fact that the system is not sold and that it is deployed to a single internal customer distinguishes these projects from commercial systems. Web sites can also be good examples.

The second thing that is unique about IS/IT projects is that the development group is controlled by the same company that controls the customer group. That means that the customer demands have a stronger influence on the development of the software system than with a commercial system. Commercial systems take in customer input from thousands of customers and must develop products in such a way as to remain profitable. Often in-house systems have only one customer, and the internal software group is held responsible not for making a profit, but for providing the functionality the customer demands as quickly as possible.

The distinctiveness of these projects leads to some unique SCM characteristics, as follows:

- Typically, only one release of a software system is in use at any given time.
- Major release planning and long-term development projects occur with less frequency.
- A procedure for performing emergency fixes is usually employed instead of rolling back to a previous working version.
- An approval process for features is very critical because planning is handled differently from that of commercial projects.
- A feature-oriented rather than architecture-oriented approach is common.

The single release installed and running at any given time is often referred to as the production release. The commercial requirement of maintaining parallel releases in the field is not an issue for these projects.

Parallel development of major releases does not occur as frequently in IS/IT projects. Instead, parallel development occurs on smaller features being added to the production system. There is usually no major release planning; releases occur biweekly, weekly, or, for some organizations, daily. Instead of planning the contents of a release, managers define what goes into the release by what is ready when the release occurs. If a feature misses a release, it just gets put into the next release.

NOTE

These IS/IT systems will undergo some major change at some point, such as a change to of the underlying database, an alteration to the platform the production system is running on, or a major rearchitecting effort. While these major changes occur, parallel development of a planned release is taking place with changes to the current production system. When the new system is ready, a switch-over plan is executed. The new production system is brought online, and the old production system is decommissioned. This is often a major effort for the IS/IT organization because it can involve retraining internal personnel and a Big Bangstyle shift. This type of development is the same as that covered in the section "Coordinating Multiple Parallel Releases," earlier in this chapter.

7.6. Coordinating Documentation Projects or Small Teams

In some cases, the more advanced capabilities in the UCM model are not required. Two examples are projects that are doing documentation and projects that are very small. Projects with two to five developers working on one single software system are considered small (see "Small Project Teams" in Chapter 2 for more details).

This section discusses applying the UCM model to these types of projects. It covers how you use UCM differently than with larger projects and what you give up by using this approach.

First, make sure your project actually falls into this category. The following are the consequences of giving up the UCM features not included in the approach discussed in this section:

- **Developers cannot checkpoint their work**. *Checkpointing* is the ability of developers to check in an intermediate version of a file they have been working on without the changes being made visible to other team members. In the approach described here, the check-in operation makes the change available to other project members. If this presents a problem for your developers, this approach is not for you.

- **Development workspaces are not fully isolated**. In this approach, if developers are using dynamic views, they will see changes made by other developers when those developers check in changes (not when they deliver). This can be desirable for very small teams (fewer than five members) because it encourages early integration. If your developers are using snapshot views, they will still be isolated until they perform a Snapshot View Update operation. If you want your developers to fully control when they see changes, you should not use the approach described here.

- **Development workspaces might become inconsistent**. UCM ensures that changes are moved around and made visible as a whole. That is, all changes made for a given activity become visible at the same time. The approach described here does not maintain this feature. Instead, files are made visible to the team one at a time as they are checked in, introducing the possibility that another developer will pick up a new version of one file but not the corresponding new version of another. Therefore, if multiple people are working on the same file or files, version skew can occur, creating an inconsistent development workspace. If your development team members frequently work on the same sets of files at the same time, the approach described here is not for you.

- **Deliver and rebase operations are not performed**. The approach described here involves everyone working in the integration stream. Because no development streams are in use, deliver and rebase operations are not performed. This is a side effect of the lack of developer isolation, as just described. You can avoid this situation if, instead of working on the integration stream, you have the team work on a shared development stream. This enables developers to deliver completed changes to the integration to be picked up by QA.

- **The entire system should be contained in one VOB**. If you are considering this approach and you are managing a system that cannot be contained in one single VOB, you should consider using full UCM. However, if you have a system that can be contained in one VOB component, this approach can reduce some overhead (but at a cost, as described in the previous points).

CLEARCASE PRO TIP

7.7. Summary

This chapter touched on many key organization issues surrounding software development. Your organization should understand where to create a new project, how to finish the project, and how to organize your components and your team of developers to work on those components. One of the most important things you can do as a development organization is to agree on a common usage model for your development project. Different projects might use slightly different usage models, but all the team members should understand how to interact with the SCM system based on the role they are playing and the project they are working on. You should also invest the time to review how the usage model is working with the key stakeholders of the teams. If you cannot achieve this, you will not be an optimized development organization, and you will have to rely on heroic efforts more often at crucial stages in your product's life cycle.

Chapter 8. Development Using the ClearCase UCM Model

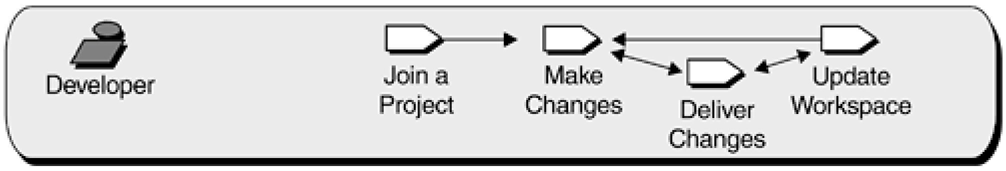
The goal of any software-development effort is to create high-quality software that is on time and under budget, and that also satisfies the agreed-upon requirements. The last thing you want your SCM tool to do is make it hard for your developers to do the right thing. You want a tool that is easy to use, that integrates seamlessly with your developer's IDE, and that, for the most part, stays out of the way of your developers as they perform their tasks on the project. ClearCase Unified Change Management (UCM) provides the process guidance and bookkeeping to allow your developers to focus on software development, without having to waste time wondering whether they are performing changes against the right code base, whether they are up-to-date with other developer's changes, and where they should integrate their changes. If you are using ClearQuest-enabled UCM, you can streamline your development effort further by assigning activities to your developers to work on. These activities will show up on your developers' to-do list, making it even easier to track progress toward your development goals.

This chapter explores UCM for the developer. After reading this chapter, you should understand how to work on an existing project, make changes to files to accomplish an activity, deliver the changes associated with one or more activities, and rebase your development stream to integrate in your development view changes made by other developers on your project.

8.1. A Developer's Perspective of UCM

As a developer using ClearCase UCM, you must understand what project or projects the activities you are working on are destined for. Figure 8-1 shows the developer's workflow (see "ClearCase UCM Process Overview" in Chapter 3, "An Overview of the Unified Change Management Model," for more details).

Figure 8-1. Developer process flow.



Before you can make changes to code, you need to establish your working environment (your development view). ClearCase does this automatically when you join a project. When you have your view created, you can make changes to accomplish a specific activity (or task). This is done by checking out elements, making changes (editing files), and checking the elements back in. As elements are modified and checked in, new versions are created in the VOB and associated with the activity you are working on. These versions are referred to as the activity's change set.

When you have finished implementing and testing the changes you have made, you deliver your changes to your project's integration area. This is done by performing a deliver operation and specifying the activities you want to deliver. The versions of elements that get delivered are dictated by the selected activity's change set. On a periodic basis, you update (rebase) your workspace (your development stream and development view) with changes that have been made by other developers on the same project. Some development projects require you to rebase to the latest recommended baseline from the integration area before performing deliver operations. In the general case, you can also deliver from child streams or other project streams into your stream to integrate changes before delivering them into your project's integration stream. These integration strategies are discussed in detail in Chapter 9, "Integration."

The following sections describe these steps in more detail.

8.2. Working on a Project

ClearCase UCM organizes development teams into projects. For example, a project might be created to develop Release 2 of a software system. You join a project to participate in that project. Joining a project results in the creation of two logical workspaces: one for doing development and one for integrating your changes with other developers' changes. As you read in Chapter 4, "A Functional Overview of ClearCase Objects," a logical workspace is implemented with two ClearCase objects, a view and a stream.

ClearCase UCM provides a Join Project Wizard to walk you through getting started. This GUI wizard is available on both Windows and UNIX platforms. (It is also possible to completely script this step by using the command-line interface and by making all setup decisions.) The Join Project Wizard is available in the ClearCase Project Explorer and the UCM Panel from the ClearCase Explorer, shown in Figure 8-2. It is also available in IDE integrations with ClearCase.

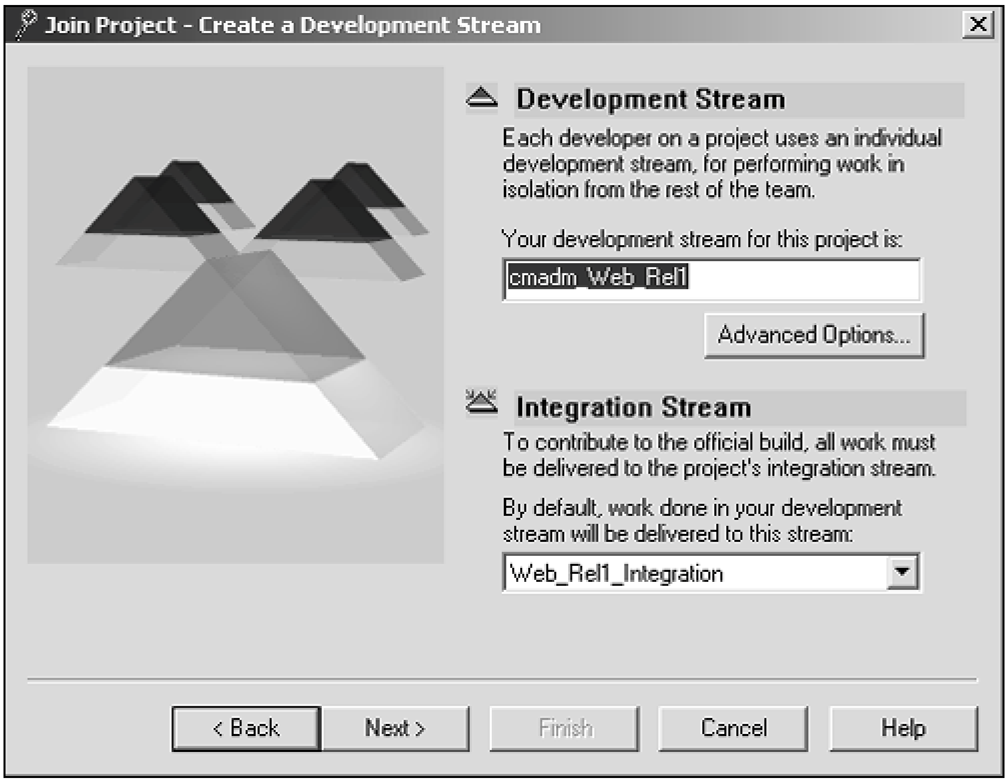
Figure 8-2. ClearCase Explorer: UCM toolbox panel.

