Rational. ClearCase and Rational ClearCase LT

Version 7.0.0
Windows, UNIX, and Linux

Guide to Managing Software Projects

GI11-6712-00

Rational® ClearCase and Rational ClearCase LT

Version 7.0.0
Windows, UNIX, and Linux

Guide to Managing Software Projects

GI11-6712-00

Before using this information, be sure to read the general information under Appendix D, "Notices," on page 295.

7th edition (May 2006)

This edition applies to version 7.0.0.0 of IBM Rational ClearCase (product number 5724G29) and IBM Rational ClearCase LT (product number 5724G31) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces **G126-5330-00**.

Contents

Figures

Tables

About this book

This manual shows project managers how to set up and manage a configuration management environment for their development team. It describes how to use the Unified Change Management (UCM) process and the customizable features of base ClearCase.

IBM Rational ClearCase is a configuration management system designed to help software development teams track the objects in software builds. You can adopt the UCM process, or you can use base ClearCase to create a customized configuration management environment.

Who should read this book

A reader needs to understand the base concepts of Rational ClearCase and be able to use either the command line or graphic user interface of Rational ClearCase.

Typographical conventions

This manual uses the following typographical conventions:

- *ccase–home–dir* represents the directory into which Rational ClearCase, Rational ClearCase LT, or Rational ClearCase MultiSite has been installed. By default, this directory is /opt/rational/clearcase on the UNIX system and C:\Program Files\Rational\ClearCase on Windows.
- *cquest-home-dir* represents the directory into which Rational ClearQuest has been installed. By default, this directory is /opt/rational/clearquest on the UNIX system and C:\Program Files\Rational\ClearQuest on Windows.
- **Bold** is used for names the user can enter; for example, command names and branch names.
- A sans-serif font is used for file names, directory names, and file extensions.
- **A serif bold font** is used for GUI elements; for example, menu names and names of check boxes.
- *Italic* is used for variables, document titles, glossary terms, and emphasis.
- `A monospaced font` is used for examples. Where user input needs to be distinguished from program output, **bold** is used for user input.
- Nonprinting characters appear as follows: <EOF>, <NL>.
- Key names and key combinations are capitalized and appear as follows: Shift, Ctrl+G.
- [ ] Brackets enclose optional items in format and syntax descriptions.
- { } Braces enclose a list from which you must choose an item in format and syntax descriptions.
- | A vertical bar separates items in a list of choices.
- ... In a syntax description, an ellipsis indicates you can repeat the preceding item or line one or more times. Otherwise, it can indicate omitted information.

  Note: In certain contexts, you can use "**...**" within a pathname as a wildcard, similar to "*" or "?". For more information, see the **wildcards_ccase** reference page.

- If a command or option name has a short form, a “slash” (/) character indicates the shortest legal abbreviation. For example:

  lsc/heckout

Online documentation

This section describes how you can access the online documentation for Rational ClearCase products.

Help system

To access the Help, use the **Help** menu, the **Help** button, or the F1 key. To display the contents of the online documentation set, perform one of the following actions:

- On Linux or the UNIX system, type `cleartool man contents` .
- On Windows, click **Start > Programs > IBM Rational > IBM Rational ClearCase > Help**.
- On Windows, Linux, or the UNIX system, to display contents for Rational ClearCase MultiSite, type `multitool man contents`.
- Use the **Help** button in a window to display information about that window, or press F1.

Reference pages

To access reference pages from the *IBM Rational ClearCase Command Reference*, use the **cleartool man** and **multitool man** commands. For more information, see the man reference page in the *IBM Rational ClearCase Command Reference*.

Command syntax

To access online documentation by using the command line, use the **–help** command option or the **cleartool help** command.

Tutorial

The tutorial for a Rational ClearCase product provides a step-by-step tour of the important features of the product. To start the tutorial, perform one of the following actions:

- On Linux or the UNIX system, type `cleartool man tutorial` .
- On Windows, click **Start > Programs > IBM Rational > IBM Rational ClearCase > ClearCase Tutorial**.

PDF manuals

To access PDF manuals for Rational ClearCase products, use the command line to navigate to the following directories:

- On Linux or the UNIX system, *ccase–home–dir*/doc/books
- On Windows, *ccase–home–dir*\doc\books

Product-specific features

This manual describes Rational ClearCase and Rational ClearCase LT. Rational ClearCase LT does not include all features available in Rational ClearCase. In addition, some user interfaces differ in the two environments. This manual uses the following label to call out differences: **Product Note**. When the term Rational ClearCase is used outside of a **Product Note** section, it refers to both products.

Manual organization

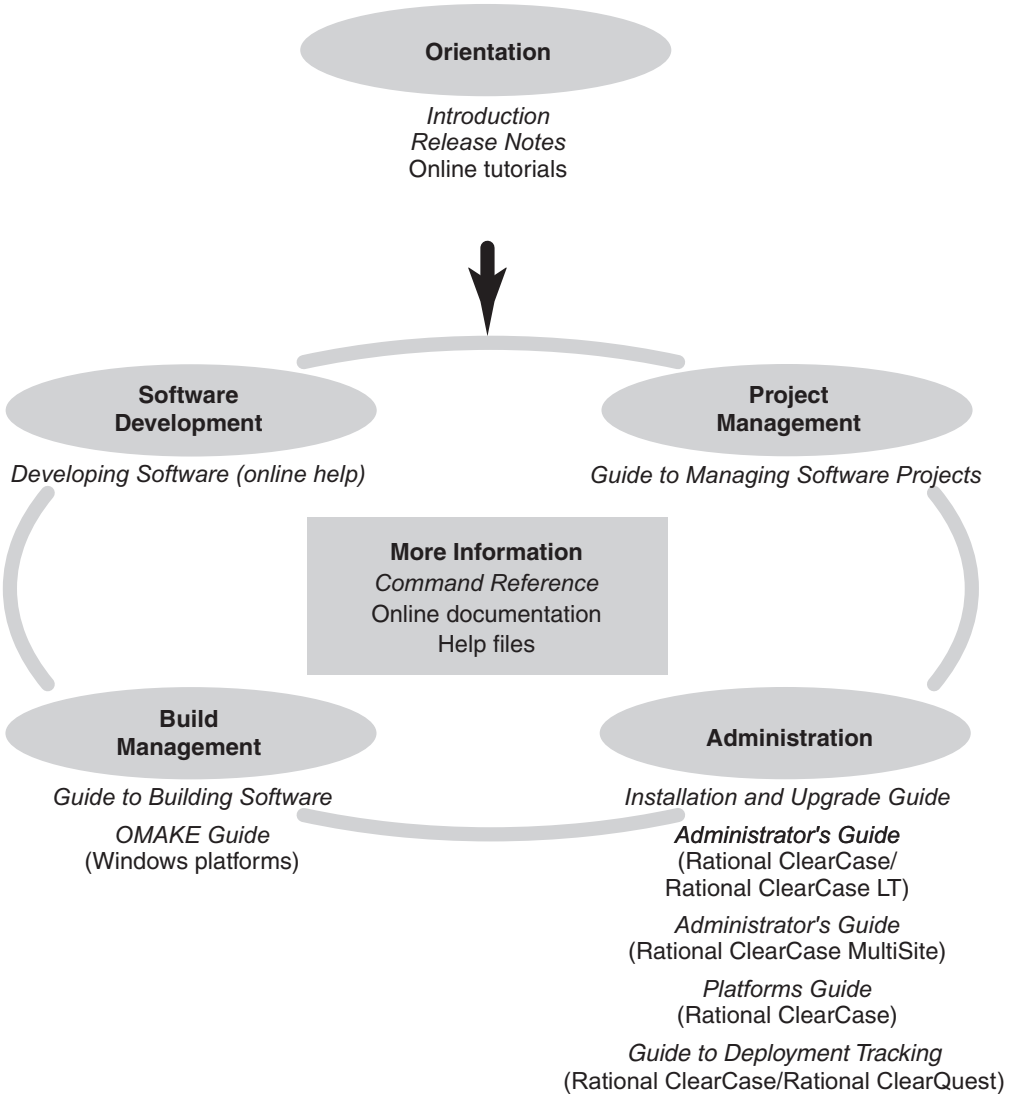
The manual is divided into the following parts:

- Part 1, "Introduction." An introductory part highlights the features of UCM and base ClearCase.
- Part 2, "Working in UCM." Read this part if you plan to use UCM to implement your team's development process.
- Part 3, "Working in base ClearCase." Read this part if you plan to use the base ClearCase features to implement a customized development process for your team.

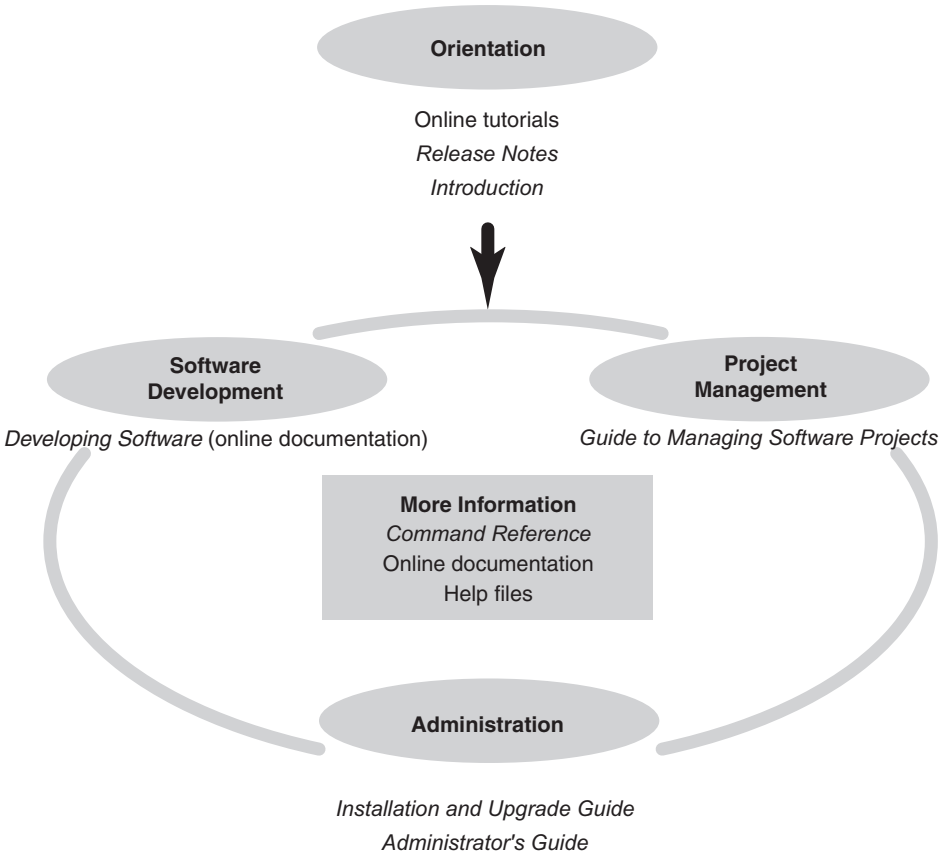
Several appendices carry information of special interest and legal notices.

Related information

Rational ClearCase documentation roadmap



Rational ClearCase LT documentation roadmap



Contacting IBM Customer Support for Rational software products

If you have questions about installing, using, or maintaining this product, contact IBM Customer Support as follows:

The IBM software support Internet site provides you with self-help resources and electronic problem submission. The IBM Software Support Home page for Rational products can be found at http://www.ibm.com/software/rational/support/.

Voice Support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to http://www.ibm.com/planetwide/.

Note: When you contact IBM Customer Support, please be prepared to supply the following information:

- Your name, company name, ICN number, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your PMR number (if you are following up on a previously reported problem)

Downloading the IBM Support Assistant

The IBM Support Assistant (ISA) is a locally installed serviceability workbench that makes it both easier and simpler to resolve software product problems. ISA is a free, stand-alone application that you download from IBM and install on any number of machines. It runs on AIX, (RedHat Enterprise Linux AS), HP-UX, Solaris, and Windows platforms.

ISA includes these features:

- Federated search
- Data collection
- Problem submission
- Education roadmaps

For more information about ISA, including instructions for downloading and installing ISA and product plug-ins, go to the ISA Software Support page.

IBM Support Assistant: http://www.ibm.com/software/support/isa/

Summary of changes

This edition adds material describing composite baselines, the usage of multiple UCM projects, and the set up of the base ClearCase integration with Rational ClearQuest.

- In Chapter 2, new sections under "Starting from a baseline" on page 13, describe composite baselines, baselines and their uses, and baselines and streams. Under "Making a baseline" on page 19, new sections describe the rebase operation, directions of rebase operations (advance, revert, and lateral), and rules for rebase operations.
- In Chapter 3, the following information is new:
  - Under "Identifying read-only components" on page 33, new text describes modifiability of components without a VOB root directory.
  - Under "Choosing a stream strategy" on page 34, new sections describe stream configurations, baseline contents, and stream relationships.
  - Under "Pure composite baselines" on page 47, new sections describe dependency relationships in composite baselines, pure composite baselines and whether to use them, and creation of composite baseline descendants.
  - Under "Multiple PVOBs and feature levels" on page 56, a new section describes feature levels in environments with multiple PVOBs.
  - Under "Using mergetype to manage merge behavior" on page 56, a new mergetype, copy, is described.
  - In "Planning how to use the UCM integration with Rational ClearQuest" on page 57, under "Use of multiple user databases" on page 59, the need for unique names is described.
- In Chapter 4, under "Policies for the UCM integration with Rational ClearQuest" on page 69, two new policies are described: "Disallow submitting records from ClearCase client" on page 69 and "Allowed record types" on page 70.
- In Chapter 5, under "Creating users and adding credentials" on page 82, a new section describes creating and maintaining credentials for Rational ClearQuest database sets used in the UCM integration.
- In Chapter 7, the following information is new:
  - Under "Adding components" on page 109, a new section, "Element relocation" on page 111, describes the use of the mkelem_cpver.pl script.
  - Under "Resolving baseline conflicts" on page 120, new information is added to the section "Conflicts between composite baselines" on page 120.
- In Chapter 8, under "Supported triggers" on page 129, **lock** and **unlock** are added. Also, under "Using the same script" on page 132, the invocation of **ratlperl** is described.
- Chapter 9, in "Project uses" on page 141, has new information that describes release-oriented and component-oriented projects and composite baselines in each type, and describes bootstrap projects.
- In Chapter 12, under "Using the same script" on page 189, the invocation of **ratlperl** is described.
- Chapter 13 consolidates information from multiple sources to describe the base ClearCase integration with Rational ClearQuest.
  - "Planning for the base ClearCase integration with Rational ClearQuest" on page 196

  - "Setting up the Rational ClearQuest user database for base ClearCase" on page 196
  - "Editing the configuration file" on page 200
  - "Connecting Rational ClearCase clients and a Rational ClearQuest user database" on page 203
  - "Making policy choices" on page 209
  - "Enhancing performance" on page 211
  - "Debugging and analyzing operations" on page 215
- In Chapter 14, under "Common merge scenarios" on page 222, a new procedure is described in "Merging a new release of an entire source tree" on page 225 for using **clearfsimport** to accomplish the merge.

Part 1. Introduction

Chapter 1. Choosing between UCM and base ClearCase

Before you can start to use IBM® Rational® ClearCase® to manage the version control and configuration needs of your development project, you need to decide whether to use the out-of-the-box Unified Change Management (UCM) process or base ClearCase. This chapter describes the main differences between the two methods from the project management perspective.

The next two parts of this manual present conceptual and usage material for each method from the perspective of the project manager and project integrator. Part 2, "Working in UCM" describes how to manage a project using UCM. Part 3, "Working in base ClearCase," on page 155 describes how to manage a project using the various tools in base ClearCase.

Differences between UCM and base ClearCase

Base ClearCase consists of a set of powerful tools to establish an environment in which developers can work in parallel on a shared set of files, and project managers can define policies that govern how developers work together.

UCM is one recommended method of using Rational ClearCase for version control and configuration management. UCM is layered on base ClearCase. Therefore, it is possible to work efficiently in UCM without having to master the details of base ClearCase.

UCM offers the convenience of an out-of-the-box solution; base ClearCase offers the flexibility to implement virtually any configuration management solution that you deem appropriate for your environment.

Branching and creating views

Branches are used in base ClearCase to enable parallel development. A *branch* is an object that specifies a linear sequence of versions of an element. Every element has one **main** *branch*, which represents the principal line of development, and may have multiple subbranches, each of which represents a separate line of development. For example, a project team may use the **main** branch for new development work while using a subbranch simultaneously for fixing a bug.

Subbranches can have subbranches. For example, a project team may designate a subbranch for porting a product to a different platform. The team may then decide to create a bug-fixing subbranch off that porting subbranch. You can create complex branch hierarchies. Figure 1 illustrates a multilevel branch hierarchy. As a project manager in such an environment, you need to ensure that developers are working on the correct branches. Developers work in views. A *view* is a work area for developers to create versions of elements. Each view includes a *config spec*, which is a set of rules that determines which versions of elements the view selects.

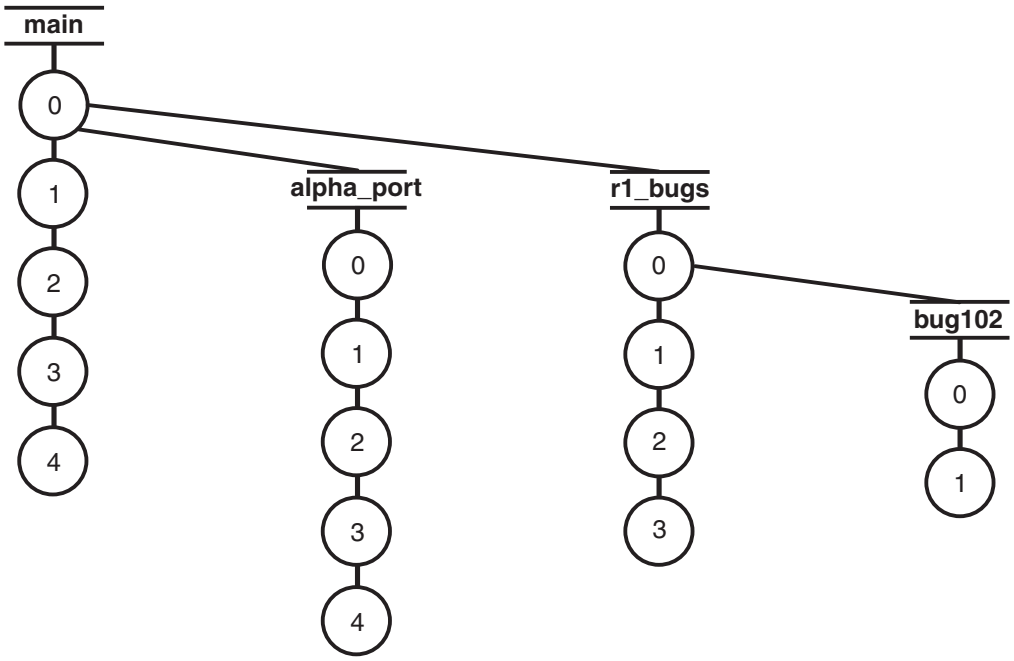


Figure 1. Branching hierarchy in base ClearCase

As project manager, you tell developers which rules to include in their *config specs* so that their views access the appropriate set of versions.

UCM uses branches also, but you do not have to manipulate them directly because it layers streams over the branches. A *stream* is a Rational ClearCase object that maintains a list of activities and baselines and determines which versions of elements appear in a developer's view. In UCM, a multiple-stream project contains one *integration stream*, which records the shared set of elements of the project, and multiple *development streams* in which developers work on their parts of the project in isolation from the team. The project integration stream uses one branch. Each development stream uses its own branch. You can create a hierarchy of development streams, and UCM creates the branching hierarchy to support those streams.

Although most customers use Rational ClearCase to implement a parallel development environment, UCM and base ClearCase also support serial development. In base ClearCase, you implement a serial development environment by having all developers work on the same branch. In UCM, you create a single-stream project, which contains one stream, the integration stream. All developers work on the integration stream rather than on development streams. Serial development is intended only for very small project teams whose developers work together closely.

As project manager of a UCM project, you need not write rules for config specs. Streams configure developers' views to access the appropriate versions on the appropriate branches.

Using components to organize files

As the number of files and directories in your system grows, you need a way to reduce the complexity of managing them. In UCM, you use components to simplify the organization of your files and directories. The elements that you group into a component typically implement a reusable piece of your system architecture.

By organizing related files and directories into components, you can view your system as a small number of identifiable components, rather than one large set of directories and files.

Creating and using baselines

A *baseline* identifies one version of every element in one or more components. You use baselines to identify the set of versions of files that represent a project at a particular milestone. For example, you may create a baseline called **beta1** to identify an early snapshot of project source files.

Baselines provide two main benefits:

- The ability to reproduce an earlier release of a software project
- The ability to tie together the complete set of files related to a project, such as source files, a product requirements document, a documentation plan, functional and design specifications, and test plans

UCM automates the creation process and provides additional support for performing operations on baselines. In base ClearCase, you can create the equivalent of a baseline by creating a version label and applying that label to a set of versions.

In UCM, baseline support appears throughout the user interface because UCM requires that you use baselines. When developers join a project, they must first populate their work areas with the contents of the recommended baseline of their parent stream. This method ensures that all team members start with the same set of shared files. In addition, UCM lets you set a property on the baseline to indicate the quality level of the versions that the baseline represents. Examples of quality levels include "project builds without errors," "passes initial testing," and "passes regression testing." By changing the quality-level property of a baseline to reflect a higher degree of stability, you can, in effect, promote the baseline.

Managing activities

In base ClearCase, you work at the version and file level. UCM provides a higher level of abstraction: activities. An *activity* is a Rational ClearCase object that you use to record the work required to complete a development task. For example, an activity may be to change a graphical user interface (GUI). You may need to edit several files to make the changes. UCM records the set of versions that you create to complete the activity in a *change set*. Because activities appear throughout the UCM user interface, you can perform operations on sets of related versions by identifying activities rather than having to identify numerous versions.

Because activities correspond to significant project tasks, you can track the progress of a project more easily. For example, you can determine which activities were completed in which baselines. If you use the UCM integration with IBM Rational ClearQuest®, you gain additional project management control, such as the ability to assign states and state transitions to activities. You can then generate reports by issuing queries such as "show me all activities assigned to Pat that are in the Ready state."

Enforcing development policies

A key part of managing the configuration management aspect of a software project is establishing and enforcing development policies. In a parallel development environment, it is crucial to establish rules that govern how team members access and update shared sets of files. Such policies are helpful in two ways:

- They minimize project build problems by identifying conflicting changes made by multiple developers as early as possible.
- They establish greater communication among team members.

These are examples of common development policies:

- Developers must synchronize their private work areas with the project recommended baseline before delivering their work to the project shared work area.
- Developers must notify other team members by e-mail when they deliver work to the project shared work area.

In base ClearCase, you can use tools such as triggers and attributes to create mechanisms to enforce development policies. UCM includes a set of common development policies, which you can set through the graphic user interface (GUI) or command-line interface (CLI). You can set these policies at the project and stream levels. In addition, you can use triggers and attributes to create new UCM policies.

Part 2. Working in UCM

Chapter 2. Understanding UCM

This chapter provides an overview of Unified Change Management (UCM), which is available with Rational ClearCase. Specifically, it introduces the main UCM objects and describes the tasks involved in managing a UCM project. Subsequent chapters describe in detail the steps required to perform these tasks.

Overview of the UCM process

In UCM, your work follows a cycle that complements an iterative software development process. Members of a project team work in a UCM *project*. A project is the object that contains the configuration information needed to manage a significant development effort, such as a product release. A project contains one main shared work area and typically multiple private work areas. Private work areas allow developers to work on activities in isolation. The project manager and integrator are responsible for maintaining the project shared work area. Work within a parallel development environment progresses as follows:

1. You create a project and identify an initial set of baselines of one or more components. A *component* is a group of related directory and file elements, which you develop, integrate, and release together. A *baseline* is a version of one or more components.
2. Developers join the project by creating their private work areas and populating them with the contents of baselines that are used by the team.
3. You or your developers create activities and the developers work on one activity at a time. An *activity* records the set of files that a developer creates or modifies to complete a development task, such as fixing a bug. This set of files associated with an activity is known as a *change set*.
4. When developers complete activities, they build and test their work in their private work areas.
5. They share their tested work with the project team by performing *deliver* operations. A deliver operation merges work from the developer's private work area to the project shared work area.
6. Periodically, the integrator builds the project executable files in the shared work area, using the delivered work.
7. If the project builds successfully, the integrator creates new baselines. In a separate work area, a team of software quality engineers performs more extensive testing of the new baselines.
8. Periodically, as the quality and stability of baselines improve, the integrator adjusts the promotion level attribute of baselines to reflect appropriate milestones, such as Built, Tested, or Released. When the new baselines pass a sufficient level of testing, the integrator designates them as the *recommended* set of baselines.
9. Developers perform *rebase* operations to update their private work areas to include the set of versions represented by the new recommended baselines.
10. Developers continue the cycle of working on activities, delivering completed activities, updating their private work areas with new baselines.

Figure 2 illustrates the connection between the project management, development, and integration cycles. This manual describes the steps performed by project managers and integrators. See *Developing Software* online help for information about the steps performed by developers.

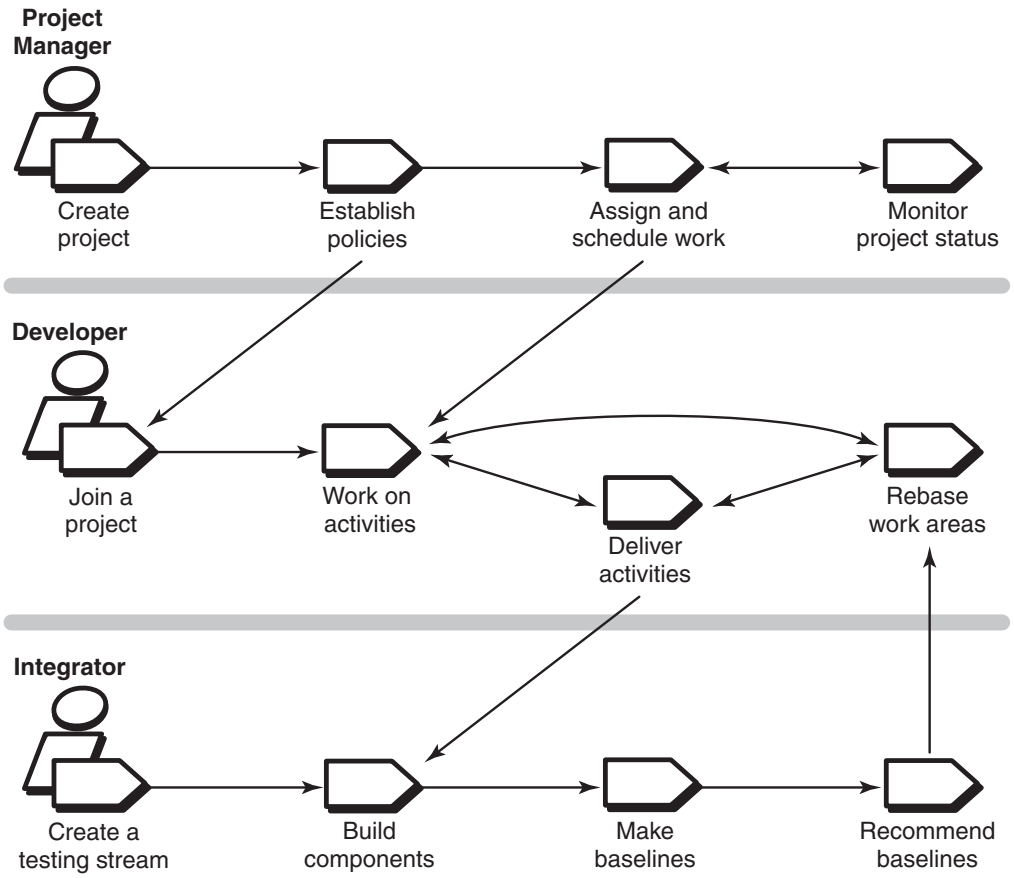
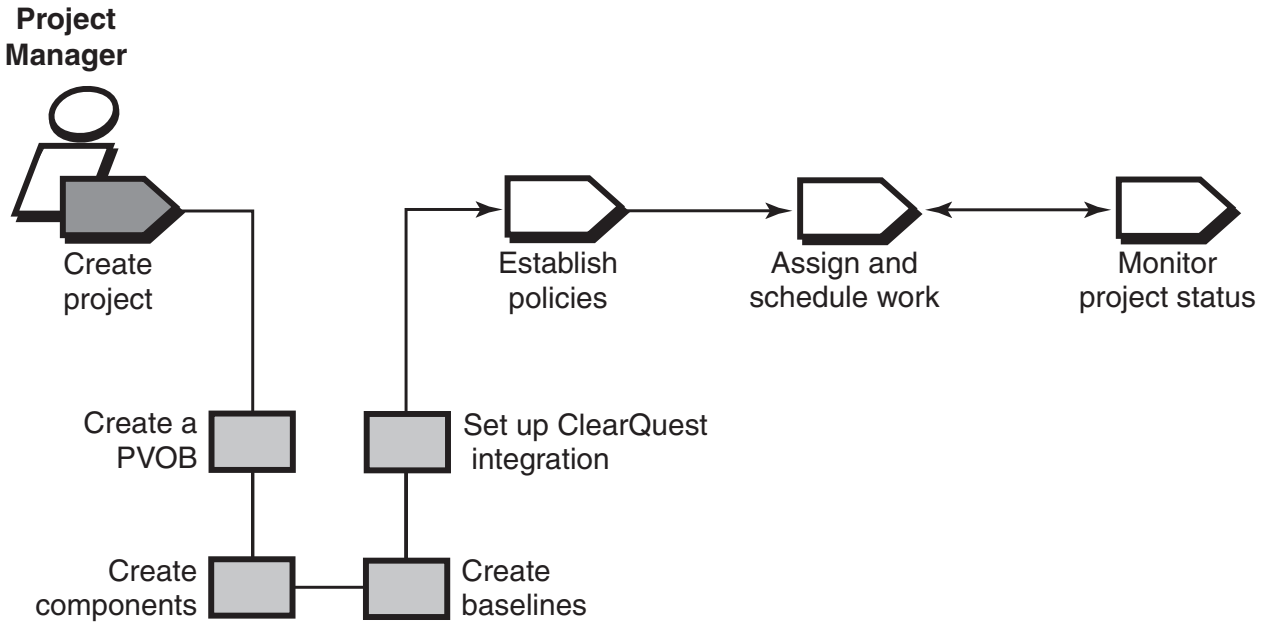


Figure 2. Project manager, developer, and integrator work flows

Creating the project



To create and set up a project, you must perform the following tasks:

- Create a repository for storing project information
- Create components that contain the set of files that the developers work on
- Create baselines that identify the versions of files with which the developers start their work

To use the UCM integration with Rational ClearQuest, you must perform additional setup tasks.