[View full size image]

When you join a project, ClearCase creates three new objects: a development stream, a view associated with your development stream (development view), and a view associated with the project's integration stream (integration view). You use your development view for doing development and your integration view for delivering and integrating your changes. Figure 8-3 shows the Join Project Wizard during the creation of your development stream. Figure 8-4 shows the Join Project Wizard during the creation of your views.

Figure 8-3. Creating your development stream.

[View full size image]



Each developer on a project uses an individual development stream, for performing work in isolation from the rest of the team.

Your development stream for this project is: cmadm_Web_Rel1

To contribute to the official build, all work must be delivered to the project's integration stream.

By default, work done in your development stream will be delivered to this stream: Web_Rel1_Integration

Figure 8-4. Creating a development and integration view.

Team LiB

PREVIOUS NEXT

8.3. Making Changes

The real work happens when you modify files and directories to implement a specific activity or task. The following sections discuss how changes are made in the context of the UCM model.

8.3.1. Working with Activities

One aspect of the UCM model is that it is activity based (see the "Activities" section in Chapter 4). This means that all the versions you create when modifying elements must be associated with some named activity. You usually do this at check-out time (you can also specify a new activity at check-in time).

If your project is using only ClearCase (that is, you are not working on a ClearQuest-enabled project), you create these activities when you begin working. They identify the purpose for your work. ClearCase activities are designed to be lightweight and involve low overhead. If your project is using ClearCase and ClearQuest, the UCM model uses ClearQuest entities (such as defects and enhancement requests). Therefore, activities will usually already exist, be assigned to you, and be available on your to-do list when you start work. The complexity of an activity when ClearQuest is in use depends on the processes put in place for that activity type. See Chapter 12, "Change Request Management and ClearQuest," for more information on integrating ClearCase UCM with ClearQuest.

Either way, your responsibility is to ensure that the changes you make to a file are recorded against the appropriate activity. If you don't, the real benefits of activity-based CM cannot be realized. You will not be able to deliver complete, consistent changes automatically. You will not be able to see accurately what changes are included in an update to your workspace. Testers will not be able to see easily what changed from one build to the next. Release notes cannot be generated accurately and automatically.

It is up to you, the developer, to ensure that the change set is accurately recorded against the correct activity. ClearCase UCM has made this process straightforward, using either the Check Out dialog box or the command line. In either case, the activity you select is remembered and used as the default for all subsequent check-outs until you change it.

WARNING

The most common mistake made when associating changes with activities is to make two different changes (for example, two different defect fixes) in a file and then check in that file. In that case, you have just included two different changes in one element version. The UCM model does not allow you to associate two activities with one file version, so you would inaccurately record that this version includes changes for only one activity. The appropriate approach is to make one change, check in the change, check out the file again (specifying the second activity as the reason for this change), and make the second change, thereby associating each check-out with the appropriate activity.

(Some SCM systems do allow you to associate two changes with a single version. However, after this is done, these two activities [changes] cannot be separated in an automated fashion. It becomes impossible to automatically include one of the changes in a build without the other. Because of this issue, the UCM model does not support associating multiple changes with one file

8.4. Delivering Changes to the Project

At some point, you will complete work on one or more activities. Because you are working in isolation, you need to take additional steps to make the changes you have made available to the project integrator. The process of making your changes available is called a delivery.

Delivering a change is a multistep process:

1. Check in any outstanding checked-out elements.
2. Optionally rebase from the project's latest recommended baselines. If you do this, you need to build, test, and debug changes related to this integration effort.
3. Run the ClearCase `Deliver` command.
4. Build and test the delivery.
5. Complete or cancel the delivery.

After delivery, your changes can be incorporated into the next project baseline and project-level build. The steps that make up a delivery are described in the following sections.

8.4.1. Check In Any Outstanding Checked-Out Elements

When you deliver an activity, only the latest checked-in element versions are delivered. Remember that you can check in multiple times without your changes being seen by other developers. You must check in changes to deliver them. If you deliver an activity and it has outstanding check-outs, the changes in the checked-out files are not delivered.

You can search for outstanding check-outs in a number of ways. First, the deliver operation warns you that you have outstanding check-outs (see Figure 8-10). Second, you can use the Find Check-Out Wizard GUI to list check-outs. Third, you can go to the activities change set either from the ClearCase Project Explorer or ClearQuest (if your project is using ClearQuest) and check in from the Unified Change Management tab (see Figure 8-11).

Figure 8-10. Deliver outstanding check-outs warning dialog box.

8.5. Rebasing Your Development Stream

Your development view selects a stable set of element versions. Periodically, you need to update your development stream's configuration, thus updating the versions of the elements displayed in your view. This is done using a command called `Rebase`. Rebase updates your development environment, making changes other developers have made visible to you.

When you rebase, you do not get the latest and greatest element versions. Those versions might be broken or might not even build together (let's hope not, but it is a possibility if other developers are not following good process when delivering their changes). Instead, you receive a stable set of baselines. The project integrator creates new project baselines and builds and tests them. When a set of baselines has reached a known level of stability (generally, this means it has passed some level of testing), the integrator declares the baselines as the recommended baselines (see Chapter 9 for more detail on the integrator's work). The recommended baselines are presented to you as the default when you rebase.

NOTE

It is possible to override these default baselines, choosing either older or newer baselines. However, it is a good idea to accept the recommended baselines until you get familiar with ClearCase UCM.

Rebasing is a multistep process, as follows:

1. Run the rebase operation.
2. Build and test.
3. Complete or cancel the rebase.

These steps are described in the following sections.

8.5.1. Run the Rebase Operation

The ClearCase `Rebase` command can be issued from the ClearCase Explorer (see Figure 8-2), the ClearCase Project Explorer, or a number of IDEs, such as Rational Application Developer. A rebase can also be started from the command line using this command:

```
prompt> cleartool rebase
```

When you rebase, you are updating the baselines you see in your development view. These baselines, of course, contain new activities delivered by you and other members of your project. You can see this set of activities and explore the changes you are accepting into your development stream by clicking Details on the Rebase dialog box (see Figure 8-13). If you are using Rebase from the command line, you can do this by using the following command:

```
prompt> cleartool rebase -preview
```

Figure 8-13. Rebase activities: activity details.

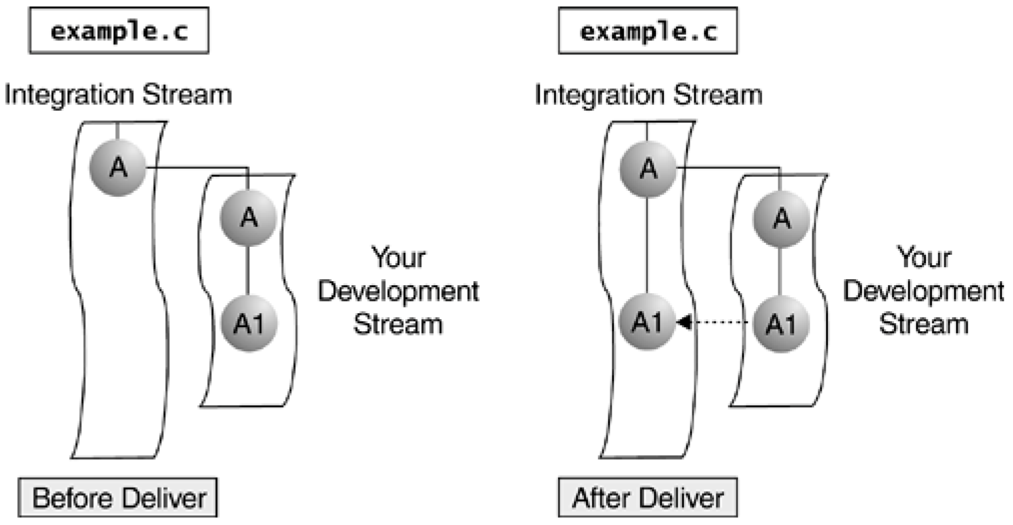
8.6. Dealing With Conflicting Changes

When working in a serial development environment, only one person is allowed to change a file at a time. As discussed in Chapter 2, "Growing into Your SCM Solution," this approach can cause development bottlenecks and make it very difficult to maintain multiple releases. ClearCase UCM supports parallel development, which means that while you are working in your development view, you might be modifying the same files at the same time as another team member in his or her development view. Obviously, these changes must be merged or integrated at some point. With ClearCase UCM, this might happen during deliver and rebase operations. ClearCase provides specific tools to automate as much of this merging work as possible, and tools to assist you in integrating changes when conflicts occur that cannot be automatically resolved.

8.6.1. Delivery Scenario 1 (No Conflicts)

Conflicts occur only when more than one developer works on the same file at the same time. If this never happens, you will never need to resolve conflicts. Figure 8-15 shows a delivery scenario in which no merging is required. You have made a change to `example.c`, but no other member on your project team has modified the file. When you deliver, ClearCase simply copies the contents from your development stream to the integration stream. In ClearCase terms, this is called a *trivial merge*.

Figure 8-15. Trivial delivery scenario (no conflicts).



8.6.2. Delivery Scenario 2 (No Conflicts)

This case is similar to delivery scenario 1. No conflict resolution is required as part of delivery, and ClearCase performs a trivial merge. However, in this case, another team member did modify the file `example.c` and, in fact, delivered his or her changes before you. The difference from scenario 1 is that you performed a rebase operation before doing your delivery. The arrow shown in Figure 8-16 pointing from the integration stream to your development stream indicates that a rebase operation was performed. This is also a trivial merge for ClearCase.

Figure 8-16. Trivial delivery scenario post-rebase (no conflicts).

8.7. Seamlessly Integrating with Developer's IDE

You have seen how the UCM development model makes it easy for developers to keep track of their work at a higher level. This gives development managers a better understanding of how their software-development project is progressing. What defects are fixed? Who is working on these defects? Why does this component have so many defects? What has changed between baselines BL2 and BL1? The UCM development model can easily answer these and many more questions. UCM has allowed the developer to focus on development, without having to worry about tedious bookkeeping or maintaining a home-grown layer on top of a CM tool. Another key attribute of ClearCase UCM is its tight integration into the IDEs of choice. This means that developers never have to exit their IDE while performing common ClearCase UCM tasks. Figure 8-24 shows the Eclipse platform's integration to ClearCase. At the top (enclosed in the bold rectangle overlay) is the ClearCase context-sensitive toolbar integration that enables developers to quickly operate on the element that is in focus. In the case of Figure 8-24, it is Logon.java. Another feature of the tight integration to Eclipse is the ClearQuest perspective. The ClearQuest Navigator on the right hand side is showing ClearQuest's query workspace. From here you can run ClearQuest queries, charts, and reports. The ClearQuest Query Results view (in the lower middle window) is presenting the resulting records from the "all defects" query. You can perform many ClearQuest actions and query executions without having to go outside your Eclipse environment.

Figure 8-24. Eclipse Platform: ClearCase toolbar.

[View full size image]

Another ClearCase integration point to Eclipse is through the standard right-click action on the element in the Package Explorer in the left window. This context-sensitive integration is exposed through the Team menu, as shown in Figure 8-25 (showing the check-in operation because `Logon.java` is in the checked-out state, indicated by the green check mark on the element).

Figure 8-25. Eclipse Platform: ClearCase team integration.