Creating a PVOB

File elements, directory elements, derived objects, and metadata are stored in a Rational ClearCase repository called a versioned object base (*VOB*). In UCM, each project must have a project VOB (PVOB). A *PVOB* is a special kind of VOB that stores UCM objects, such as projects, activities, and change sets. A PVOB must exist before you can create a project. Check with your site Rational ClearCase administrator to see whether a PVOB has already been created. For details on creating a PVOB, see "Creating the project VOB" on page 86.

Organizing directories and files into components

As the number of files and directories in your system grows, you need a way to reduce the complexity of managing them. Components are the UCM mechanism for simplifying the organization of your files and directories. The elements that you group into a component typically implement a reusable piece of your system architecture. By organizing related files and directories into components, you can view your system as a small number of identifiable components, rather than as one large set of directories and files.

The directory and file elements of a component reside physically in a VOB. The component object resides in a PVOB. Within a component, you organize directory and file elements into a directory tree (see Figure 3).

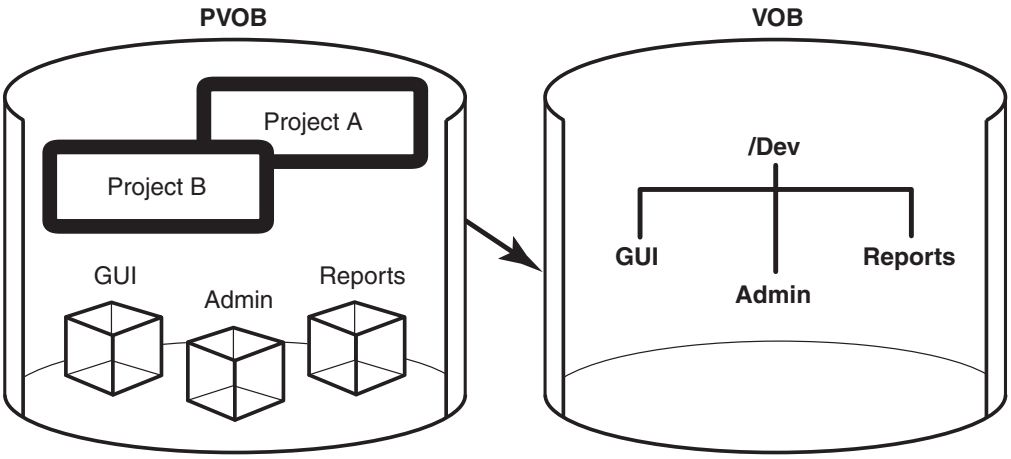


Figure 3. VOB containing multiple components

The directory trees for the **GUI**, **Admin**, and **Reports** components appear directly under the VOB root directory. You can convert existing VOBs or directory trees within VOBs into components, or you can create a component from scratch. For details on creating a component from scratch, see "Creating components for storing elements" on page 88. For details on converting a VOB into a component, see "To make a VOB into a component" on page 97.

Shared and private work areas

A work area consists of a view and a stream. A *view* is a directory tree that shows a single version of each file in your project. A *stream* is a Rational ClearCase object that maintains a list of activities and baselines and determines which versions of elements appear in your view.

A project contains one *integration stream*, which records the project baselines and enables access to shared versions of the project elements. The integration stream and a corresponding integration view represent the project main shared work area.

In a typical project, each developer has a private work area, which consists of a development stream and a corresponding development view. The *development stream* maintains a list of the developer's activities and determines which versions of elements appear in the developer's view.

When you create a project from the UCM graphic user interface (GUI), the integration stream is created for you. If you create a project from the command-line interface, you need to create the integration stream explicitly. Developers create their development streams and development views when they join the project. See *Developing Software* online help for information on joining a project.

Stream hierarchies

In the basic UCM process, the integration stream is the only shared work area for the entire project. In a multiple-stream project, you may want to create additional shared work areas for developers who are working together on specific parts of the project. You can accomplish this by creating a hierarchy of development streams. For example, you can create a development stream and designate it as the shared

work area for developers working on a particular feature. Developers then join the project at the development stream level (rather than at the integration stream) and create their own development streams and views under the development stream for this feature. The developers deliver work to and rebase their streams to recommended baselines in the development stream for the feature. See “Choosing a stream strategy” on page 34 for details on development stream hierarchies.

Single-stream projects

Although UCM is typically used to implement a parallel development environment, UCM also supports serial development by letting you create a single-stream project. A single-stream project contains one stream, the integration stream. All developers work on the integration stream rather than on development streams. Developers have their own views that are attached to the integration stream. Serial development should be used only for very small project teams whose developers work together closely. See “Choosing a stream strategy” on page 34 for details on single-stream projects.

Starting from a baseline

After you create project components or select existing components, you must identify and recommend the baseline or baselines that serve as the starting point for the team of developers. Just as a component represents a collection of elements, a baseline represents a collection of versions within a component. An ordinary baseline identifies one version of every element visible in a single component (see Figure 4).

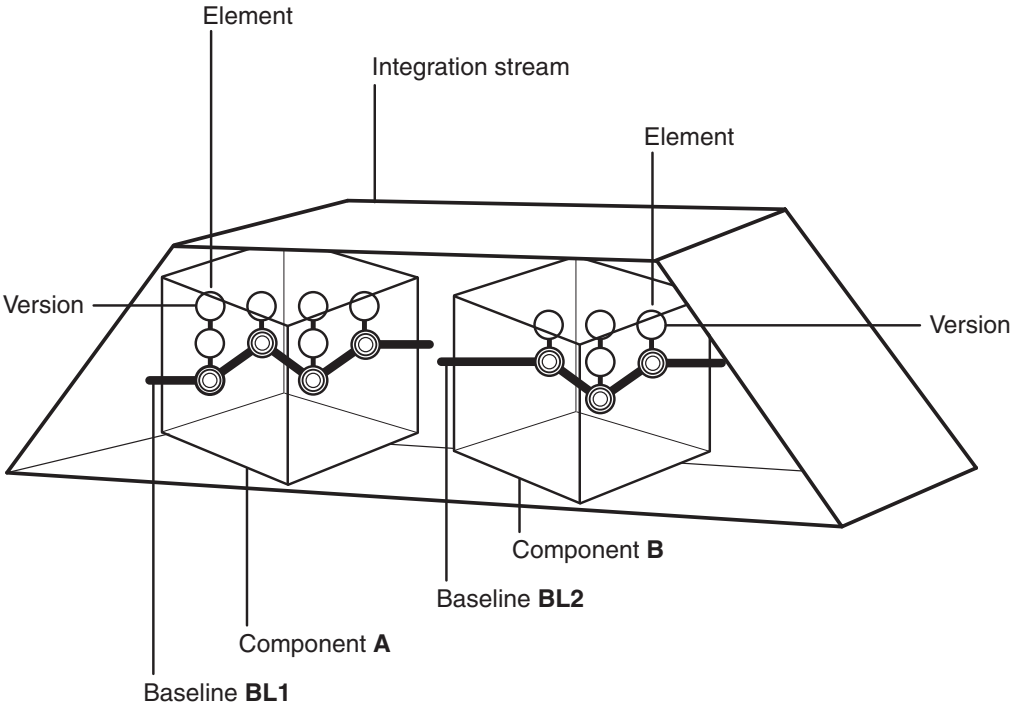


Figure 4. Baselines of two components

Baselines named **BL1** and **BL2** in the integration stream identify the versions in component **A** and component **B**, respectively.

When developers join the project, they populate their work areas with the versions of directory and file elements represented by the recommended baselines of the project. Alternatively, developers can join the project at a feature-specific

development stream level, in which case they populate their work areas with the development stream's recommended baselines. This practice ensures that all members of the project team start with the same set of files.

Composite baselines

If your project team works on multiple components, you may want to use a composite baseline. A *composite baseline* selects baselines in other components (see Figure 5).

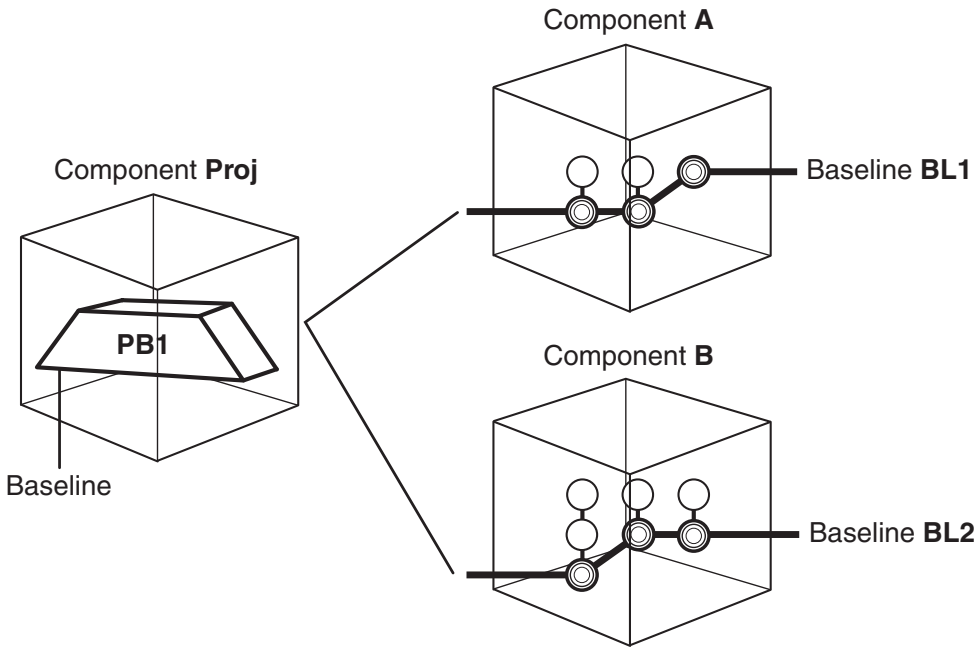


Figure 5. Composite baseline

The **PB1** composite baseline selects baselines **BL1** and **BL2** of components **A** and **B**, respectively. The **Proj** component does not contain any elements of its own. It contains only the composite baseline that selects the recommended baselines of the project components. By using a composite baseline in this manner, you can identify one baseline to represent multiple baselines, and, by extension, the entire project.

Baselines and their uses

A baseline is a snapshot of a component at a particular time. It comprises the set of versions that are selected in the stream at the time the baseline was made. When a new stream is configured, baselines are used to specify which versions are to be selected in that stream. Baselines are immutable so that a particular configuration can be reproduced as needed and streams that use the same set of baselines are guaranteed to have the same configuration. Therefore, the set of versions included in a baseline cannot be modified.

Baselines that are created in the context of a stream are ordered relative to each other (see Figure 6). Within a single stream, an old baseline is referred to as an *ancestor* of a newer baseline. The newer baseline is called a *descendant* of the old baseline. The closest ancestor of a baseline is its *predecessor*. The *foundation* baselines (or the foundation set) of a stream, which are created in a different stream, are the predecessors of the first baselines created in this stream.

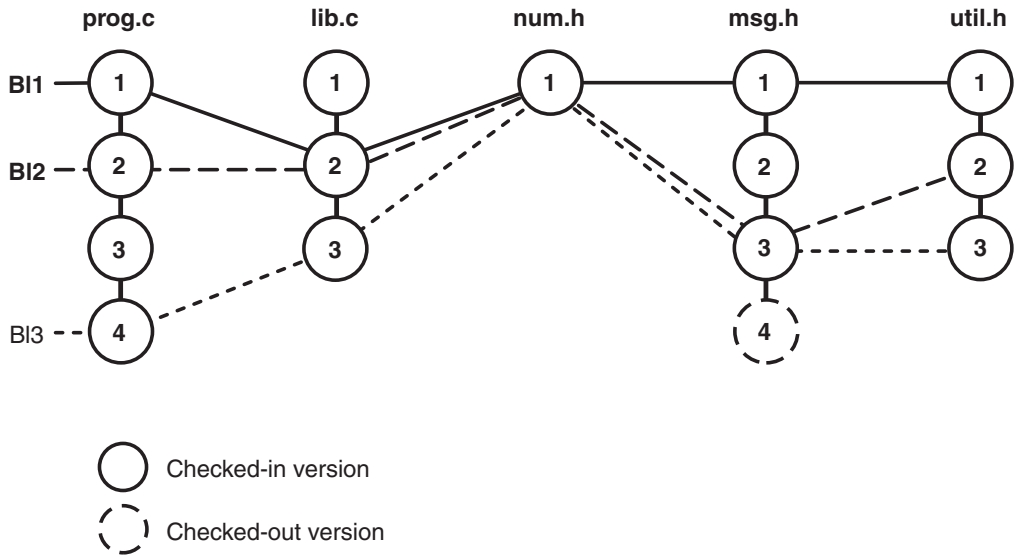


Figure 6. Baseline predecessors and descendants

In Figure 6, baseline **BL1** is the predecessor of baseline **BL2** and baseline **BL2** is a descendant of baseline **BL1**. When baseline **BL2** was created, there were new versions of prog.c, msg.h, and util.h, but for the files lib.c and num.h, baseline **BL2** falls back to the baseline **BL1** versions. Similarly, baseline **BL3** is a descendant of baseline **BL2**; and baselines **BL1** and **BL2** are predecessors of baseline **BL3**. Baseline **BL3** captures changes made after baseline **BL2** was created, but it uses the baseline **BL1** version of num.h and the baseline **BL2** version of msg.h. Because version 4 of msg.h is checked out, it is not included in baseline **BL3**.

In the relationship among baselines, a descendant contains its predecessors so that, for example, all changes captured in baseline **BL2** are also in baseline **BL3**.

The relationship between a baseline and a component is very similar to the relationship between a version and an element. For example, baselines exist in streams, but versions exist on branches. Both baselines and versions have predecessors.

Baselines have the following uses:

- Record work done and mark milestones.
- Define stream configurations.
- Provide access to delivered work.

Baselines and streams

Baselines and streams have a mutual relationship: baselines are produced by streams, and streams use baselines for their configuration. A stream is configured with a set of baselines, called its *foundation*, which defines which versions are selected in that stream. Views that are attached to the stream see the versions of elements that are selected by the foundation baselines and any new versions that are created from changes that are made in the stream.

A stream includes a baseline from every component that it needs to access, both for modifiable and non-modifiable components.

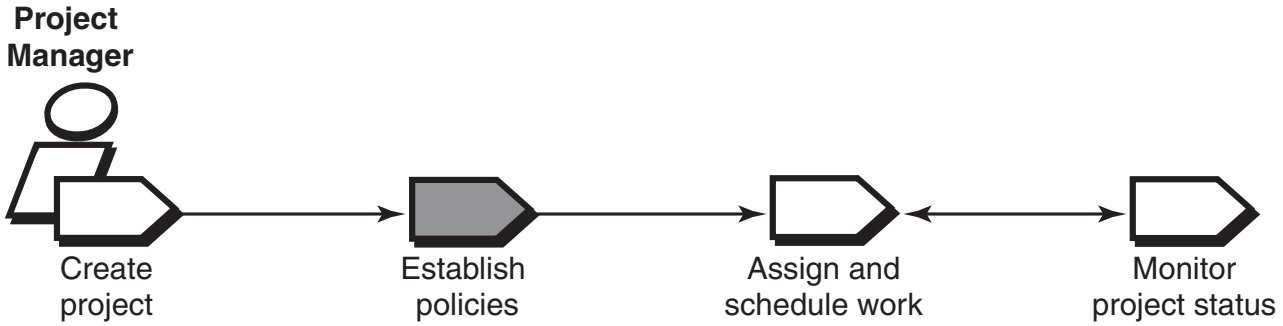
Setting up the UCM integration with Rational ClearQuest

You can use UCM without Rational ClearQuest, the change request management tool, but the UCM integration with RationalClearQuest adds significant project management and activity management capabilities. When you enable a UCM project to work with Rational ClearQuest, the integration links all UCM activities to Rational ClearQuest records. You can then take advantage of the UCM and Rational ClearQuest state transition model and the query features of Rational ClearQuest. Reporting and charting features are available on the Windows® system. These features allow you to do the following:

- Assign activities to developers
- Use states and state transition rules to manage activities
- Generate reports based on database queries
- Select additional development policies to be enforced

To set up the UCM integration with Rational ClearQuest, you enable a Rational ClearQuest schema to work with UCM or use a predefined schema that is enabled for UCM. Then, you either create a new Rational ClearQuest user database or upgrade an existing Rational ClearQuest user database to use the UCM-enabled schema. When the Rational ClearQuest environment is established, you enable your UCM project to work with Rational ClearQuest. For additional information about the integration, see "Overview of the UCM integration with Rational ClearQuest" on page 26.

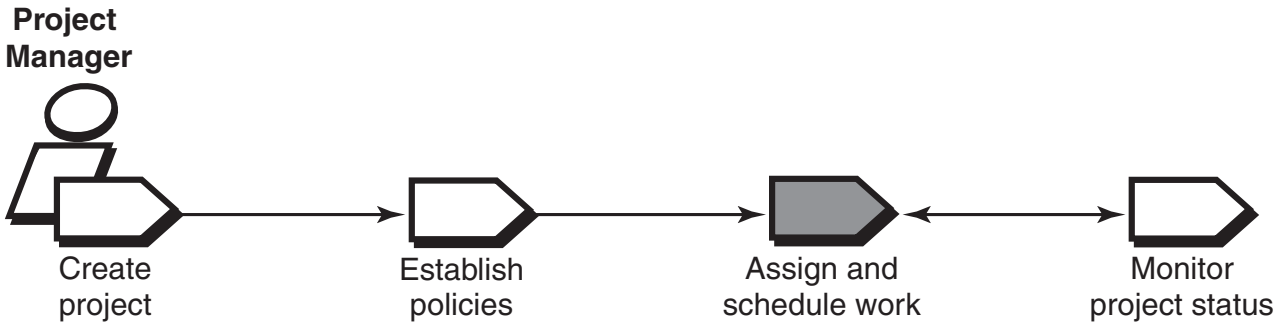
Setting policies



UCM includes a set of policies that you can set to enforce development practices among members of the project team. By setting policies, you can improve communication among project team members and minimize the problems you may encounter when integrating their work. For example, you can set a policy that requires developers to update their work areas with the latest recommended baseline of the project before they deliver work. This practice reduces the likelihood that developers will need to work through complex merges when they deliver their work. For a description of all policies you can set in UCM, see Chapter 4, "Setting policies," on page 63. You can set policies on projects and streams.

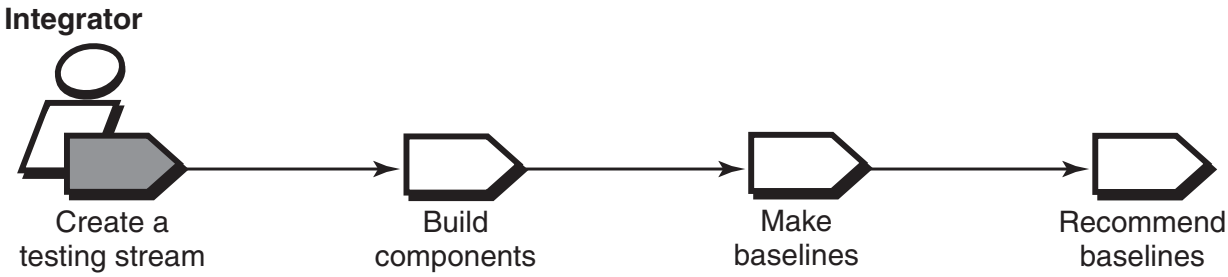
In addition to the set of policies that UCM provides, you can create triggers on UCM operations to enforce customized development policies. See Chapter 8, "Using triggers to enforce UCM development policies," on page 129 for details about creating triggers.

Assigning work



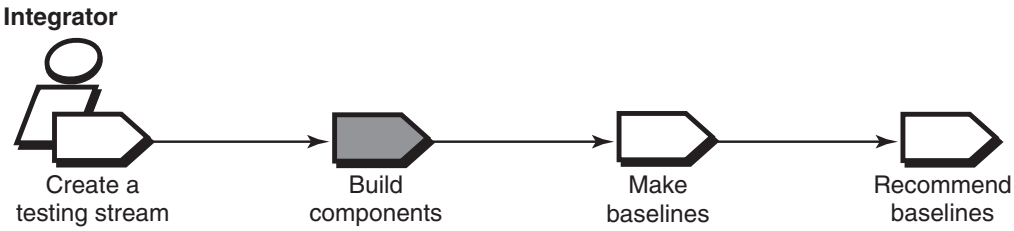
This task is optional and is possible only if you use the UCM integration with Rational ClearQuest. As project manager, you are responsible for identifying and scheduling the high-level tasks for your project team. In some organizations, the project manager creates activities and assigns them to developers. In other organizations, the developers create their own activities. See "Assigning activities" on page 102 for details on creating and assigning activities in a Rational ClearQuest user database.

Creating a testing stream



In your role as project integrator, you are responsible for building the work delivered by developers, creating baselines, and testing those baselines. When you make baselines in the integration stream, you lock the stream to prevent developers from delivering work. This practice ensures that you work with a static set of files. It is acceptable to perform quick validation tests of the new baselines in the integration stream. However, you should not lock the integration stream for a long time because you will create a backlog of deliveries. To perform more rigorous testing, such as regression testing, you should create a development stream to be used solely for stabilizing and testing baselines. See "Creating a development stream for testing baselines" on page 106 for details on creating a testing stream.

Building components



Before you make new baselines, build the components in the integration stream by using the current baselines plus any work that developers have delivered to the stream since you created the current baselines. Lock the integration stream before you build the components to ensure that you work with a static set of files. If the build succeeds, you can make baselines that select the latest delivered work. If your project uses feature-specific development streams, perform this task on those streams and on the integration stream.

Rational ClearCase MultiSite consideration

Product Note: Rational ClearCase LT does not support Rational ClearCase MultiSite®.

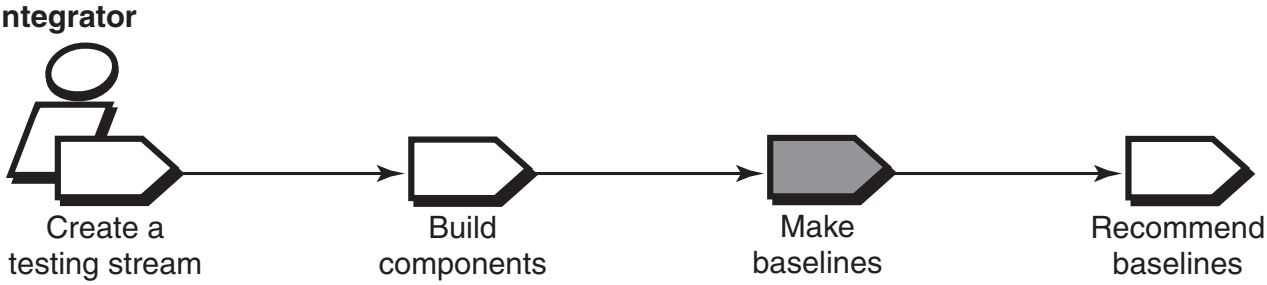
In most cases, developers complete the deliver operations that they start. If your project uses Rational ClearCase MultiSite, you may need to complete some deliver operations before you can build the components. Many customers use Rational ClearCase MultiSite, a product layered on Rational ClearCase, to support parallel software development across geographically distributed project teams. Rational ClearCase MultiSite lets developers work on the same VOB concurrently at different locations. Each location works on its own copy of the VOB, known as a *replica*.

To avoid conflicts, Rational ClearCase MultiSite uses an exclusive-right-to-modify scheme, called *mastership*. VOB objects, such as streams and branches, are assigned a *master replica*. The master replica has the exclusive right to modify or delete these objects.

In a Rational ClearCaseMultiSite configuration, a team of developers may work at a remote site, and the integration stream of the project may be mastered at a different replica than the developers' development streams. In this situation, the developers cannot complete deliver operations to the integration stream. As project integrator, you must complete these deliver operations. UCM provides a variation of the deliver operation called a *remote delivery*. When UCM determines that the integration stream is mastered at a remote site, it makes the deliver operation a remote delivery and posts the delivery, which starts the deliver operation but does not merge any versions. You then find the posted delivery and complete the deliver operation at the remote site.

For information on completing remote deliver operations, see "Finding work that is ready to be delivered" on page 113.

Making a baseline



To ensure that developers stay in sync with each other's work, make new baselines regularly. A new baseline includes the work developers have delivered to the parent stream since the last baseline. If your project uses feature-specific development streams, perform this task on those streams and on the integration stream. In some environments, the lead developer working on a feature may assume the role of integrator for a feature-specific development stream.

After making a baseline

After your team of software quality engineers tests the new baseline more extensively and determines that it is stable, you make the baseline the recommended baseline.

The rebase operation

To take advantage of a newly recommended baseline, developers update their work areas with the new baseline by performing a rebase operation (see Figure 7).

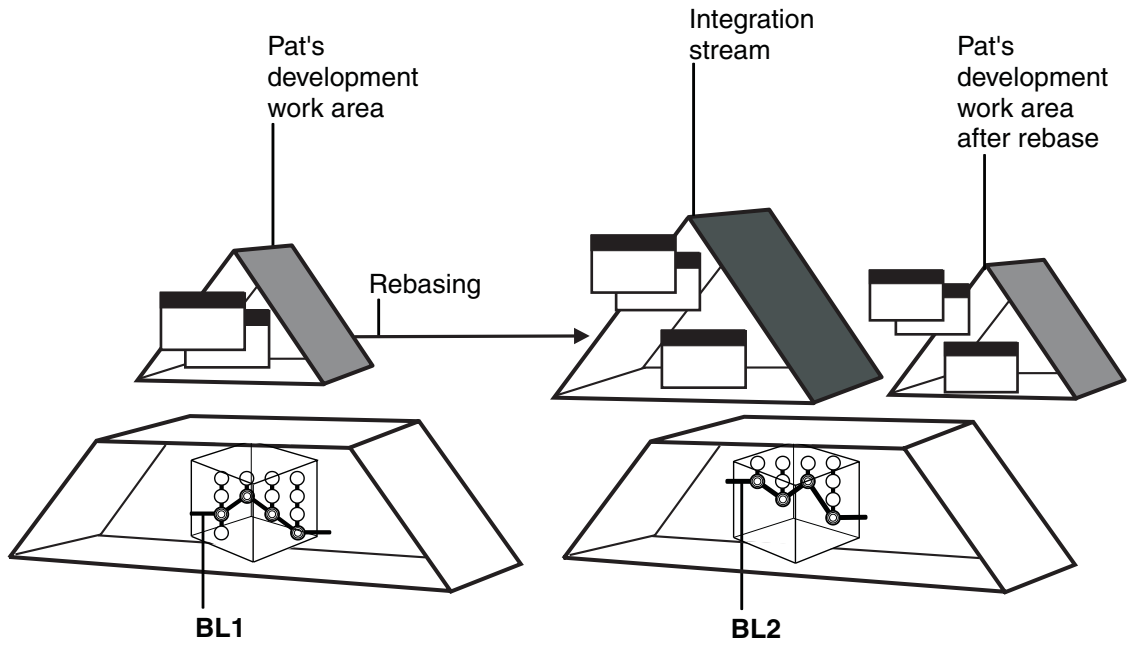


Figure 7. Rebase operation

A component in Pat's development stream is configured with baseline **BL1**. A rebase operation changes the configuration of the stream to baseline **BL2** from the

integration stream. The rebase operation merges files and directories from the integration stream or feature-specific development stream to the development stream.

A rebase operation reconfigures a stream by adding, dropping, or replacing one or more of the stream foundation baselines. It is typically used to advance a stream's configuration, that is, to replace its current foundation baselines with descendant ones. For more information about baselines, see "Baselines and their uses" on page 14.

Foundation baselines of the target stream are replaced with the set of recommended baselines from the source stream.

If an element in the stream being rebased contains any changes, the rebase operation merges the changes into the latest version of that element in the stream, thereby creating a new version. All such new versions are captured in the change set of the integration activity that the rebase operation creates.

The rebase operation changes the foundation baselines of a stream. Baselines provide a configuration that includes delivered work. If a specific stream must use work that has been included in a baseline in an appropriate stream, you rebase the specific stream to the desired baseline.

In a rebase operation, a developer selects one or more baselines to add or drop from the configuration of the stream. Just as a view can only select one version of an element, a stream can only include one baseline for each component. If more than one baseline were allowed in a stream foundation for a particular component, the selection of versions in that component would be ambiguous.

During a rebase operation, the specified baselines replace the current baselines, if any, for their components. Changes that have been made on the stream are merged into new versions, if necessary. A deliver operation always involves merging elements. A rebase operation only involves merging if elements that have been modified in the stream also have new versions selected by the new baseline.

Directions of rebase operations

The relationship between the old baseline and the new baseline for a component defines the *direction* of the rebase operation. If the new baseline is a descendant of the old baseline, the rebase operation *advances*. That is, the stream is configured with work done in other streams from the same starting point.

Conversely, if the new baseline is an ancestor of the old baseline, the rebase operation is said to *revert*. That is, the stream moves back to an earlier baseline. However, if two baselines share an ancestor, but both contain significant development work or if the relationship between the baselines cannot be established, the rebase operation is *lateral*. A lateral rebase operation is typically used to configure a new version of a read-only component, for example, a compiler.

A single rebase operation might involve many baselines; the rebase direction is determined on each baseline in the rebase operation. Thus, in one rebase operation, a stream might advance for one component, revert for another component, rebase laterally for a third component, and leave a fourth component unchanged.

Advance rebase operations

Most development streams over the life of a project advance from predecessor baselines to descendant baselines that integrate work performed in the project. Streams are usually allowed to advance (see Figure 8).

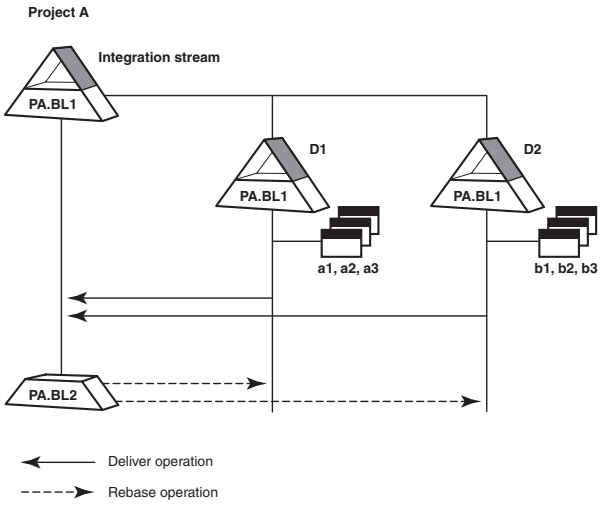


Figure 8. Advance rebase operation

Development streams **D1** and **D2** deliver work (activities **a1**, **a2**, **a3**, **b1**, **b2**, and **b3**) to the integration stream. In the integration stream, a descendant baseline **PA.BL2** is made to capture the work. To include the new work in their configurations, streams **D1** and **D2** rebase to descendant baseline **PA.BL2**.