[View full size image]

ClearCase also provides a ClearCase menu integration to Eclipse. This integration point provides access to ClearCase operations that affect the workspace rather than the element. Some of these operations are also available from the ClearCase toolbar. Figure 8-26 shows the Eclipse Platform ClearCase menu integration with the Deliver option highlighted.

Figure 8-26. Eclipse Platform: ClearCase menu.

[View full size image]

8.8. Summary

This chapter presented the basic developer workflow with ClearCase UCM. It highlighted how ClearCase UCM keeps track of activities and manages streams, and how a developer interacts with those objects on a daily basis. It also showed how easily these powerful features are exposed to developers in various ways: through command-line, GUI, and IDE integrations. ClearCase UCM has made it very easy for the developer to do the right thing while working on the daily tasks of modifying software-development artifacts. ClearCase UCM not only makes this development experience easy to use and understand, but it also makes it easy for project managers to track progress of their development projects by providing insight into the activities that are under development. By integrating ClearCase UCM with ClearQuest, you get even more power to model your software-development process by enriching activities to be defects, enhancement requests, or any other type of development entity you want to model. This chapter also touched lightly upon integration issues by showing how ClearCase UCM handles merging of activities to project-integration areas. It showed merge algorithms and logical concepts of how changes move between streams during deliver and rebase operations. This is the foundation for understanding how to model more complex and larger software-development efforts because, at some level, the developers will act in this fashion within their development streams. Chapter 9 introduces more complex integration strategies that arise in software development across the team and across the organization.

Chapter 9. Integration

To integrate means to form, coordinate, or blend into a functioning or unified whole. Integration within software development is the act of bringing together separate efforts into one. During software development, there are many forms of integration, and they can occur at different places within the software development life cycle and at different times. For instance, before you can expose your software-development effort to another group for testing and verification, you need to make sure that you have all your features and defects together in a place where they can be built and assembled into whatever form your application or product needs to exist in before it can be tested. It is critical to define within your software life cycle when these events will take place. Before integration, you have many different changes developed in some sort of isolation. After successful integration, you have a system that contains these different changes, and the system functions against the desired requirements at some quality level. If you were forced to deliver something to your customer, it would usually be the most recent highest-level quality configuration you had, depending on the particular customer milestone you were meeting. Integration is the first step toward the validation of your system.

Many people participate in the integration activities. However, there is usually a role for the individual who performs the integration activity that will bring together the changes from the individual efforts into a functioning whole. It is the job of the integrator to use the work of the team to construct a single version of a software system or set of software components. It is also the integrator's job to assemble the work of the many teams into a single version of a software system that can be tested against a set of requirements and ultimately deployed or released.

This chapter discusses merge and assembly, two types of software integration, and their application to the categories of software teams (see the section "Five Project Team Categories" in Chapter 2, "Growing into Your SCM Solution"). It also discusses how ClearCase branches are used for isolation and integration, using either your own branching strategy or ClearCase UCM. Build, staging, release, and deployment processes are covered in Chapter 10, "Building, Baselining, and Release Deployment."

9.1. Software Integration

Software integration is the process of bringing together independently developed changes to form a testable piece of a software system. It can occur at many levels, eventually culminating in a complete software system. The larger the software system is and the larger the teams working on that system are, the more levels of integration are required to manage the software-development effort.

It is always cheaper to find problems as early as you can in the software development life cycle. If you find problems at the customer site, it is more expensive to fix. If a developer finds an issue during integration activities, he or she can fix it much more cheaply because fewer people, systems, and processes are involved. Another reason to perform integration activities sooner is that usually the problems found during integration are harder to solve (because they might involve many pieces of the system). If you delay integration, you could be delaying the hardest-to-fix problems, which might end up delaying your product release.

Basically two types of integration are relevant to SCM: merge and assembly integration.

9.1.1. Merge Integration

Merge integration involves the resolution of parallel changes made by different team members to common files or components. In this case, multiple people have modified the same set of system artifacts in parallel. Therefore, it is necessary to combineor, in ClearCase terms, mergethese changes. In some cases, this can be automated with tools that understand the structure of the files. In other cases, the merge must be performed manually (for example, if there are conflicting changes). It should be clear that merge integration requires some knowledge of the changes being made to the software system. In addition, more changes might be introduced to the software as part of performing merge integration (for example, changes required to resolve merge conflicts).

9.1.2. Assembly Integration

Assembly integration involves combining baselines of software components into a larger piece of the overall system. The Rational Unified Process defines integration as "the software development activity in which separate software components are combined into an executable whole" [RUP 5.5 1999]. Unlike merge integration, assembly integration does not modify the source code; it puts together the puzzle pieces of the software system (hopefully, they all fit).

Assembly integration can occur at build time or runtime or both. With build-time assembly, you bring together two sets of source components, build them, and then link them together to form a testable executable. With runtime assembly, you copy a set of prebuilt objects into a runtime environment, which can then be executed. A set of dynamic link libraries (DLLs) from two different software components is a good example of runtime assembly integration.

The type of integration and the number of integration levels used are largely determined by the size of the software system and the size of the team. Integration choices also depend on whether the teams are organized around architecture or around features (see "Organizing Large Multiproject Development Efforts" in Chapter 7, "Managing and Organizing Your ClearCase Projects"). At some level, a system that has well-defined software architecture uses assembly integration. A monolithic system is more likely to use merge integration all the way to the top.

9.1.3. Integration Scenarios for Teams of Differing Sizes

Let's take a look at some integration scenarios based on the team sizes defined in the section "Five Project Team Categories" in Chapter 2.

9.2. Isolation and Integration with ClearCase

Developer isolation occurs when developers are working in stable workspaces where changes other developers are making do not affect them and where changes they make do not affect other developers. Similarly, teams can be isolated from other teams. Isolation is used to manage complexity. Problems or new features are divided up across teams and individuals. To work efficiently, teams and individuals isolate themselves from others' changes that could destabilize their workspace.

For example, Developer A might be making a change that requires alterations to a common interface as well as underlying code. If you tried to build and test the system after Developer A changed the interface but before Developer A changed any calls to that interface, the system would be broken. So, if Developer B was not isolated from Developer A's changes, Developer B might be blocked from proceeding even if Developer B was working on a completely separate part of the system.

When you isolate work, at some point, you must integrate. The difficulty is deciding when and how often this should take place: the developer isolation versus project integration dilemma. You are balancing a stable development environment against a better understanding of whether the pieces of the system being developed independently really work together. Infrequent integration means that you discover integration issues late in the project, causing significant and unplanned redesign work, which often results in a missed project deadline. Integrating too often causes unnecessary lost productivity. A destabilizing change that is integrated too early can cause an entire team to be unable to complete its work on time. Changes likely to cause destabilization should be isolated and unit-tested before being integrated. Deciding when and how often to integrate is how you tune the performance of your SCM process.

An improper approach to isolation and integration is probably the no. 1 cause of SCM-related problems on a software project. How you apply your SCM tool does have a serious impact on how well your development organization performs. ClearCase supports a number of integration approaches, which are described in the next few sections. Only you can pick the right approach for your projects. This section covers the topic of using branching and merging to implement your own integration strategy and how the ClearCase UCM model supports integration. We begin by discussing two interesting integration approaches: no isolation and branch/LATEST development.

9.2.1. The Shared View: No Isolation

This section on shared views is included only for completeness; it is not recommended that you use shared views for development. The only place I've seen them used is for browsing purposes only (such as a build or release view that might contain derived objects). A view is typically used by one individual. However, when you desire no isolation between team members, multiple individuals can share a single ClearCase view. The best way to think about this is as a single copy of your source tree in which everyone works at the same time.

Everyone working in a shared view is isolated from each other's changes only while a change is being made in an editor. Any commitment of a change to disk (such as a save) makes the change visible to others working in the shared view. Basically, this means that you have almost no isolation between team members and that integration occurs automatically and constantly. When a file is checked out, it becomes writeable by anyone working in that view. ClearCase checks out files to a view rather than an individual. Because the view is shared, anyone has access to it. Thus, developers cannot work on the same file in parallel.

If you use shared views as a general-purpose solution, you will experience many of the problems encountered in early SCM systems, as described in Chapter 2.

9.3. Summary

This chapter showed many ways to integrate software with ClearCase and ClearCase UCM. No matter what strategy you end up using, you should plan for regularly scheduled integration activities during your software development life cycle. During integration of the system, you tend to find problems that affect the entire system, which are harder to resolve. The sooner you can find these problems, the sooner (and cheaper) they can be fixed. Some companies continuously integrate and build; they want to find integration issues as soon as they can. If you delay integration, integration activities will take longer and require heroics from your development organization to not let the schedule slip because of integration issues found late in the development life cycle. You can use streams to model iterative development, which promotes integration more frequently, or use them for feature, bug fix, patching, and so on.

You should architect your system so that it can be worked on by component development teams, which produce baselines that are assembled by a system-level assembly project. This allows for teams of people to easily consume new changes from other components and perform integration activities sooner, verifying new functionality. This also makes integration much easier and more scalable to larger systems. After integrating or assembling changes from various subsystems, the fastest way to get a warm, fuzzy feeling on how well this new configuration will work is to build and test the system. The next chapter focuses on building, testing, and release/deployment.

Chapter 10. Building, Baselining, and Release Deployment

In the last chapter, you saw how to integrate the changes produced from the various development teams. To make sure your system still works, you need to build and verify your software. If there is a problem during this stage, you can reject this build attempt, triage the problem, and work at resolving the issue. No other external organization will have been exposed to the problem because it would have still been using the previous configuration from your team. This verification step is needed before you can expose others to your software.

It's not just your customers you need to protect from these release candidates that do not achieve a certain quality level. Time is critical in the software development life cycle. If you allow your development organization to haphazardly throw release candidates over the wall for the testing organization to test, you might find that your testing group is spending time setting up its testing environment, securing resources to perform the testing (both hardware and people). When it finally performs the tests, it fails because of issues that could have easily been discovered during a testing effort within the development organization. Now, not only has this testing effort been wasted, but the resources that performed the effort also have been wasted by not allowing them to be available to test other products. As you can see, a small oversight in this process has a major impact on the development organization.

Mistakes happen. You can limit and reduce the chances of mistakes occurring in your development process if you have a documented and repeatable build and release process. In today's world, where audits of development practices are becoming more the norm, your development organization needs to establish best practices within the development, testing, and deployment processes. Every release needs to be tracked back to the configuration that built it. From that configuration, you should easily know which features and defect fixes went into the release. And from those features and defect fixes, you should know what customer, marketing, or government requirements you are trying to address. Haphazard development and release processes will only get you into trouble. You should spend the time and money up front to address these issues within your organization before they become major problems.

This chapter explores baselining, building, and release-deployment strategies and how ClearCase and UCM fit into those strategies.

10.1. Baselining and Building with UCM

In your development organization, developers perform a build to verify their changes. They might perform another build after they integrate their changes with the rest of the team (see Chapter 9, "Integration," for more details on integration). However, the build process that is performed in the integration area for the product is usually performed by an integrator, project lead, or release administrator. This person is responsible for building, baselining, and smoke-testing the software. This could be a part-time job for a development lead on a small project or a full-time job on a much larger project, or it might be performed by several individuals. To get a stable configuration in place, you want your release area baselining and build process to be documented and, ideally, automated so that you can have it run unattended and at times when the local development team is not working (if possible). During this time, changes to the integration area will not be allowed. Developers can still perform changes against the next iteration or the next release while the current release candidate is being built.

With ClearCase UCM, the integrator needs to perform the following steps in the product-integration area to ensure a reproducible configuration.

1. Lock the integration stream.
2. Baseline the components.
3. Build.
4. Execute any smoke tests available.
5. Promote the software component baselines to the desired state.
6. Optionally recommend the baselines for use by the project.
7. Unlock the integration stream.

The following subsections discuss each of these steps and the variants of this sequence, including how to automate it.

10.1.1. Locking the Integration Stream

The first step is to lock the integration stream. This can be carried out most easily from the ClearCase Project Explorer (although, when automating a build system, this step and all steps that follow will most likely be scripted through the use of cleartool commands; automating the build system is discussed later). Figure 10-1 shows how to bring up the properties sheet for the integration stream. Figure 10-2 shows the properties sheet of the integration stream for the Web_Rel1 project. The Lock tab is displayed. Make sure that you exclude yourself from the lock. As shown, the user cmadm is still allowed to make changes in the integration stream. Locking the integration stream stops deliveries from occurring so that you can baseline and build against a stable code base.

Figure 10-1. Bringing up the properties sheet from the ClearCase Project Explorer.

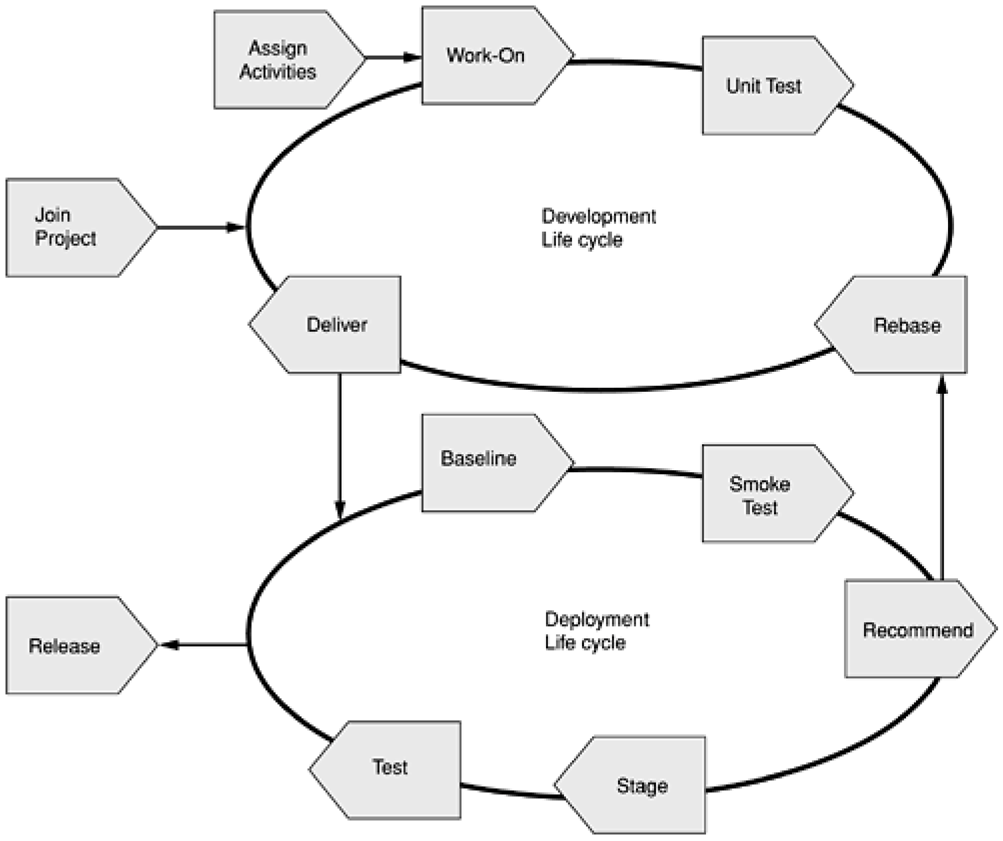
[View full size image]

Figure 10-2. Locking the integration stream from the properties sheet.

10.2. Staging, Deployment, and Release

Up until now, we have talked mainly about versioning of source code. Two other very important processes with respect to SCM are staging and release. *Staging* is the process of putting derived object files (executables, libraries, data files, generated header files, and so on) under version control. *Release* is the process of putting the runtime software into its final form and making it available to its intended users. These staged artifacts typically get *deployed* to various systems for further testing. They also get deployed to the production environment. In this case, the system gets released when it has been deployed to production. So, for some development shops, a release might be deploying the artifacts to manufacturing where CDs get burned and shipped out to customers, or are available for download. In other shops, a release is deployed to production, where it is brought online and is serving the needs of its customers. In general, the UCM development and deployment process is shown in Figure 10-10.

Figure 10-10. The general UCM life cycle. Source: [Brown, 2004]



The primary purpose of staging is to store copies of the executable or runtime parts of the system so that they are secure and can be reliably recalled. The reasons behind staging and how staging is performed can be very different, depending on the type of software system you are developing. In fact, it is very hard to generalize the process of staging.

Most companies model their UCM baseline-promotion levels to reflect how the software artifacts move through their development and deployment life cycle. By integrating ClearCase UCM with deployment tools, you can have the promotion of a baseline trigger a deployment of the software artifacts from the UCM staging components to the desired test or production environment.

The primary purpose of release is to make the software available to its end users. The act of

10.3. Summary

In this chapter, you have seen how typical build and release systems can be put together with ClearCase UCM. A lot of what your organization will do in this area depends upon the type of software you are developing and the frequency of the releases. You also need to consider the time it takes to build your system and any deployment considerations.

By defining a release model and implementing it within ClearCase UCM, you can explicitly understand the requirements and checks and balances in the process. You will understand what it means to promote a baseline from one state to another. And if you understand this, you can explain it to any auditing agency that will want to know in detail about your development and release practices. When you have your integration build, release build, and baselining process documented and working, you can look to automate it to schedule it more frequently or perhaps during off-hours. Hopefully, by putting time and energy into this area of your software development, you will be able to release higher-quality software to your people testing it and, eventually, your customers using it. In the end, this will allow you to either decrease your software-development costs or be able to deliver more releases in the same time frame.

Chapter 11. Geographically Distributed Development

Geographically distributed development is the development of software systems by team members w0ho are not located in the same geographic region. This could mean teams distributed at different sites in the same city or in different countries around the world. Although this presents a number of challenges, many companies are finding sound business reasons to do it, including the global nature of their own company, use of third-party components, coordination with third-party software houses, and mergers and acquisitions.

A whole book could be dedicated to this topic. This chapter briefly discusses the organizational, communication, and technical challenges that need to be overcome for success in distributed or geographically dispersed development. This chapter also covers the support for distributed development provided by ClearCase and an add-on product, called ClearCase MultiSite.

This chapter then explores how best to apply ClearCase to three common distributed development scenarios:

- **Multiple teams: producer/consumer** Multiple project teams are located at different sites that share software components in a producer/consumer relationship. The consumer does not modify the components delivered by the producer.
- **Multiple teams: shared source code** Multiple project teams at different sites are modifying the same shared software.
- **Single team: distributed members** A single project exists with team members at different sites working on shared software components. This differs from the previous two scenarios in that the distributed members are not organized into a remote team. Instead, there are many individuals working remotely (perhaps from home).

The chapter finishes with a brief discussion of other uses for ClearCase MultiSite.

11.1. Distributed Development Challenges

Developing a single software system with distributed teams is no easy task. Organizational, communication, and technological issues must all be addressed to succeed. This section covers these issues in general. Later sections cover specific approaches to doing distributed development and discuss how each approach relates to these three challenges.

11.1.1. Organization

Organization deals with how team members are grouped into projects, who is responsible for leading the team, how multiple teams are interrelated, who is responsible for making project-wide decisions, and, who ultimately, is responsible for the success or failure of the project. There are probably an infinite number of ways to organize development efforts into projectsan infinite number of ways to assign team members to these projects, to distribute those team members around the globe, and to divide the work among team members. Interestingly, the managerial structure within an organization might not reflect the project team organization. The first issue a project manager faces is how to organize the available members of the development staff into projects.

In distributed development environments, often there are different cultures and different development styles. This is easily seen between sites located in two different countries, but it is also true within a country. These cultural differences introduce additional and sometimes subtle obstacles to attaining success.

These key facts need to be determined and understood by all team members:

- Who is responsible for the overall success of the project?
- Who is responsible for the managerial issues the project encounters?
- Who is responsible for the overall system architecture?
- Who are the team members on this project?

Distributed development efforts tend to be large in scope. So, in general, there will be many smaller projects, with the teams from each collaborating on an overall project, which we'll refer to as the "superproject." All members of the collaborating teams should be able to answer the questions just listed about the superproject.

In our experience, many distributed development projects fail because they do not establish this superproject organizational structure between teams working at two or more sites. Without this infrastructure, the tendency for the project teams is to make independent architectural and technological decisions. Ultimately, this makes the system more difficult or impossible to integrate. Even if integration can be done, the lack of overall direction could make project boundaries visible to the end users through the look and feel and other subtle behavior of the resulting system.

Smaller projects that have team members that are geographically dispersed might be able to get away without the superproject organizational structure, but the requirement for rapid and robust communication among team members remains.

11.1.2. Communication

The second challenge of distributed development is communication. Because the teams are geographically separated, communication is impaired. Even today, with e-mail, instant messaging, fax, voice mail, and videoconferencing, two teams that are not at the same location have far fewer communication channels. The value of the community environment

11.2. How ClearCase Supports Distributed Development

ClearCase supports five technological solutions for remote developers. Each approach has its own characteristics that make it the right choice in different scenarios. The five options are as follows:

- **Remote terminal or desktop access** Logging into the primary site from a remote location (for example, home)
- **Remote client access** Using client software to access the primary site from a remote location
- **Web access** Using the Web and HTTP protocol to access the primary site
- **Disconnected use** Working disconnected from the network with a local view
- **Local access** Copying VOBs and keeping changes in sync so that all work is done locally, regardless of site

11.2.1. Remote Terminal or Desktop Access

Remote terminal or desktop use is simple and can be accomplished in one of two ways. The remote developer logs into the primary site using a terminal or terminal emulator and works remotely. In this scenario, all aspects of ClearCase are readily available for use because the developer is technically working at the primary site. The drawbacks to this approach are that if a terminal interface is being used, only the command-line interface to the SCM tool can be used, with no GUI support. The primary advantage is that little or no additional infrastructure is needed.

The remote developer uses a remote desktop or windowing tool and works remotely. Again, in this scenario, all aspects of ClearCase are readily available for use because the developer is again technically working at the primary site. The drawback of this approach is that the network connection between the remote developer and the primary site must be fast enough to allow for a quick transport of the GUI data inherent in desktop and windowed environments. This approach usually requires some additional infrastructure to support a remote desktop or windowed environment. This can become even more complex if the remote connection must work through corporate firewalls.