Restrictions on advancing rebase operations occur when the descendant baseline is not from the parent stream. In a rebase operation, you can select a baseline from any stream. For example, you can select a baseline from a stream that is a descendant of the stream foundation that is in a stream that is not its parent stream. Thus, a development stream could rebase to a baseline created in a sibling development stream. Therefore, the rebasing stream could acquire work that has not been delivered to the parent stream. If several streams were allowed to deliver the same work, there would be confusion during the merge operation. (Alternative target delivery is a special case, and a project manager can set policies to allow a stream to accept changes that did not originate in the delivering stream).

A common example of an advance rebase operation occurs when a project uses a test stream (see Figure 9). The project integrator creates the baseline **PA.BL1** for a milestone. The work to stabilize the code in the baseline is done on the test stream **DS** that is dedicated to this task. Because the project **A** integration stream can have more activities delivered as the baseline **PA.BL1** is being tested, the integration stream is not used. The test stream is isolated from deliver operations ongoing in the parent stream, the project **A** integration stream.

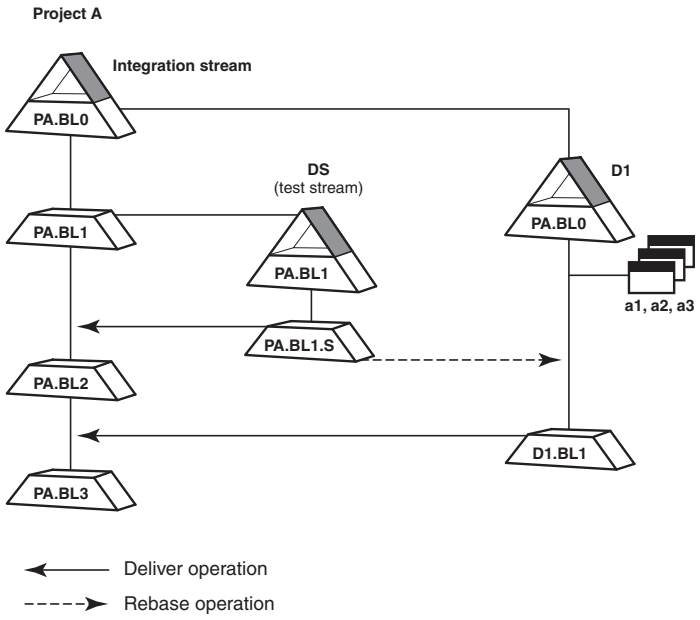


Figure 9. A test stream to stabilize a baseline

Although the **PA.BL1.S** baseline is a descendant of their current foundation baseline **PA.BL0**, development streams like **D1** are not allowed to rebase to it. If this rebase operation were allowed, the development streams could deliver the build stabilization work before stream **DS** does. Therefore, the test stream **DS** must first deliver its work in the baseline **PA.BL1.S** to the parent of the development streams, the project **A** integration stream. When the work from test stream **DS** is contained in the parent stream and the baseline that contains that work is ready to be released, the project integrator can recommend the baseline. Then the development streams can rebase to the baseline **PA.BL1.S** from the test stream.

Tip:: The deliver operation changes the relationship between baselines in a stream. In Figure 9, when a new baseline **PA.BL2** is created in the project **A** integration stream, it becomes a descendant of **PA.BL1** as with any baseline, but it also becomes a descendant of **PA.BL1.S** from the test stream. Because baseline **PA.BL1.S** was delivered to the integration stream, baseline **PA.BL2** contains baseline **PA.BL1.S**, which is a requirement of the predecessor and descendant relationship.

Revert rebase operations

A revert rebase operation is used when a stream needs to remove some unwanted changes. If a baseline has some serious problems and there are no changes in the component in the context of that stream, a stream can revert to an ancestor baseline of that component involved in the unwanted baseline. The merge algorithm cannot remove the unwanted changes.

If developers need to use a questionable baseline, have them use it in a read-only stream. If a stream that has made changes needs to revert, the developer has to explicitly remove the new versions before rebasing. A read-only stream is guaranteed to have no changes. If developers encounter difficulty with the questionable baseline, in read-only streams they can always revert to a stable ancestor baseline.

Lateral rebase operations

A lateral rebase operation occurs when the baseline to which the stream rebases and the baseline in the stream being rebased have no relationship to each other or if the relationship between the baselines is too distant for merging. For example, rebasing to an imported baseline is usually a lateral rebase operation, except when the imported baseline is the ancestor of the current baseline. (This condition makes a revert rebase operation.) Imported baselines have no predecessors; therefore, they are related only to their descendant baselines.

A lateral rebase operation is typically used in a project that has vendor software, for example, a set of compilers or other tools. The project does no development on these tools, but it frequently receives a new release of the tools. The new release is added to source control in a VOB, certified, and, if it passes, labeled. The label is then imported into a baseline; rebasing to this imported baseline is a lateral rebase operation.

Summary of rules for rebasing a stream

This section summarizes the rules for rebase operations. You can rebase a stream to a baseline that meets any of the following criteria:

- The baseline is not from the stream that is being rebased.
- The baseline is labeled. (Baselines created by deliver operations are not labeled by default. You can change the labeling status of a baseline.)

Additional rules apply to integration streams and development streams in selecting a baseline. The following are general rules that apply to all types of rebase operations:

- An integration stream can be rebased only to a baseline that is created in another project or to an imported or initial baseline of that project.
- A development stream can be rebased to a baseline that meets one of the following criteria:
  - The baseline was created in its parent stream.
  - The baseline is in the foundation set of its parent stream.
  - The baseline is an ancestor of the foundation baseline of the parent of the development stream and was created on the same stream as the foundation baseline of the parent stream.
  - The baseline was created in a stream other than its parent stream and is contained in its parent stream. (A baseline is contained in another baseline if all changes in the first baseline are included in the second baseline.)

You need to satisfy only the general rules if you are adding a component to a stream.

Note: Read-Only streams and nonmodifiable components in a development stream are exempt from the general rules. However, if the modifiability of the component changes in the future, the development stream might not be able to modify the component at the baseline with which it is configured. The development stream might be able to modify the component at the baseline it is configured with if the baseline is contained in its parent stream for this component. Otherwise, it may not until the baseline is rebased to a compatible baseline for that component.

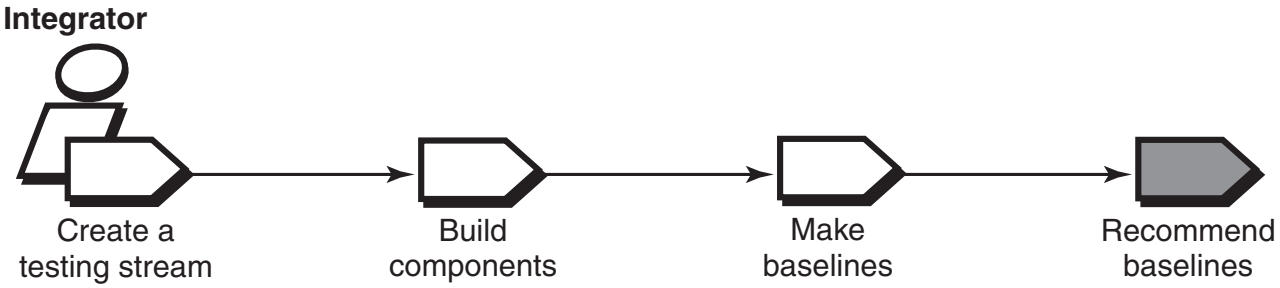
Rebase typically advances the configuration of a stream, that is, it replaces the current foundation baselines of the stream with more recent ones (see “Advance rebase operations” on page 21). However, under certain conditions, rebase can be

used to revert a baseline (see “Revert rebase operations” on page 22); to add or drop a component in the configuration of a stream; and to switch to a baseline that is neither an ancestor nor a descendant of the current foundation (see “Lateral rebase operations” on page 23). When you advance, revert, drop, or switch a baseline, you need to satisfy the general rules and the following additional ones:

- To advance the configuration of a stream, the new baseline must contain the current foundation baseline.
- To revert or drop a baseline for a component in a stream, one of the following conditions must be met:
  - The component is nonmodifiable.
  - The component is modifiable but has not been modified in the stream, and the component is not in the configuration of any child streams.
- To switch to a baseline that is neither an ancestor nor a descendant of the current foundation, one of the following conditions must be met:
  - The component is nonmodifiable.
  - The component is modifiable but has not been modified in the stream, and the component is not in the configuration of any child streams.
  - The component has been modified, but the new baseline contains the current foundation baseline; and the component is not in the configuration of any child streams.

These rules ensure that any changes made in a stream are not lost when the configuration changes.

Recommending the baseline



As work on your project progresses and the quality and stability of the components improve, change the baseline promotion level attribute to reflect important milestones (see Figure 10).

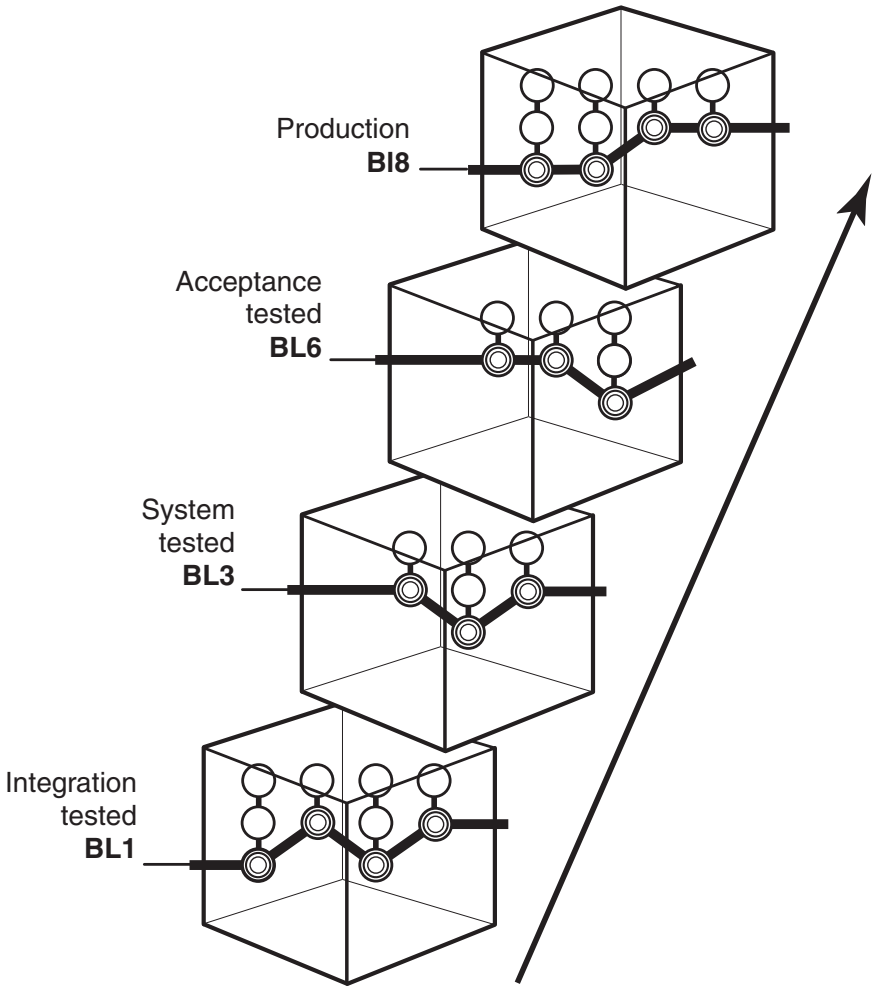


Figure 10. Promoting baselines

The promotion level attribute typically indicates a level of testing. For example, Figure 10 shows the evolution of baselines through three levels of testing; the **BL8** baseline is ready for production.

Recommended baselines

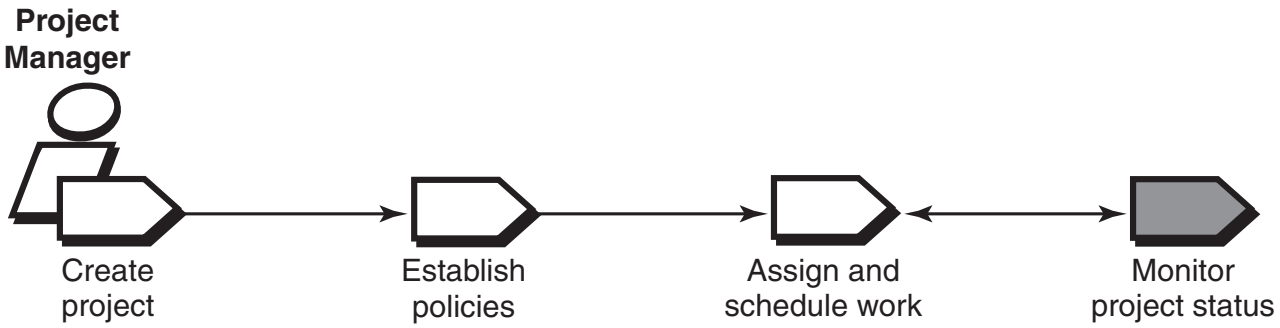
When baselines pass the level of testing required to be considered stable, make them the recommended set of baselines. Developers then rebase their development streams to the recommended baselines. You can set a policy that requires developers to rebase their development streams to the set of recommended baselines before they deliver work. This policy helps to ensure that developers update their work areas whenever a baseline passes an acceptable level of testing.

Every stream has foundation baselines. The foundation baselines of an integration stream are its default recommended baselines. A development stream has no default recommended baselines.

A stream can recommend a baseline if certain rules are true. These rules establish consistency in child streams. If a child stream rebases to the new recommended baseline and subsequently delivers activities to its default target, only activities created on the development stream need to be delivered. The rules also prevent a stream from reverting to the configuration of a development stream that has rebased to baselines that are ahead of the current recommended baselines.

For more information about recommending baselines, see “Recommending the baseline” on page 118.

Monitoring project status



Several tools are provided to help you track the progress of your project:

- The UCM integration with Rational ClearQuest includes some Rational ClearQuest queries, which you can use to retrieve information about activities in your project. For example, you can see all activities that are in an active state or all active activities assigned to a particular developer. In addition, you can create customized Rational ClearQuest queries.
- The Compare Baselines GUI compares any two baselines of a component and displays the differences in activities and versions associated with each baseline. You can use this feature to determine when a particular feature was included in a baseline.
- The Component Tree Browser (Windows only) displays the baseline history of a component. The GUI includes a feature that lets you filter the display so that you see only specified streams or baselines at or above a specified promotion level.
- The Rational ClearCase Report Builder and Report Viewer (Windows only) let you generate and view reports specific to your project environment. The Report Builder provides a set of reports organized by Rational ClearCase object, such as project, stream, element, and view. In addition, you can customize the procedures used to generate and display reports.

For more information about using these tools, see “Monitoring project status” on page 122.

Overview of the UCM integration with Rational ClearQuest

This section describes the following concepts related to the UCM integration with Rational ClearQuest.

Associating UCM and Rational ClearQuest objects

Setting up the UCM integration with Rational ClearQuest links UCM and Rational ClearQuest objects (see Figure 11).