Each scenario relies on establishing secure, reliable, and reasonably fast connections between the remote developer and the primary site.

Remote terminal or desktop use is illustrated in Figure 11-1.

Figure 11-1. Remote access.

11.3. Multiple Teams: Producer/Consumer Scenario

In the producer/consumer model, the geographically distributed project teams produce or consume components of the overall system. Only the producer team can modify a component. Projects that consume components do not make any modifications to the components they consume.

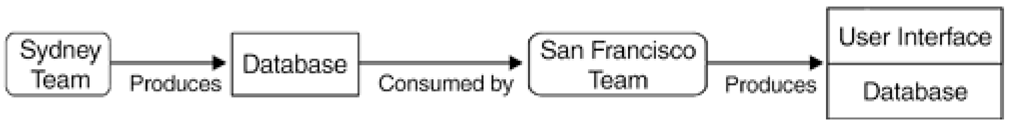
Sharing is accomplished by first defining the architecture of the system. When the architecture is defined, you then assign each component of the system to a single, locally situated project team.

NOTE

If projects at a site develop more then one component, the components chosen for that site should be cohesive, if possible. They should also have very little coupling (particularly build dependencies) with components being developed at other sites.

The simplest producer/consumer model is based on a system composed of two software components with two project teams located at different sites. One project produces a software component; the second project consumes that software component, incorporating it into the final system. For example, let's say that one project team is in Sydney and another is in San Francisco. The Sydney team is producing a software component that provides all the database services and isolates the application from any specific database technology. The San Francisco team builds the final application on top of this database component (see Figure 11-12).

Figure 11-12. Producer/consumer model.



Of course, most software systems are much more complex. A simple but more realistic example might be a multitier architecture in which there are one or two commercially available databases, a database abstraction layer, a middle tier that captures the business logic, and a client layer that provides the user interfaces (see Figure 11-13).

Figure 11-13. A simple architecture.

11.4. Multiple Teams: Shared Source Code Scenario

In the producer/consumer model, the consumer does not modify the components that are produced. Now let's examine the case in which multiple teams are working on components that may be modified by any team at any site. When two or more teams are modifying the same set of source code in parallel, the development and integration processes are more complicated if your objective is to maintain a common code base and not diverge from it. Doing this when the teams are located at the same site is difficult. Geographically distributed teams make it even more complicated.

Because of this complexity, it is best to avoid the shared source code scenario, if at all possible, and use the producer/consumer scenario. In the real world, this is not always possible or practical. Therefore, optimally, some combination of the producer/consumer model and the shared source code model would be employed.

In a number of legitimate situations, you need to support shared source code. A number of legacy problems also can lead to the shared source code model as the only solution. Many of these problems are avoidable if detected early. A number of situations that could cause you to support shared source code with distributed teams are listed here. Some of these are similar, and often you will find more than one of these in play at any given site.

- The system architecture is monolithic or brittle. If the system architecture is monolithic or brittle, it becomes impossible to define components on any functional boundary. Because the system cannot be decomposed, the development teams cannot be assigned pieces of the system that do not have a high degree of coupling.

- The system is in maintenance mode. If the system is fairly old and most work being performed is maintenance work, it is often easier to think in terms of adding features even if those features span architectural boundaries. Sometimes maintenance team sizes are smaller, as compared to new development, so dividing the work by components is not always practical.

- The organization favors a feature-based approach. A feature-based approach is often applied during system maintenance. It assigns features to individuals or teams even if those features cross architectural boundaries. The individual or team is responsible for implementing the feature, regardless of what code is touched. Organizations that favor this approach require developers to have a broad knowledge of the entire system.

- It just happened that way. In many cases, shared code just happens. One team needs some of the code from another team and just takes a copy. In this case, each team modifies all parts of the code, but there are essentially multiple variants of the same code evolving in different groups. Inevitably, this leads to costly project delays during integration or when the receiving teams want to incorporate further changes made by the original team.

- There are remote porting/platform teams. This case is slightly different, and it is a very legitimate and unavoidable scenario for shared source code. In this case, both teams work on the shared source code. One team produces new functionality, and the second makes secondary changes. Typically, the second team is porting the product to a new platform, making changes to support a different language, or both.

  This case comes close to a producer/consumer relationship because it is largely one-way. However, to optimize future porting efforts, it is often a good idea to incorporate changes made by the porting/platform team into the primary team's source code base.

- You deliver source code to your customer. In this case, "you" may be a software house that delivers components to an outside party, which incorporates or modifies the

11.5. Single Team: Distributed Members Scenario

This scenario is a bit different from either of the other scenarios. It deals with a single project team that has distributed team members. Typically, fewer developers are involved and less remote management structure is in place than with either of the multiple-team scenarios. For example, if you have 20 developers at one site and 50 developers at another site, you are likely to have project management at each site. However, if you have 20 developers at one site and 3 developers at a remote site, you might have project management only at one site (that is, the 3 remote developers report into the same management structure as the others).

So, in this scenario, you will have a shared architecture and project management located at one site. The site you pick should contain the majority of your development team members. The remote site should contain only a small group of developers. (It is difficult to give concrete guidelines on how many developers can be at a remote site before you need to establish a remote team and adopt one of the preceding scenarios. If pressed, I would say that when a remote site has more than five developers, you should consider establishing a technical lead to handle coordination and day-to-day interactions. With more than 10 developers, I would move to a multiteam approach.) Another variant of this approach is to have multiple remote sites with only one or two developers at each site. For example, you might be running a project in which each developer is working from home.

Architecturally, things are not much different from the previous two scenarios. You must define a system architecture, divide it into components, and assign components to individuals rather than teams. This approach reduces contention among different individuals for the same source files. However, completely dividing components among individuals is rarely possible in practice. Some code must be shared, so the shared source scenario on an individual level can be applied to these pieces. This approach is best implemented by assigning specific features to individual team members; it works well on projects in a maintenance phase (see "Organizing Large Multiproject Development Efforts" in Chapter 7, "Managing and Organizing Your ClearCase Projects," for a discussion on architectural versus feature orientation).

Whether or not code is shared, you will want to integrate changes much earlier and much more often. In fact, you want to treat each remote developer as if he or she is not remote at all. That is, he or she should deliver changes in the same way as other team members, and those changes should be incorporated into the nightly build.

It is inefficient to establish a producer/consumerstyle relationship at the individual developer level. You want the development team to be cooperating closely and integrating early and often. This means that the technology you choose must facilitate this kind of support. In particular, the ClearCase Remote Client lends itself to this kind of development model.

You can use any of the four approaches described in the section "How ClearCase Supports Distributed Development," earlier in this chapter, to support distributed team members. If you are using local access and ClearCase MultiSite, other interesting aspects are discussed next.

11.5.1. How the UCM Model Supports Local Access

ClearCase UCM provides support for local use. After the project VOB and source VOBs have been replicated to the remote site, the remote developer can join a project exactly as if he or she were working at the primary site. ClearCase UCM handles the details. The remote developer's development stream is mastered at the remote site, and the remote developer can create activities and make changes as necessary.

After changes have been made, the remote user performs a deliver operation. The delivery process is slightly different for remote users. The deliver is performed using the pull model instead of the default push model used by developers at the primary site. In pull deliveries, the remote developer marks a set of activities as ready for delivery. The project integrator at

11.6. Other Uses for ClearCase MultiSite

ClearCase MultiSite can be applied in three other ways. Although not originally designed for these purposes, it has been adopted to solve backup, product delivery, and platform interoperability problems. Because MultiSite is an add-on product to ClearCase, some of these uses may be cost-effective only if you are already using MultiSite to support distributed development.

11.6.1. MultiSite for Backup

The data you store in your SCM system is critical to your company, especially if your company's business is dependent on that software (for example, integrated software vendors, defense, e-commerce, and mission-critical IT applications). You must back up your VOB data.

To back up a VOB, a system administrator locks the VOB, backs up the files associated with it, and then unlocks it. The locking time depends on the size of the VOB but requires some downtime. One of the benefits to using MultiSite as a backup strategy is that you can back up the replica, thus leaving the primary VOB unlocked and available 24 hours a day. The cost associated with this strategy is that recovery takes longer because you need to use MultiSite in the recovery process instead of just simply recovering the original VOB files.

The frequency of VOB backups depends on your strategy but generally occurs nightly. So, another benefit of using MultiSite is that you can set up an hourly synchronization between the master and the backup replica. In a recovery scenario, you then have all but the last hour's worth of changes, without the overhead of doing hourly backups.

11.6.2. MultiSite for Delivery

MultiSite has also been employed as an automated delivery vehicle. Using MultiSite to replicate the data for and synchronize two sites that often exchange files and information means that a user can just check in a new version of a file; after the synch time, the changes automatically are available at the remote site. If the information is important, this mechanism is more secure and reliable then using `ftp` to simply copy the files from one site to the other.

11.6.3. MultiSite for Platform Interoperability

ClearCase supports interoperability between Windows and UNIX development environments. In particular, a Windows client can access UNIX VOBs and use UNIX views. Setting this up properly can require additional file system software, such as an SMB server on UNIX or an NFS client on Windows if you are using dynamic views.

MultiSite can be used to simulate a homogeneous environment by creating a replica on a UNIX platform and another on a Windows platform. In this case, each side can be better isolated from one another, and the additional steps needed to communicate between UNIX and Windows go away because the VOB exists locally on both platforms. This strategy works particularly well if you have one team developing the UNIX version of a product and one developing the Windows version against a shared source.

11.7. Summary

This chapter described the three fundamental models for distributed development and explored how ClearCase supports these models using combinations of native ClearCase client tools, the ClearCase Remote Client, the ClearCase Web client, and ClearCase MultiSite. Furthermore, you were presented with a brief description of ClearQuest MultiSite. We noted that it extends support for distributed development by replicating, controlling, and tracking change-request data. That data, in turn, can be integrated with the control and tracking of development artifacts that ClearCase and ClearCase MultiSite provide to give organizations that are engaged in distributed development a robust and complete solution.

Chapter 12. Change Request Management And Clearquest

This chapter provides a brief introduction to a topic that is closely related to SCM: change request management (CRM). An effective and comprehensive change-management solution can be achieved only through the application of the appropriate tools and processes in both disciplines. This chapter introduces CRM and the software product ClearQuest, and discusses how ClearQuest extends the UCM model to support CRM. (CRM is a very important part of a complete change-management solution, and an entire book couldand shouldbe devoted to the topic. Regrettably, I have only scratched the surface. Refer to [Humphrey, 1989], [Whitgift, 1991], and the Rational Unified Process [RUP 5.5, 1999] for additional information.)

Team LiB PREVIOUS NEXT

12.1. What Is Change Request Management?

Change request management is the recording, tracking, and reporting of requests from any stakeholder to change a software system. It includes the processes an organization uses to decide what changes to make and the resolution processes used to make them. (The acronym CRM is also used in the related but different domain of customer support. In this domain, it stands for customer relationship management. This chapter uses CRM to refer to defect and request management in the software-development domain.)

Change request management is a central part of a complete change-management solution. Without recording, change requests might be lost or might remain unknown. Without tracking, existing change requests might be forgotten or might remain unaddressed. Without reporting, project managers might have a difficult time assessing project status, determining the level of product quality, and conveying project status to upper management.

Rational Unified Process [RUP 5.5, 1999] defines change request management as a process that "addresses the organizational infrastructure required to assess the cost and schedule impact of a requested change to the existing product. Change Request Management addresses the workings of a Change Review Team or Change Control Board." Change request management is like the central nervous system of your software-development process and is integral to good software-development practice. CRM processes and tools manage data that is essential to project stakeholders and the smooth operation of a software project. Other development disciplines related to and supported by change request management are requirements management, testing, release management, customer support, and project management.

Team LiB PREVIOUS NEXT

12.2. What Are Change Requests?

Rational Unified Process defines a *change request* as "a general term for any request from a stakeholder to change an artifact or process. Documented in the change request is information on the origin and impact of the current problem, the proposed solution, and its cost" [RUP 5.5, 1999]. CRM processes are often closely related to a company's internal organization, and terminology in this area is far from standard. However, change requests are often divided into two major categories: enhancement requests and defects. (The types of change requests used to manage change vary widely from company to company and even internally between projects within the same company. The types described here, defect and enhancement request, represent only the most basic kinds of change requests.)

Enhancement requests specify a new feature of the system or a change to the "as designed" behavior of a system. *Defects* are "an anomaly, or flaw, in a delivered work product. Examples include such things as omissions and imperfections found during early life cycle phases and symptoms of faults contained in software sufficiently mature for test or operation. A defect can be any kind of issue you want tracked and resolved" [RUP 5.5, 1999]. Although much of the data maintained for enhancement requests and defects is similar, these two types of change requests are often handled very differently in the CRM process.

12.3. The Change Request Management Process

When implementing CRM, you must make a number of decisions. Typically, defining a CRM process involves a number of stakeholders in a number of different functional organizations (such as project management, development, and testing). The decisions you must make revolve around what types of change requests you will track, what data you will track for these change requests, and how you will track the change requests.

The types of change requests you choose to track could be as simple as defects and enhancements requests, described previously. For large organizations, change requests can become much more complex and multilayered. For example, you might define a type of request that represents an external customer request. One or more external customer requests might spawn one or more engineering-level enhancement requests.

When you have defined the types of requests you are going to track, the next step is to define the type of information you need to record throughout the life of the request. For example, if you are tracking defects, you might want to record things such as who submitted the defect, when it was submitted, what the resolution was, whether it was a duplicate of another defect, or during what phase in the life cycle the defect was found and fixed.

The most important and often most difficult thing to define for each change request is the process used to monitor the change request. This process is typically represented in CRM tools by a state transition model. That is, you define a set of states and a set of actions to transition the change request from one state to the next.

Change request types and state transition models vary widely. However, almost all CRM processes include (or should include) the following six stages:

1. **Submission** Requests to change a software system are recorded (submitted).
2. **Evaluation** Change requests are evaluated, categorized, and prioritized.
3. **Decision** Based on the evaluation, a decision is made regarding which change requests to implement and in what order.
4. **Implementation** Changes are made to system artifacts, and new artifacts are produced, with the goal being to implement the requested change. The software system documentation is updated to reflect the change.
5. **Verification** The change request implementation is verified as either meeting the requirements or fixing a defect.
6. **Completion** The change request is closed, and the requestor is notified.

The following sections look at each stage of change request management, and compare and contrast the treatment of enhancement requests and defects in these stages.

12.3.1. Submission

During submission, requests to change a software system are recorded. Defects and enhancement requests usually differ in the origin of the request and the type of information collected. Enhancement requests come from a wide variety of sources. In many cases, they come from customers and arrive in engineering directly or indirectly through marketing or customer support. The key data captured for enhancement requests are the importance of the request to the customer, as much detail about the request as possible, and the identity of the original requestor (if submitted indirectly) so that engineering can ask for clarification. In some cases, enhancement requests come internally from either testing or in-house use. In these cases, make sure that product management is aware of these requests.

12.4. What Is ClearQuest?

ClearQuest is an IBM Rational product that provides out-of-the-box support for the change-request management process. ClearQuest is a complementary product to ClearCase. It can be integrated with other SCM tools, for projects that require more advanced change request management before they require more advanced software-configuration management.

ClearQuest has three key parts: a user interface (the client); a back-end core, which provides an interface to the data store (ClearQuest 2003.06.14 supports Oracle [on Windows or UNIX], Microsoft SQL Server, SQL Anywhere, DB2 [on Windows or Unix] and Microsoft Access databases); and a designer used to create and customize the CRM processes. The user interface has native variants that run on Windows, UNIX, a ClearQuest adapter plug-in that runs in an Eclipse IDE, and a Web client. The Windows and UNIX clients have three key panels (see Figure 12-1). On the left side is the tree view of saved queries, charts, and reports. On the right side is the results panel divided into a master/slave display. One line is shown per request in the top panel (master), with the lower panel (slave) showing the details of the request selected in the top panel.

Figure 12-1. ClearQuest Windows display.

[View full size image]

The ClearQuest Eclipse Client provides the same information, but in a slightly different format, providing the tree view of saved queries, charts, and reports; the results panels appear in a tabbed interface. Furthermore, the ClearQuest Eclipse Client adapter allows multiple user logins in the same Eclipse session. You can now customize your Eclipse perspectives to include whatever ClearQuest data you are interested in by adding ClearQuest views. Figure 12-2 shows some of the features of the Eclipse ClearQuest adapter. Figure 12-3 shows an Eclipse Java perspective with ClearQuest views included.

Figure 12-2. ClearQuest Eclipse Client.

[View full size image]

Figure 12-3. Eclipse Java Perspective.

[View full size image]

ClearQuest provides out-of-the-box record types for defects and enhancement requests. If your change-request management processes are different from those provided, you can use the out-of-the-box types as a starting point or design your own record types from scratch. Instead of doing this definition in some proprietary language, you have available in ClearQuest a design tool called the ClearQuest Designer. It enables you to modify all aspects of the schema, creating new record types and new fields, defining a state transition model, and controlling the layout of those fields in the submission and display forms. The ClearQuest Designer also supports basic version control for the CRM schema.

Figure 12-4 shows the ClearQuest Designer. The left panel provides a way to navigate to the various pieces of the CRM process. This example shows a folder that contains all the record types in the out-of-the-box schema. You can see three types: Activity, Defect, and Enhancement Request. The Defect record type is open, and under the folder States and Actions, the state transition matrix is selected. The right panel shows the states and actions for the defect record type. This is where you define the state model for your record and the legal transitions, called actions, that get you from one state to the next.

12.5. How Do I Use ClearQuest Data?

ClearQuest records a significant amount of essential project datafor example, what requests for change have been made, which ones are critical, which ones are being worked on, who is working on them, and which defects have been fixed. This is all very good and very valuable information. However, the data in a change-request management tool is useful only if there is an easy way to extract it in a form that enables you to use it. Project managers usually need this information to assess project statusdetermining the level of product qualityand to convey project status to upper management.

ClearQuest provides mechanismsqueries, reports, and chartsto deliver the data in a variety of forms. Queries provide a flexible means of browsing a subset of all the requests in the database (for example, "show me all the defects assigned to me"). Reports provide a means to collect data and format it in a way it can be printed, included in an Excel spreadsheet, or posted to a Web site. Charts provide online and printable graphs that offer insight into data trends, data distribution, and request aging. The following sections cover each of these areas.

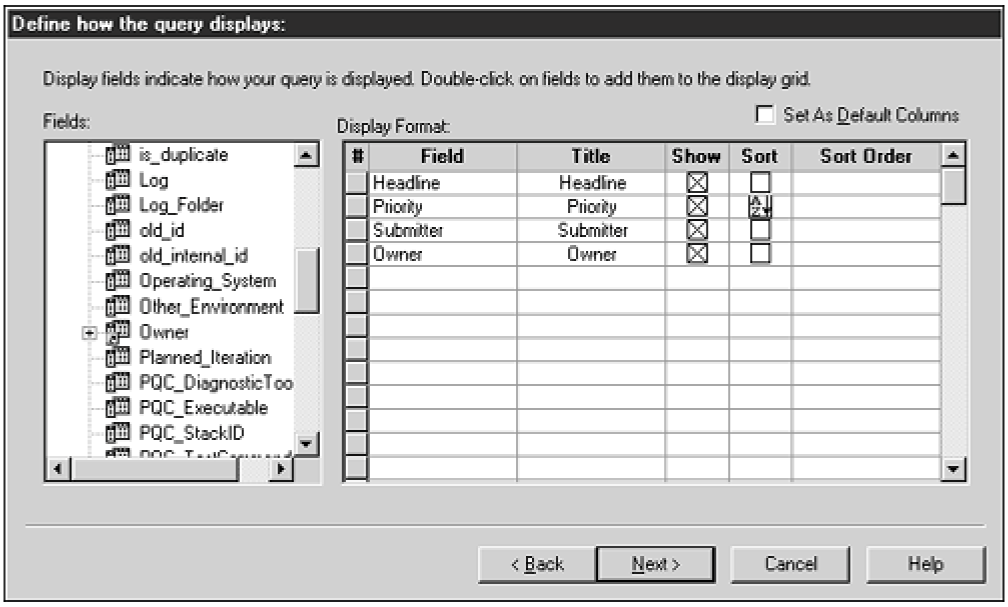
12.5.1. Queries

The purpose of a ClearQuest query is to return a subset of all the records stored so that you can evaluate those records. For example, a query could be "show me the defects assigned to me," "show me all open defects on Project X," or "show me all resolved defects in Release Y." A ClearQuest query is a set of search criteria defined by a filter that returns a list of records in a master-slave style display. That is, the top window (the master) shows the list of requests one per line, and the bottom window (the slave) shows the details of the request selected in the master window (refer to Figure 12-1).

Queries and how their results are displayed are constructed graphically by using the ClearQuest Query Creation Wizard. This tool walks you through the process of defining the query. The first step is to decide which data fields to display in the results window and how the data should be sorted. In Figure 12-6, you can see that the `headline`, `priority`, `submitter`, and `owner` fields have been chosen from all the fields available and will be shown in the master display. You can also see that we have decided to sort this list by priority.

Figure 12-6. ClearQuest Query Wizard: results definition.

[View full size image]



Define how the query displays:

Display fields indicate how your query is displayed. Double-click on fields to add them to the display grid.

Fields:

- is_duplicate
- Log
- Log_Folder
- old_id
- old_internal_id
- Operating_System
- Other_Environment
- Owner
- Planned_Iteration
- PQC_DiagnosticToo
- PQC_Executable
- PQC_StackID

Display Format: ☐ Set As Default Columns

#	Field	Title	Show	Sort	Sort Order
	Headline	Headline	☒	☐	
	Priority	Priority	☒	↕	
	Submitter	Submitter	☒	☐	
	Owner	Owner	☒	☐	

12.5.3. Charts

ClearQuest charts graphically represent data, both online and in printable form, making it easier to analyze. ClearQuest provides three different types of charts: distribution, trend, and aging. As with ClearQuest queries, a Chart Creation Wizard walks you through the process of creating your own charts. You can then save and rerun charts as needed.