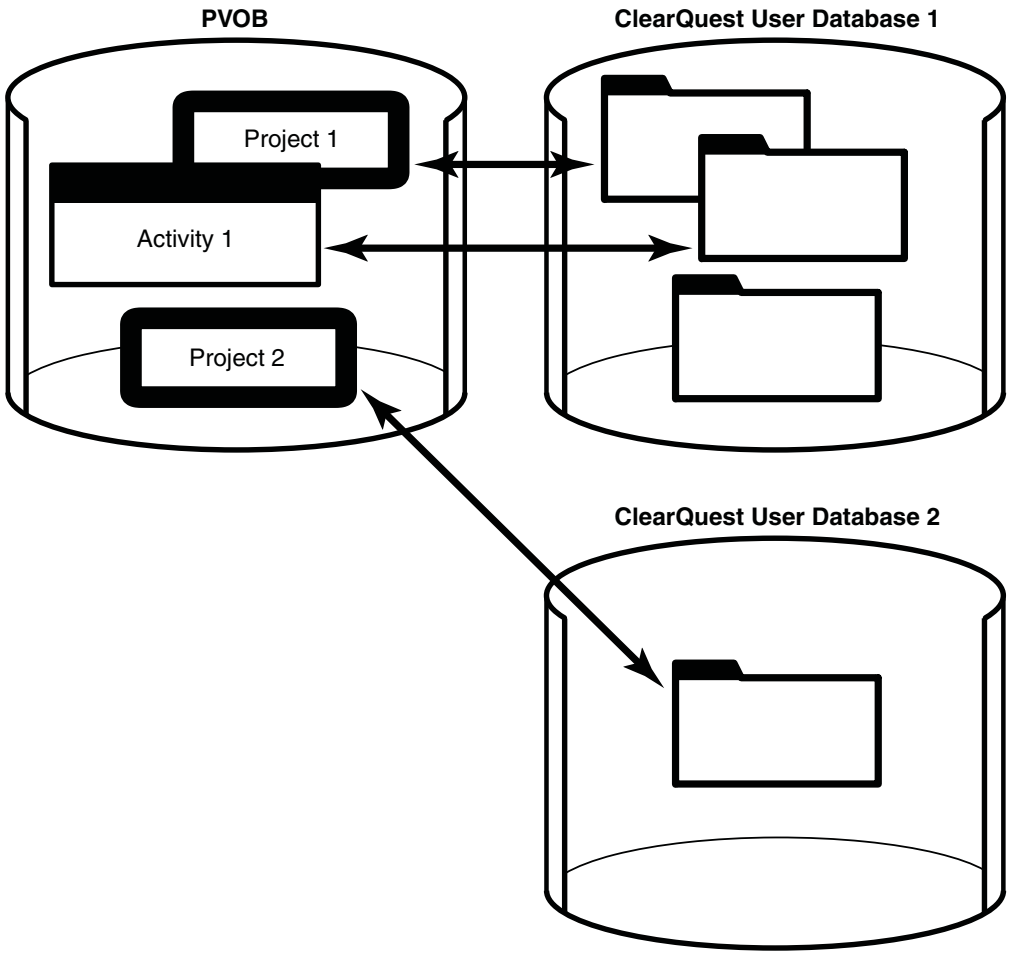


Figure 11. Association of UCM and Rational ClearQuest objects in integration

The links between the project and activity objects in the PVOB and the record objects in the Rational ClearQuest user database show the bidirectional linking of these objects. When you enable a project to link to a Rational ClearQuest user database, the integration stores a reference to that database in the project's PVOB. Every project that is enabled for Rational ClearQuest is linked to a project record of record type **UCM_Project** in the Rational ClearQuest user database.

Every activity in a project that is enabled for Rational ClearQuest is linked to a record in the database. An activity's headline is linked to the headline field in its corresponding Rational ClearQuest record. If you change an activity's headline in a Rational ClearCase repository, the integration changes the headline in the Rational ClearQuest user database to match the new headline, and the reverse is also true. Similar to the linking of the activity headline, an activity's ID is linked to the ID field in its Rational ClearQuest record.

It is possible for a Rational ClearQuest user database to contain some records that are linked to activities and some records that are not linked. In Figure 11, **ClearQuest User Database 1** contains a record that is not linked to an activity. You may encounter this situation if you have a Rational ClearQuest user database in place before you adopt UCM. As you create activities, the integration creates corresponding Rational ClearQuest records. However, any records that existed in

that user database before you enabled it to work with UCM remain unlinked. In addition, UCM does not link a record to an activity until a developer sets work to that record.

Schema enabled for UCM

In Rational ClearQuest, a schema is the definition of a database. To use the integration, you must create a new Rational ClearQuest user database or upgrade a current Rational ClearQuest user database that is based on a schema that is enabled for UCM. Such a schema contains certain fields, scripts, actions, and state types. You can use predefined schemas that are enabled for UCM. You can also enable a custom schema or another predefined schema to work with UCM. For information about schemas enabled for UCM, see "Deciding which schema to use" on page 59.

State types

States are used to track the progress of change requests from submission to completion. A state represents a particular stage in this progression. Each movement from one state to another is a state transition. The UCM integration with Rational ClearQuest uses a particular state transition model. To implement this model, the integration uses state types. A state type is a category of states that UCM uses to define state transition sequences. You can define as many states as you want, but all states in a UCM-enabled record type must be based on one of the following state types:

- Waiting
- Ready
- Active
- Complete

Multiple states can belong to the same state type. However, you must define at least one path of transitions between states of state types as follows: Waiting to Ready to Active to Complete. For details on state types, see "Setting state types" on page 78.

Queries in a Rational ClearQuest schema enabled for UCM

A UCM-enabled schema includes some Rational ClearQuest queries. When you create or upgrade a Rational ClearQuest user database to use a UCM-enabled schema, the UCM integration with Rational ClearQuest installs these queries in two subfolders of the Public Queries folder in the user database workspace. These queries make it easy for developers to see which activities are assigned to them and for project managers to see which activities are active in a particular project. For details on these queries, see "Querying Rational ClearQuest user databases" on page 124.

Chapter 3. Planning the project

This chapter describes the issues you need to consider in planning to use one or more UCM projects as your configuration management environment in Rational ClearCase. You should write a configuration management plan before you begin creating projects and other UCM objects. After you create your plan, see Chapter 6, "Setting up the project," on page 85 for information on how to implement it.

Using the system architecture as the starting point

Essential to developing and maintaining high-quality software is the definition of the system architecture. The IBM Rational Unified Process states that defining and using a system architecture is one of the best practices to follow in developing software. A system architecture is the highest level concept of a system in its environment. The IBM Rational Unified Process states that a system architecture encompasses the following:

- The significant decisions about the organization of a software system
- The selection of the structural elements and their interfaces of 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 subsystems
- The architectural style that guides this organization, these elements, and their interfaces, their collaborations, and their composition

A well-documented system architecture improves the software development process. It is also the ideal starting point for defining the structure of your configuration management environment.

Mapping system architecture to components

Just as different types of blueprints represent different aspects of building architecture (for example, floor plans, electrical wiring, and plumbing), a good software system architecture contains different views to represent its different aspects. The IBM Rational Unified Process defines an architectural view as a simplified description (an abstraction) of a system from a particular perspective or vantage point, covering particular concerns and omitting entities that are not relevant to this perspective.

The IBM Rational Unified Process suggests using multiple architectural views. Of these, the implementation view is most important for configuration management. The implementation view identifies the physical files and directories that implement the system's logical packages, objects, or modules. For example, your system architecture may include a licensing module. The implementation view identifies the directories and files that make up the licensing module.

From the implementation view, you should be able to identify the set of UCM components you need for your system. You typically develop, integrate, and release components together. Large systems normally contain many components. A small system may contain one component.

Deciding what to place under version control

In deciding what to place under version control, do not limit yourself to source code files and directories. The power of configuration management is that you can record a history of your project as it evolves so that you can re-create the project quickly and easily at any point in time. These include, but are not limited to the following:

- Source code files and directories
- Model files, such as Rational Rose files
- Libraries
- Executable files
- Interfaces
- Test scripts
- Project plans
- Compilers, other developer tools, and system header files
- System and user documentation
- Requirements documents

To record a full picture of the project, include all files and directories connected with it.

Mapping components to projects

After mapping your system architecture to a set of components and identifying the full set of files and directories to place under version control, you need to determine whether to use one project or multiple projects. In general, think of a project as the configuration management environment for a project team working on a specific release. Team members work together to develop, integrate, test, and release a set of related components. For many systems, all work can be done in one project. For some systems, work must be separated into multiple projects. In deciding how many projects to use, consider the following factors:

- Amount of integration required
- Whether you need to develop and release multiple versions of the product concurrently

Amount of integration

Determine the relationships between the various components. Related components that require a high degree of integration belong to the same project. By including related components in the same project, you can build and test them together frequently, thus avoiding the problems that can arise when you integrate components late in the development cycle.

Need for parallel releases

If you need to develop multiple versions of your system in parallel, consider using separate projects, one for each version. For example, your organization may need to work on a patch release and a new release at the same time. In this situation, both projects use mostly the same set of components. (Note that multiple projects can modify the same set of components.) When work on the patch release project is complete, you integrate it with the new release project.

If you anticipate that your team will develop and release numerous versions of your system over time, you may want to create a mainline project. A mainline project serves as a single point of integration for related projects over a period of time.

Figure 12 shows the initial set of components planned for the Transaction Builder system. A team of 30 developers work on the system. Because a high degree of integration between components is required, and most developers work on several components, the project manager included all components in one project. For information about using multiple UCM projects for your development, see "Project uses" on page 141 and "Using a mainline project" on page 142.

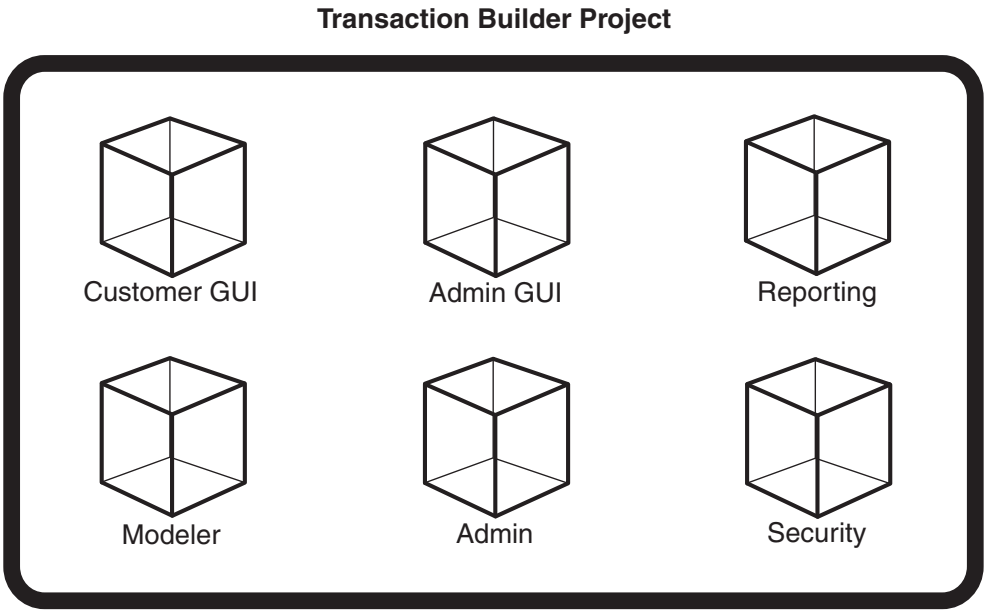


Figure 12. Components used by Transaction Builder project

Organizing components

After you map your system architecture to an initial set of components and determine which projects will access those components, refine your plan by performing the following tasks:

- Decide how many VOBs to use
- Identify any additional components
- Define the component directory structures
- Identify read-only components

Deciding how many VOBs to use

You can store multiple components in a VOB. If your project uses a small number of components, you may want to use one VOB per component. However, if your project uses many components, you may want to store multiple components in several VOBs. A VOB can store many versions of many elements. It is inefficient to use a VOB to store one small component.

Keep in mind the following restrictions:

- A component root directory must be at the level of or one level beneath the VOB root directory. A component includes all directory and file elements under its root directory. For example, in Figure 13, **Libs** cannot be a component.
- You cannot nest components. For example, in Figure 13, **GUI**, **Admin**, and **Reports** can be components only if **Dev** is not a component.

- If you make a component at the VOB root directory, that VOB can never contain more than that one component. For this reason, create components one level beneath the VOB root directory. Doing so allows you to add components to the VOB in the future.
- Whether you make a component at the level of or one level beneath the VOB root directory, the component name must be unique within its PVOB.

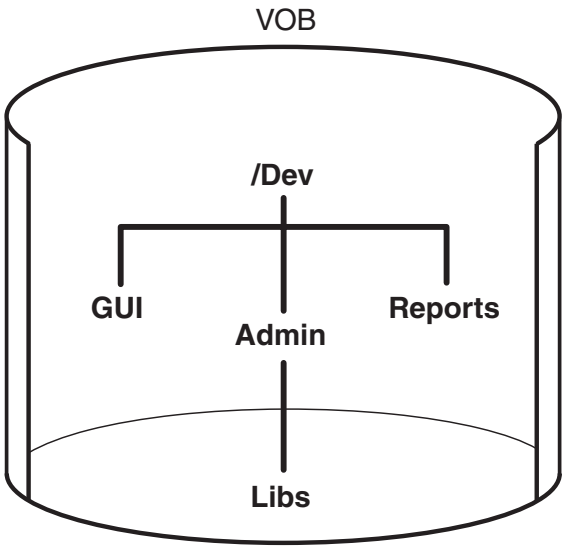


Figure 13. Storing multiple components in a VOB

Identifying additional components

Although you should be able to identify nearly all necessary components by examining your system architecture, you may overlook a few. For example:

System component
It is a good idea to designate one component for storing system-level files. These items include project plans, requirements documents, and system model files and other architecture documents.

Project baseline component
If you plan to use a composite baseline that selects baselines from all of the components of the project, store the composite baseline in its own component. See "Identifying a project baseline" on page 46 for details.

Testing component
Consider using a separate component for storing files related to testing the system. This component includes files such as test scripts, test results and logs, and test documentation.

Deployment component
At the end of a development cycle, you need a separate component to store the generated files that you plan to ship with the system or deploy inhouse. These files include executable files, libraries, interfaces, and user documentation.

Tools component
In addition to placing source files under version control, it is a good idea to place your team's developer tools, such as compilers, and system header files under version control.

Defining the directory structure

After you complete your list of components, you need to define the directory structures within those components. You can start with a directory structure similar to the one shown in Table 1; then modify the structure to suit your system needs.

In Table 1, Component_1 through Component_*n* refers to the components that map to the set of logical packages in your system architecture.

Table 1. Recommended directory structure for components

Component	Directories	Typical contents
System	plans	Project plans, mission statement, and so on
	requirements	Requirements documents
	models	Rose files, other architecture documents
	documentation	System documentation
Component_1 through Component_*n*	requirements	Component requirements
	models	Component model files
	source	Source files for this component
	interfaces	Component public interfaces
	binaries	Executable and other binary files for this component
	libraries	Libraries used by this component
	tests	Test scripts and related documents for this component
Test	scripts	Test scripts
	results	Test results and logs
	documentation	Test documentation
Deployment	binaries	Deployed executable files
	libraries	Deployed libraries
	interfaces	Deployed interfaces
	documentation	User documentation
Tools	compilers	Developer tools such as Rational WorkBench, Visual .NET and IBM Rose
	headers	System header files
Project baseline	none	Composite baseline that selects baselines from all components in the project

Identifying read-only components

When you create a project, you must indicate whether each component is modifiable in the context of that project. In most cases, you make them modifiable. However, in some cases you want to make a component read-only, which prevents project team members from changing its elements. Components can be used in multiple projects.

One project team may be responsible for maintaining a component, and another project team may use that component to build other components (see Figure 14).

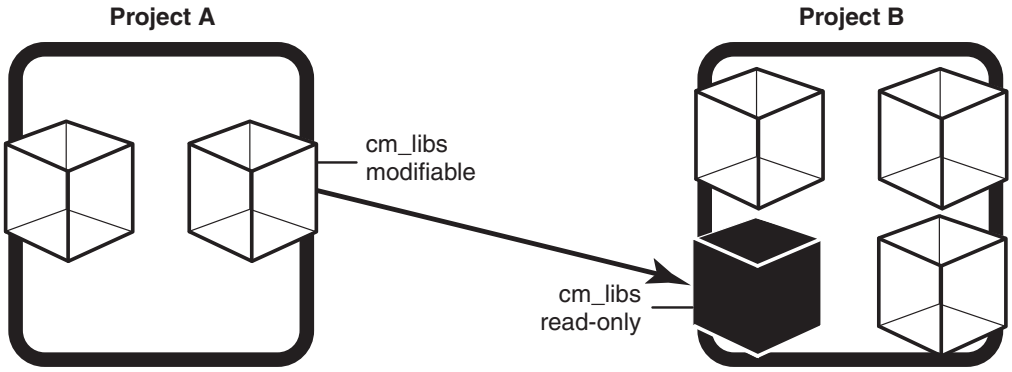


Figure 14. Using a read-only component

The **Project A** team members maintain a set of library files in the **cm_libs** component. **Project B** team members refer to some of those libraries when they build their components. In **Project A**, the **cm_libs** component is modifiable. In **Project B**, the same component is read-only. With respect to the **cm_libs** component, **Project A** and **Project B** have a producer-consumer relationship. For more information, see "Modifiable components" on page 63.

Because making a baseline of a component to change its members modifies the related component, the related component that is used for a composite baseline should be modifiable. A component without a VOB root directory (that is, one used to make a pure composite baseline) should be modifiable except in the following circumstances:

- The component is to hold only read-only components as members.
- No baseline is ever to be made in the component.

You cannot make a baseline of a read-only component without a VOB root directory.

Choosing a stream strategy

UCM provides many choices in using streams.

- A multiple-stream project with one shared work area and multiple private work areas
- A multiple-stream project with hierarchies of streams (that is, multiple shared work areas)
- A single-stream project
- A project with read-only streams

The basic multiple-stream project

The basic UCM process uses the multiple-stream project with the integration stream as the sole shared work area. Developers join the project by using the integration stream recommended baselines to populate their development streams; deliver completed work to the integration stream where the integrator incorporates the work into new baselines; and rebase their development streams to the new recommended baselines. Depending on the size of your project and the number of developers working on it, this process may be a good choice for your team.

Stream hierarchies

As an alternative to using the integration stream as the sole shared work area for the project, you can use the UCM development stream hierarchy feature to create multiple shared work areas. This approach supports a project organization that consists of small teams of developers where each team develops a specific feature in *feature-specific* development streams (see Figure 15).

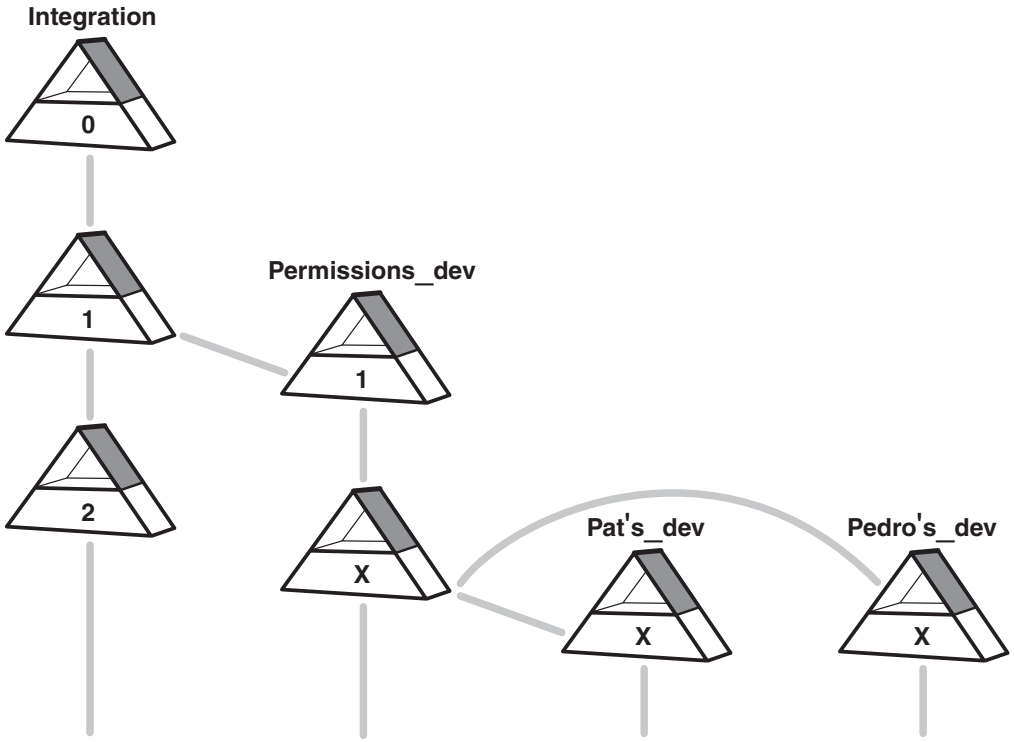


Figure 15. Using a feature-specific development stream

The project manager created a development stream called **Permissions_dev** for two developers who are working on a permissions feature. The developers, Pat and Pedro, joined the project at the **Permissions_dev** level rather than at the integration stream level. They deliver completed work to the **Permissions_dev** stream. Periodically, the integrator or lead developer responsible for managing the **Permissions_dev** stream incorporates the delivered work into new baselines, and the developers rebase their development streams to those new baselines.

When the two developers finish working on the permissions feature, they deliver their last work to the **Permissions_dev** stream. The integrator incorporates their delivered work into a final set of baselines and delivers those baselines to the integration stream.

Stream configurations and baseline contents

When project managers create projects, they add components to the project and select baselines for those components which are referred to as the foundation baselines. If you use a composite baseline for the project, the project has one baseline as its foundation baseline.

Optionally, a project manager can assign to a development stream a set of foundation baselines. Foundation baselines specify a stream's configuration by selecting the file and directory versions that are accessible in the stream.

The integration stream configuration

The integration stream is created with either baselines from another project or selected baselines from the PVOB. These foundation baselines are by default the recommended baselines of the integration stream. The recommended baselines of the project are the integration stream's recommended baselines.

In Figure 15, the set of foundation baselines chosen as the initial configuration of the integration stream are represented by baseline **0**. For an integration stream, all foundation baselines must be either baselines created in other projects' integration streams, or be imported or initial baselines. For an integration stream, you cannot use baselines created in development streams. This set of foundation baselines provides a stable, well-known configuration in the project integration stream.

Development stream configurations

When a development stream is created, you can assign it a set of foundation baselines. All foundation baselines for a development stream must be either recommended baselines in the parent stream or baselines created in the integration stream. If no baselines are recommended, baselines that were created in the integration stream of the project must be used. In Figure 15, the set of foundation baselines of the feature-specific development stream **Permissions_dev** is represented by baseline **1**, which were created in the parent integration stream.

When developers join a project, by default, their development streams are created with the set of recommended baselines in the parent stream. The set of foundation baselines of the development streams **Pat's_dev** and **Pedro's_dev** are represented by baseline **X**, a baseline that was created in the parent stream **Permissions_dev**.

These configuration rules attempt to establish a common foundation whereby there are no versions in child streams for which there is not an ancestor in the parent stream. This establishes a consistent ancestry for change flow through deliver and rebase operations.

A baseline does not have to be recommended for every component in the stream configuration.

Stream relationships

The relationships among streams determines how changes can move in a project (see Figure 16).

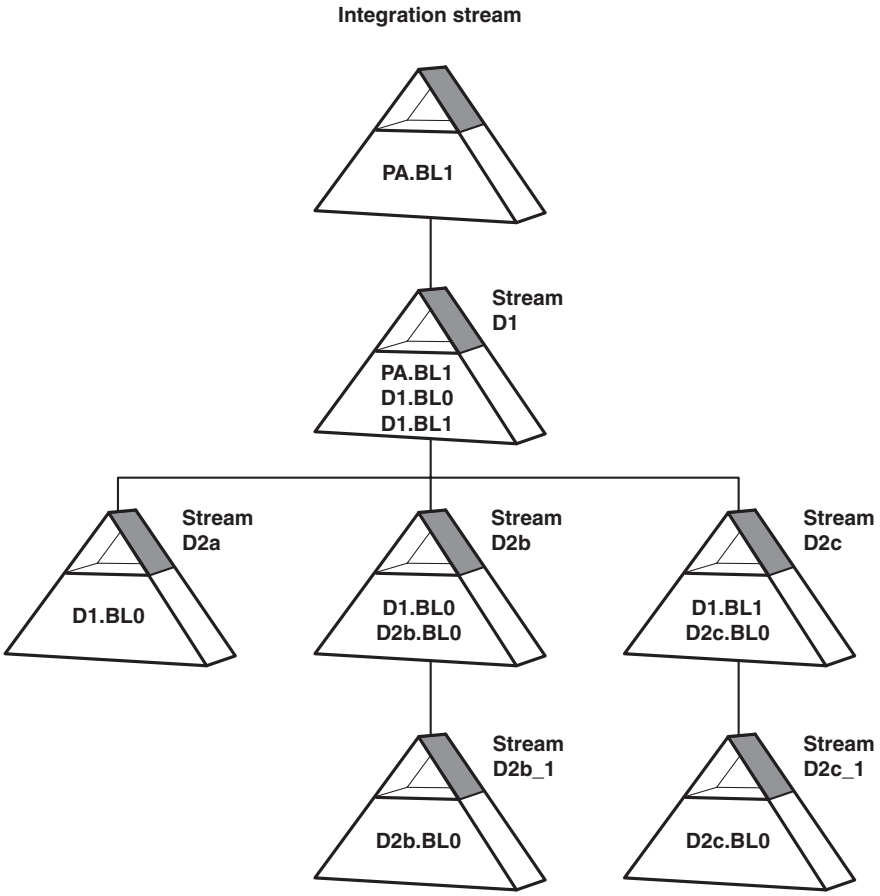


Figure 16. Stream relationships

An integration stream has foundation baselines represented by **PA.BL1** which are also its recommended baselines. Child stream **D1** has foundation baselines represented by **PA.BL1** and has recommended baselines **D1.bl0** and **D1.bl1**. Family terminology is used to describe where in the hierarchy a particular stream is located.

- Streams with the same parent are called *siblings*. For example, in Figure 16, streams **D2a**, **D2b**, and **D2c** are descendants from the same parent (stream **D1**). Streams **D2a** and **D2b** use the baselines in **D1.bl0** as their foundation baselines. Stream **D2c** uses the baselines in **D1.bl1** as its foundation baselines.

  Tip: Although stream **D2c** uses different foundation baselines than its siblings, all components in its siblings are also in its foundation baselines.

- The parent of a parent stream is called a *grandparent*. For example, in Figure 16, the integration stream is the grandparent of streams **D2a**, **D2b**, and **D2c** and the development stream **D1** is the grandparent of streams **D2b_1** and **D2c_1**.

  The foundation baselines in **D2b.bl0** of stream **D2b_1** are the recommended baselines in the parent stream **D2b**.

  The foundation baselines in **D2c.bl0** of stream **D2c_1** are the recommended baselines in the parent stream **D2c**.

- Streams whose parent streams are siblings are called *cousins*. For example, in Figure 16, streams **D2b_1** and **D2c_1** are cousins because their parent streams **D2b** and **D2c** are siblings.

  Although the cousin streams have different foundation baselines, the baselines are ancestors of the same foundation baselines in the grandparent stream **D1**.

Stream hierarchy and default targets

A project can have a hierarchy of development streams that starts with the integration stream (see Figure 17).

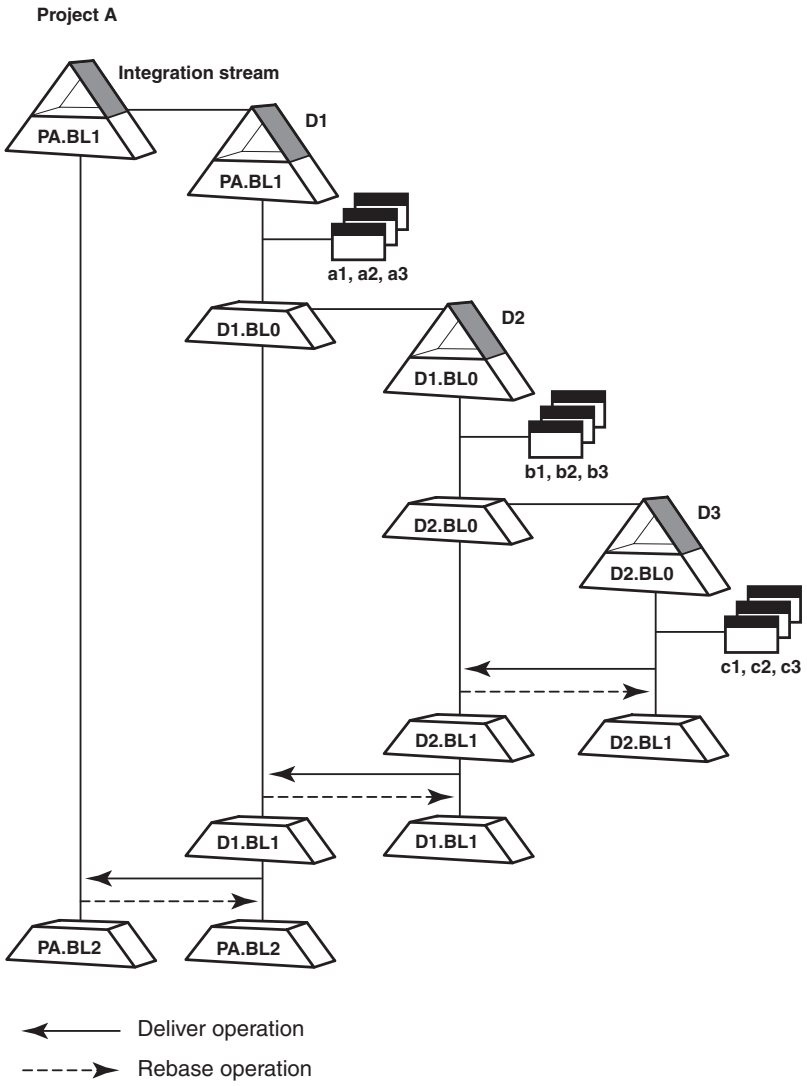


Figure 17. Stream hierarchy with multiple levels

A development stream is created as a child of either the integration stream or of another development stream. For example, stream **D1** is a child of the integration stream, stream **D2** is a child of stream **D1**, and **D3** is a child of stream **D2**.