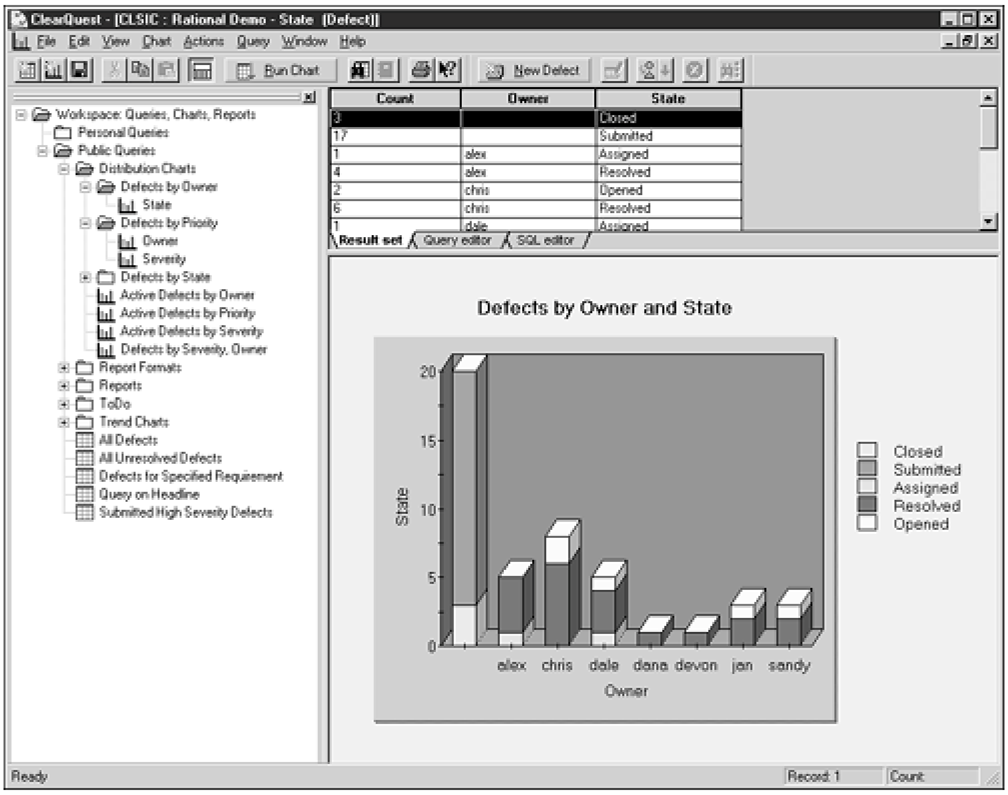
Distribution Charts

Distribution charts are used to categorize data and see how a given data sample is distributed across different categories. For example, charts can be used to balance the workload across team members and to get a look at the number of defects by category, such as priority or severity.

Figure 12-9 shows an example distribution chart. In the left panel, you see the public queries and the Distribution Charts folder. This example screen shows the Defects by Owner and State chart. In the top-right panel is the data set being used. The lower-right panel shows the chart itself. Each bar represents a developer along the x-axis. The y-axis shows the number of defects. Bars are color-coded by state. The first bar shows defects that have not been assigned or that have been closed. You can quickly see two things in this chart. First, there is a lot of unassigned work, which indicates unknown risk for the project. Second, Alex, Dana, and Devon do not have any open requests they are working on. Get to work, and assign the new requests to those three challenged individuals!

Figure 12-9. Example distribution chart.

[View full size image]



Count	Owner	State
3		Closed
17		Submitted
1	alex	Assigned
4	alex	Resolved
2	chris	Opened
6	chris	Resolved
1	dale	Assigned

Trend Charts

Trend charts are used to display how change requests are moving through the system over time. As defined by the ClearQuest documentation, "trend charts show how many records

12.6. How Does ClearQuest Support UCM?

ClearQuest can be used to extend the UCM model (described in Chapter 3, "An Overview of the Unified Change Management Model") with support for change request management. The UCM process diagram (refer to Figure 3-5) shows two steps that the project manager performs: assigning and scheduling work, and monitoring project status. ClearQuest is the technology that supports these steps. With ClearCase alone, activities have only a name, a one-line description, and a change set. ClearCase maintains the essential configuration-management information, but not the change-request management information. To support change request management, additional data must be maintainedfor example, state, assigned user, a long description, priority, severity, and so on. In a UCM environment, ClearQuest provides these capabilities. If you are interested in an integrated change-management solution, you need both ClearCase and ClearQuest.

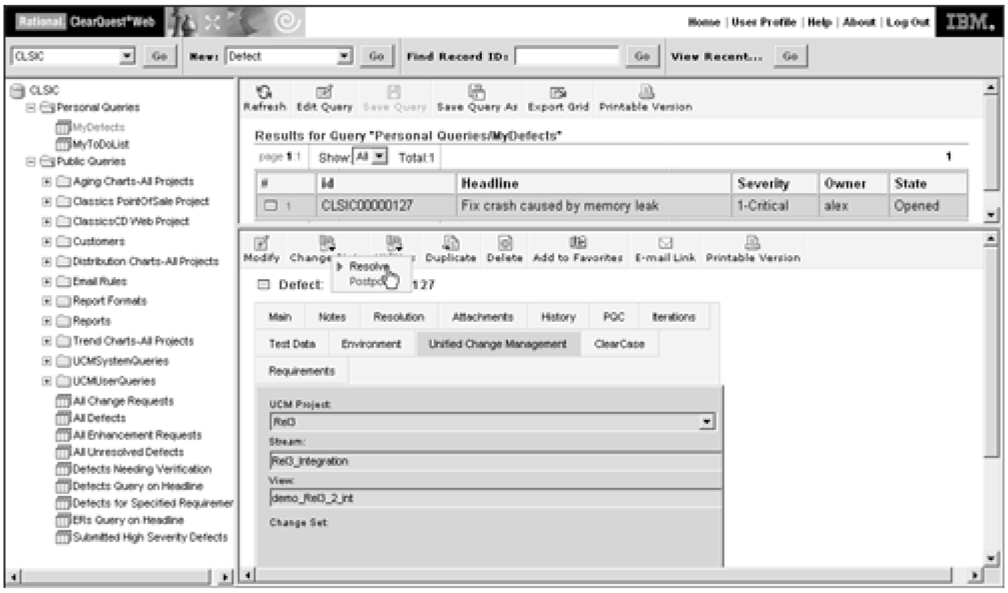
With ClearCase UCM alone, activities are used as the mechanism to track a change set (see "Organize and Integrate Consistent Sets of Versions Using Activities" in Chapter 1, "What Is Software Configuration Management?"). The primary difference between using ClearCase alone and using it in combination with ClearQuest is the level of process centered on the activities.

If you are using ClearCase UCM by itself, activities are created at the point of use. That is, individual developers, working in their development stream, create activities when they check out files. These activities consist of a one-line description and so appear to the developer as kind of a metacomment for a change spanning multiple files. "Modifying Files and Directories" in Chapter 8, "Development Using the ClearCase UCM Model," goes into this in more detail. The bottom line is that ClearCase standalone with UCM does not support the process for activities you need if you want to do change request management.

With ClearQuest, activities are submitted, evaluated, decided on, implemented, validated, and completed. They can be of any record type that you define. So, they can be defects, enhancement requests, features, incidents, tasks, and so on. This is accomplished by adding UCM required fields from the Clear Quest Designer to a given record type. Many of the predefined record types come pre-UCM-enabled. Figure 12-12 shows the additional tab, called Unified Change Management, and fields added when you apply the UCM package to a record type in the ClearQuest Designer.

Figure 12-12. UCM fields in a ClearQuest record (Web interface shown).

[View full size image]


Results for Query "Personal Queries/MyDefects"

#	Id	Headline	Severity	Owner	State
1	CLSIC00000127	Fix crash caused by memory leak	1-Critical	alex	Opened

UCM Project: ReO3
Stream: ReO3_integration
View: demo_ReO3_2_int
Change Set:

12.7. ClearQuest MultiSite

ClearQuest MultiSite is an add-on product for ClearQuest that provides the mechanisms to replicate track and control changes to ClearQuest Schema repositories and their associated databases. ClearQuest MultiSite is discussed in more detail in Chapter 11, "Geographically Distributed Development." Figure 12-13 shows USA's ClearQuest Web client looking at a record mastered in India.

Figure 12-13. A remotely mastered record in ClearQuest.

[View full size image]

12.8. Summary

This chapter discussed what change request management is, along with change requests and the processes that are involved in tracking and managing them. You have seen that a process that involves change requests moving from an initial Submitted state through various intermediate states until they reach some terminal state (such as Closed or Rejected) provides the capability to track the progress of a project and to use that tracking information to make informed management decisions regarding the project. ClearQuest provides out-of-the-box support for change request management with native, Eclipse, and Web clients. ClearQuest also provides highly flexible and easy-to-use means to customize not only the information to be tracked, but also the actual process, states, and transitions required for effective project management in a wide range of project types. Finally, you learned that ClearQuest and ClearCase are integrated, allowing UCM activities to be tracked and controlled through the ClearQuest change request management model. These capabilities allow projects to both control their development and see in real time the current status of the project. This status is based on nonintrusive data collection that occurs as an integral part of people working on the project.

Appendix A. Redoing and Undoing Change Sets with UCM

You might run into some conditions in which UCM does not supply an "out-of-the-box" solution. One condition arises when you want to "undo" or back out an activity from a stream. In most cases, this occurs as a deliver activity on the integration stream. You could always leave the activity on the stream, fix it on your development stream, and redeliver. You could also remove the activity's change set versions if it was the last activity on the stream. Or, you can utilize the `cset.pl` script to undo the activity from the stream. Another condition that might arise during UCM development is the desire to reapply an activity's change set from another stream to your current stream (a Redo operation). You can use the deliver operation to accomplish this if the desired activity has no predecessor activities on its stream that you do not want. In other cases, UCM does not allow a deliver operation to proceed, as in during interproject deliveries between projects that do not have common foundation baselines (projects that sprouted from different imported baselines off `/main`, for instance). In this case, using the `cset.pl` script to redo the activity from one stream to another can work around this issue.

A.1. Location of Script

Currently, the `cset.pl` script is maintained by David E. Bellagio. Many customers have used this at various stages when using UCM. As of the date of this publication, the script has not yet been pushed to IBM developerWorks (see www-130.ibm.com/developerworks/), which is where it will reside eventually. Just search developerWorks for "cset.pl." If you can't find it, send an email to dbellagio@us.ibm.com, and I'll send you a copy.

A.2. Limit Script Use to Integrator Role

The `cset.pl` script should be used only as needed by the integrator, not as a main development tool for the developer. If you let developers use this whenever they want, issues could arise that would have been avoided with normal UCM operations. For one, the Redo operation does not keep track of the dependencies in a way that `cleartool` diffbl will know about. So, reporting changes that were redone in other streams would have to be accomplished with other methods.

A.3. Script Interface

Here is the interface to the script (cset.pl version 6.0).

Usage:

[View full width]

```
cset.pl [-help] [-ignore] [-graphical]
        {-redo | -print | -undo | -findmerge}
➥
 [activity]
```

Arguments:

help	Displays this message and exits.
ignore	Ignores check for checked-out files.
graphical	If specified, always uses the graphical diffmerge tool.
redo	Redoes the change set specified by activity.
print	Prints the change set specified by activity.
undo	Undoes the change set specified by activity.
findmerge	Merges the change set from activity (this is not redo).
activity	The specified activity-object-selector to redo, print, or undo. If not specified, you are prompted for an activity.

More details on each of these options are available from the script's help interface. However, to understand the benefits of this script, you need to understand how the deliver operation works in relation to ClearCase's underlying merge utility.

A.4. Why Is This Useful?

`cset.pl` can be used to manipulate UCM activities in a different manner than what you can do using the "out-of-the-box" interfaces of UCM (such as deliver and rebase). It achieves these different behaviors by processing the UCM activity's change set with either the ClearCase additive merge or the subtractive merge. The additive merge is used when redoing an activity from another stream into your current stream (the stream from which your view is attached). It is important to know that `cset.pl` does not implement any new technology that does not exist in base ClearCase today. Thus, the redo and undo operations of `cset.pl` will suffer from the same side effects of the additive and subtractive merge operations within ClearCase.

`cset.pl` will redo an activity's change set from another stream into an activity on the current stream. A redo deals only with the activity's change set, not other changes from other activities that were predecessors of the activity to be redone. `cset.pl` can also undo an activity from the current stream into the current activity on the current stream. This essentially backs out the changes of the activity to be undone. This is most commonly used to back out deliver activities from an integration stream.

The following sections describe when and where to use this utility. As stated before, this utility should not be used in an ad hoc fashion, but should be used to facilitate the scenarios it was intended for.

A.5. Redoing an Activity on Another Stream

When using the redo operation of `cset.pl`, developers can merge changes of activities from one stream to any other stream in any order (although, for most changes, order is important).

To use the redo operation, you must be positioned in a view, attached to a stream that will accept the redo. UCM requires that all changes be done in an activity, so you need to also create an activity to accept the redo. Because there is no hyperlink support for this tool yet, users must rely on activity naming conventions to reflect the operation that was performed. So, if you have an activity named Change1 on stream1 and you are currently using a view attached to stream2, you would create an activity Redo_Change1 on stream2 (`cleartool mkact nc Redo_Change1`), and then issue the following command:

```
ccperl c:\bin\cset.pl -redo Change1
```

This operation would move the deltas of the change of Change1 into the current activity (Redo_Change1).

Why Is This Useful?

In some cases, people on other streams might want only certain changes propagated to their stream out of a large set of changes. A deliver of the activity would attempt to merge starting at the cset version (including activity versions up to the activity's versions being delivered). `cset.pl` would simply redo that activity's cset.

Lets look at the difference between a deliver (or `findmerge -cset`) with a `cset.pl -redo:`

In Figure A-1, if you were able to deliver activities (Actp1, Actp2, Actp3) from the integration stream of Project_2 to the integration stream of Project_1, the Deliver1 activity would be comprised of the changes incorporated in activities Actp1, Actp2, and Actp3. There are a few problems with this normal sequence to be aware of:

- Currently, UCM allows deliveries of baselines only between project integration streams, so the previous example is not possible with deliver (but can be accomplished with `cset.pl`). So, pretend that Project_2 is a developer stream, and things will work with deliver. Or, make sure you have a baseline in place before attempting to deliver between projects.

- Suppose that you want to deliver only Actp3's changes? UCM requires that Actp1 and Actp2 be delivered as well, even if they don't relate to the changes in Actp3. A developer might end up having to resolve conflicts with activities he or she did not participate in.

  With only three activities here, it might not be a big deal. Suppose that you have a stream with 150 activities on it, though, and you want only 15 of those on your project stream....

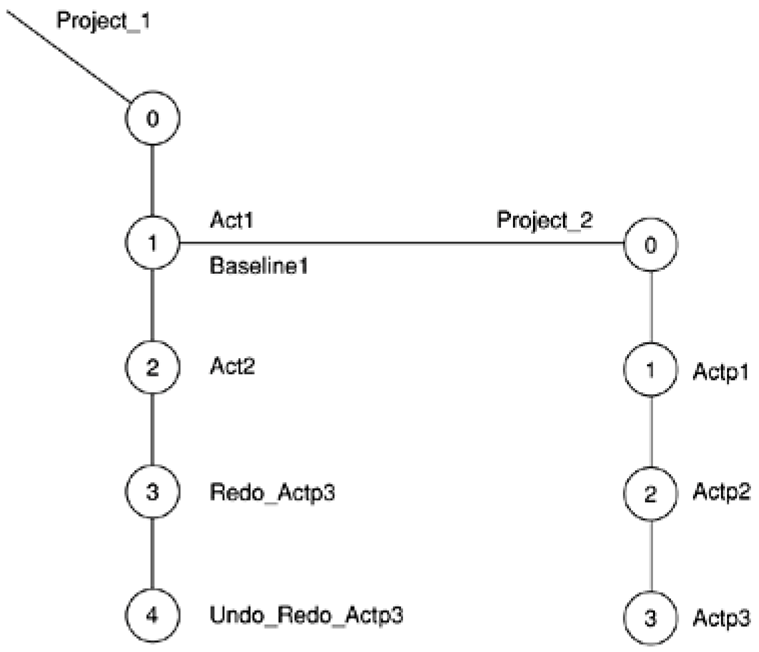
Figure A-1. A sample delivery scenario.

A.6. Undoing a Delivery or Activity

Suppose that later you want to undo the activity you just redid or undo a deliver activityor any activity, for that matter.

As with the redo option, you need to create an activity first, this time called Undo_Redo_Actp3 in your view. Then simply perform a `ccperl cset.pl undo Redo_Actp3`. Again, make sure you use a good name for your activity because that will be an indicator in future reports of what happened. Figure A-3 shows how an undo operation will look in the ClearCase version tree.

Figure A-3. Undoing the redo activity.



Glossary

Most of the definitions in this glossary are taken from one of three sources: ClearCase documentation [ClearCase, 2003], Rational Software's Rational Unified Process [RUP 2003.06.13, 2004], or the IEEE's "Standard Glossary of Software Engineering Terminology" [IEEE Glossary 1990]. In some cases, these sources define a term differently. If the difference in definition aids understanding, both definitions are listed and the source is noted.

activity

1. A unit of work an individual might be asked to perform. Activities can be of different types. For example, a defect, enhancement request, and issue are all activities. This unit of work ties directly into the change-request management system and processes. An activity can also be a child of another activity that appears in the project-management system.

2. A ClearCase UCM object that tracks the work required to complete a development task. An activity includes a text headline, which describes the task, and a change set, which identifies all versions that you create or modify while working on the activity. When you work on a version, you must associate that version with an activity. If your project is configured to use the UCM-ClearQuest integration, a corresponding Clear Quest record stores additional activity information, such as the state and owner of the activity [ClearCase, 1999].

activity-based configuration management

The management of change to a software system based on higher-level activities (such as task, defect, enhancement) rather than individual file versions. This requires an SCM tool to track which file versions go to implementing a specific activity and then to present activities as key objects. The idea is to simplify complexity and ensure that when the system says a defect is included in a specific build, it has, in fact, been included.

administration VOB

A VOB containing global type objects that are copied to client VOBs on an as-needed basis when users want to create instances of the type objects in the client VOBs. Administration VOBs are not a special type of VOB, but are defined as relationships that a VOB can have with other VOBs.

ALBD server

Atria Location Broker Daemon. This ClearCase master server runs on each Clear Case host. It starts up and dispatches messages to the various ClearCase server programs as necessary.

architecture

1. The set of significant decisions about the organization of a software system: the selection of the structural elements and their interfaces by which the system is composed, together with their behavior, as specified in the collaboration among those elements, the composition of the structural and behavioral elements into progressively larger sub systems, and the architectural style that guides this organization, these elements, their interfaces, their collaborations, and their composition. Software

Bibliography

[Appleton, 1998] Appleton, B., S. Berczuk, R. Cabrera, and R. Orenstein . "Streamed Lines: Branching Patterns for Parallel Software Development." Proceedings of the 1998 Conference on Pattern Languages of Program Design, Allerton Park, Ill., August 1998 (Technical Report #WUCS-98-25, http://acme.bradapp.net/branching).

[Beck, 2000] Beck, K. *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

[Berczuk, 2002] Berczuk, S., and B. Appleton . *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Reading, Mass.: Addison-Wesley, 2002.

[Booch, 1999] Booch, G., J. Rumbaugh, and I. Jacobson . *The Unified Modeling Language User Guide*. Reading, Mass.: Addison-Wesley, 1999.

[Brown, 2004] Brown, J., U. Wahli, M. Teinonen, and L. Trulsson . "Software Configuration Management: A ClearCase for IBM Rational ClearCase and ClearQuest UCM." IBM Redbook, December 2004. http://www.redbooks.ibm.com/abstracts/SG246399.html?Open.

[ClearCase, 2003] Rational Software. *Rational ClearCase Rational ClearCase LT IntroductionVersion 2003.06.00 and Later*. Lexington, Mass.: IBM Corp., 2003.

[Feiler, 1991] Feiler, P. *Configuration Management Models in Commercial Environments*. Pittsburgh: Software Engineering Institute, Carnegie-Mellon University, 1991.

[Humphrey, 1989] Humphrey, H. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989. http://www.sei.cmu.edu/.

[IEEE Glossary, 1990] IEEE Standard 610.12-1990. "Standard Glossary of Software Engineering Terminology." New York: Institute of Electrical and Electronics Engineers, 1990.

[IEEE 828-1998] IEEE Standard 828-1998. "IEEE Standard for Software Configuration Management Plans." New York: Institute of Electrical and Electronics Engineers, 1998.

[IEEE 1042-1987] IEEE Standard 1042-1987. "IEEE Guide to Software Configuration Management." New York: Institute of Electrical and Electronics Engineers, 1988.

[Kruchten, 2000] Kruchten, P. *The Rational Unified Process: An Introduction*, *Second Edition*. Boston: Addison-Wesley, 2000.

[Kruchten, 1995] Kruchten, P. "The 4+1 View of Architecture." *IEEE Software* 12, no. 6 (November 1995): 4550.

[Lakos, 1996] Lakos, J. *Large-Scale C++ Software Design*. Reading, Mass.: Addison-Wesley, 1996.

[Leblang, 1994] Leblang, D. "The CM Challenge: Configuration Management That Works." In *Configuration Management*, edited by W. Tichy. West Sussex, England: John Wiley and Sons, 1994.

[Milligan, 2003] Milligan, T. "Principles and Techniques for Analyzing and Improving IBM Rational ClearCase Performance: Part I." In *The Rational Edge*, July 2003. http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/archives/jul03.html.

[Milligan, 2003] Milligan, T. "Principles and Techniques for Analyzing and Improving IBM Rational ClearCase Performance: Part II." In *The Rational Edge*, September 2003.

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

PREVIOUS NEXT

Index

Index

Team LiB | PREVIOUS | NEXT

Index

Team LiB | PREVIOUS | NEXT

Index