The parent-child relationship between streams defines the default target of deliver operations and the default source of baselines for rebase operations. The default relationships are the following:

- A child stream delivers to its default target, the parent stream, any undelivered activities that it holds. For example:
  - Stream **D3** delivers activities **c1**, **c2**, and **c3** to its default target, stream **D2**.
  - Stream **D2** delivers to its default target, stream **D1**, activities **b1**, **b2**, and **b3** and the activities that have been delivered to it from its child streams.
  - Stream **D1** delivers to the integration stream activities **a1**, **a2**, and **a3** plus the activities that are delivered to it from its child streams.

- A child stream rebases to baselines in the parent stream to receive activities that were delivered by other development streams. Typically, these change sets are in recommended baselines. For example:
  - Stream **D3** rebases to recommended baselines in stream **D2**.
  - Stream **D2** rebases to recommended baselines in stream **D1**.
  - Stream **D1** rebases to recommended baselines in the integration stream.

Although the integration stream is a child of the project, it does not have a default relationship. If the project manager wants the integration stream to have a default relationship, an integration stream in another project can be specified as the default target of deliver operations and the source of recommended baselines to be used for rebase operations.

Alternate targets

All streams in the same project can deliver activities to streams other than the default target. Such alternate target streams are restricted by the foundation baselines in the source stream and the target stream.

A policy in the project controls whether streams can control access (see "Policies for deliver operations to nondefault targets" on page 66). Policies in the streams control stream access. A stream cannot be the target of a deliver operation if the project or stream policy prohibits access. One policy determines whether a stream can accept activities in a deliver operation from a stream in a different project.

Alternate targets in the same project

Within the same project, streams that share foundation baselines can share changes (see Figure 18).

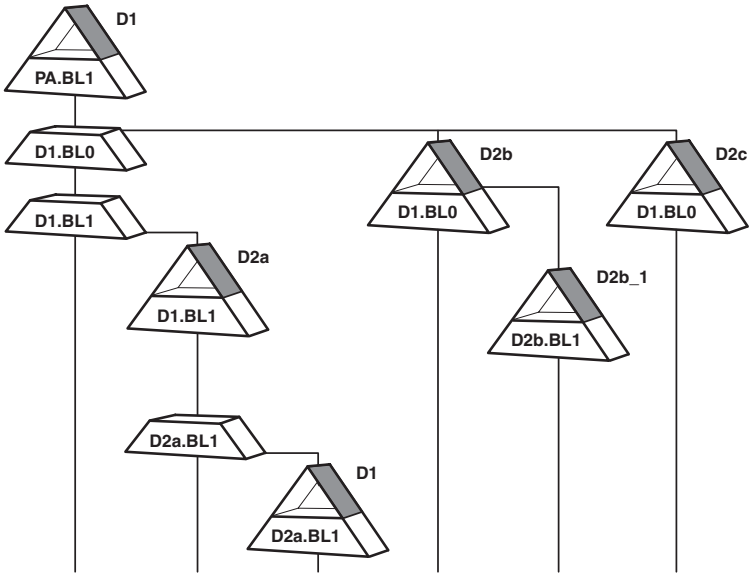


Figure 18. Direct stream relationships for alternate target deliver operations

Streams that share the same foundation baselines have a direct relationship and can share changes by using alternate target deliver operations. For example, child streams **D2b** and **D2c** share as their foundation baselines the recommended baselines in **D1.bl0** from parent stream **D1**. All the elements that are in the foundation baselines of streams **D2b** and **D2c** are in the foundation baselines of the child stream **D2a**.

Using an alternate target deliver operation, you can migrate activities to a sibling stream or to a stream related to a sibling stream rather than to a parent stream. Delivering activities to any stream other than the parent stream can also include other activities from the foundation baselines of the source stream. What changes are migrated depends on the relationship of the foundation baselines of the source stream relative to the target stream. For example, in a deliver operation from stream **D2b** to **D2c**, from stream **D2c** to **D2b**, or from stream **D2b_1** to **D2c**, because the foundation baselines are the same, only the changes in the source stream are migrated.

Some alternate target deliver operations can be *forward*. This involves delivering activities to a sibling stream that has an advanced baseline or has newer foundation baselines. For example, in Figure 18, migrating changes from stream **D2b** to **D2a** involves a forward deliver operation. Because all changes in foundation baselines of stream **D2b** are in those of **D2a**, only the activities that are in stream **D2b** are delivered.

However, alternate target deliver operations typically migrate more changes than you at first might anticipate. The alternate target deliver operation can also be *backward*. This involves delivering activities to a sibling stream that has not rebased to the newer, recommended baselines. For example, in Figure 18, migrating changes from stream **D2a** to **D2b** involves a backward deliver operation. Because the target stream **D2b** does not have all the changes that the source stream **D2a** has, the operation additionally migrates to stream **D2b** activities from baseline **D2a.bl1** in stream **D2a**.

Some alternate target deliver operations involve indirect baseline relationships (see Figure 19).

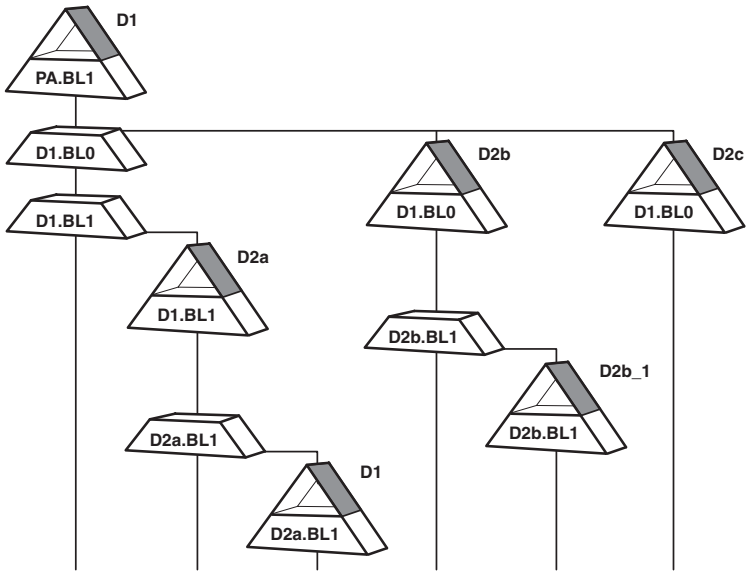


Figure 19. Indirect stream relationships for alternate target deliver operations

A stream can migrate its changes to a stream that has an indirect relationship. The deliver operation often includes activities from the foundation baselines of the source stream. The following are examples.

- From the child of one stream to the child of a sibling stream (cousin to cousin).

  For example, a deliver operation from stream **D2b_1** to **D2a_1** includes activities contained in baseline **D2b.bl1**.

- From a grandchild to a grandparent stream or to another direct ancestor.

  For example, a deliver operation from stream **D2a_1** to **D1** includes activities contained in baseline **D2a.bl1**.

  A deliver operation from stream **D2b_1** to stream **D1** includes activities contained in baseline **D2b.bl1**.

Coordinating development streams in the same project

The developers in a project can use an alternate-target deliver operation between sibling streams in the same project to coordinate work between them (see Figure 20).

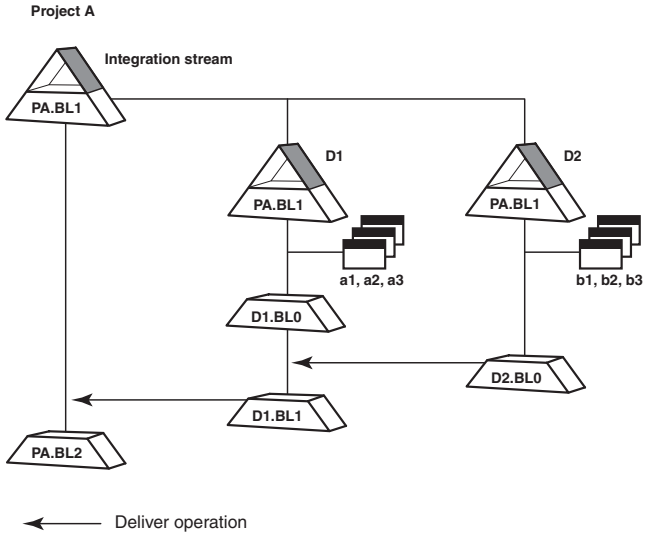


Figure 20. Alternate target intra-project deliver operation

Two developers in project **A** work on different portions of a feature in their own streams **D1** and **D2**, which are siblings. The two developers need to integrate their work before it is delivered to the integration stream. No other interaction is needed.

The developer using stream **D2** uses an alternate target deliver operation to migrate the changes in activities **b1**, **b2**, and **b3** to the sibling stream **D1**. The developer in stream **D1** tests the work and delivers all the activities to the default deliver target, the project **A** integration stream. This configuration works well where a low level of isolation is necessary.

Sharing changes by a rebase operation

Sharing by a rebase operation allows a stream to configure work done in a sibling stream (Figure 21).

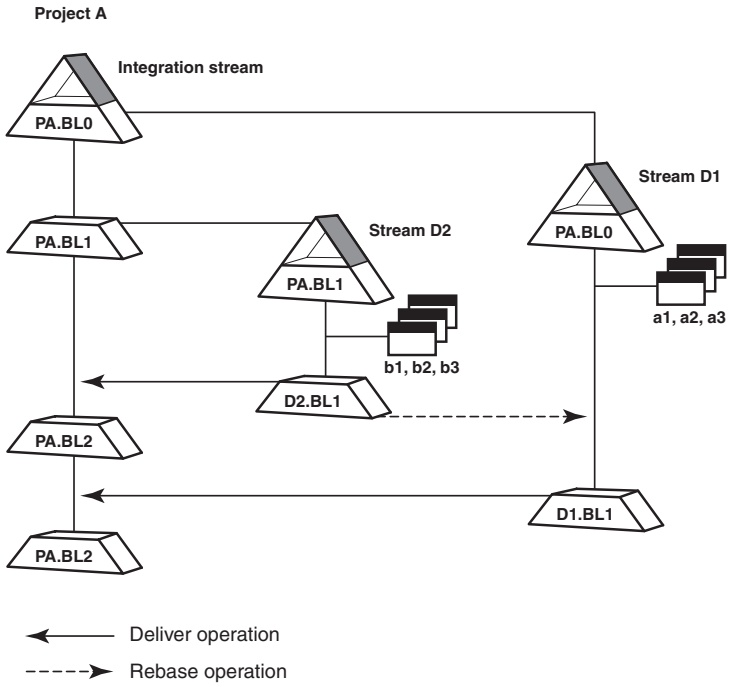


Figure 21. Sharing changes by a rebase operation

Typically, stream **D1** cannot rebase to baselines from stream **D2** because they do not meet the requirement that the baselines must be contained in the parent stream. If you deliver to the parent stream the activities in a baseline in one stream, a sibling stream can be configured with those changes. In Figure 21, if the baseline **D2.BL1** is first delivered to the parent stream (the project integration stream in this example), you can rebase stream **D1** to the baselines in **D2.BL1** in the sibling stream **D2**. Delivering to the parent stream ensures that, in a default target deliver operation, only activities that originated in stream **D1** are delivered in baselines **D1.bl1** to the parent stream.

Sharing by a rebase operation in this manner requires integration one level higher than the stream in which the changes originated.

Sharing changes by a deliver operation

Sharing can be done with deliver operations between siblings in the same project (see Figure 22).

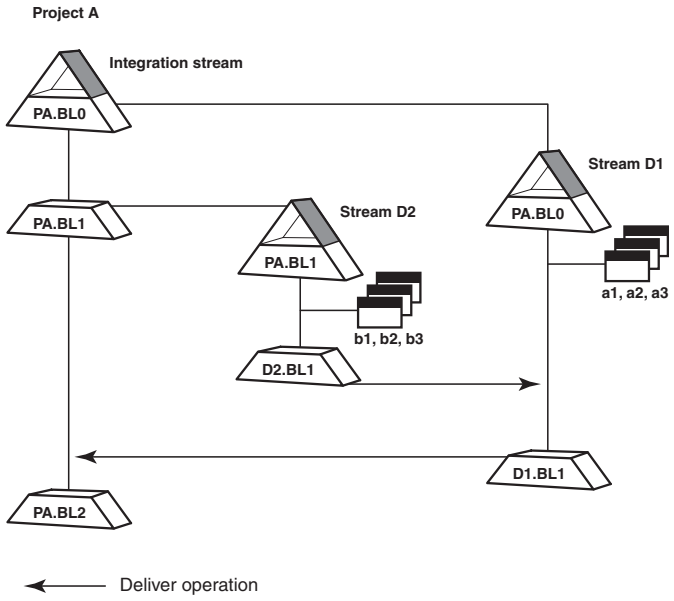


Figure 22. Sharing changes by an alternate target deliver operation

If you integrate changes in a sibling stream (rather than in a parent stream; see "Sharing changes by a rebase operation" on page 41), you can use an alternate target deliver operation to share changes between sibling streams. For example, if you deliver activities **b1**, **b2**, and **b3** in stream **D2** to stream **D1**, you must also deliver activities in baselines in **PA.BL1** and merge the changes with the versions in activities **a1**, **a2**, and **a3**. This complex deliver operation can be simplified by using a rebase operation before the alternate target deliver operation (see "Simplify a deliver operation with a rebase operation" on page 43).

All of the integrated changes are migrated to the parent stream (the project integration stream) when you deliver your work from stream **D1**.

Simplify a deliver operation with a rebase operation

A rebase operation can simplify an alternate-target deliver operation when a sibling stream needs to configure changes from a related stream (see Figure 23).

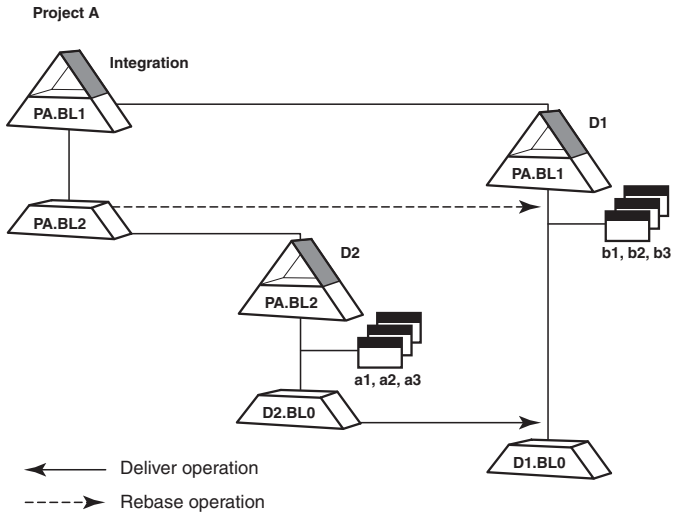


Figure 23. Rebase operation and alternate target deliver operation

Because streams **D1** and **D2** are siblings, stream **D2** can deliver its activities to stream **D1**. However, stream **D2** contains changes in it foundation baselines that are not in stream **D1**. If stream **D2** were to deliver activities **a1**, **a2**, and **a3** to stream **D1**, all the additional activities in baselines in **PA.BL2** would have to be delivered also.

However, if you rebase stream **D1** to the baselines in **PA.BL2** in the common parent stream before delivering the activities in stream **D2**, this complex alternate target deliver operation can be simplified. During the rebase operation, the changes in activities **b1**, **b2**, and **b3** are preserved. After the rebase operation, no activities other than **a1**, **a2**, and **a3** have to be delivered from stream **D2** to **D1**.

Single-stream projects

For most customers, a parallel development environment consisting of private and shared work areas makes sense. However, small teams of developers working together closely may prefer a *serial* development environment. UCM supports this by letting you create a single-stream project. A single-stream project contains one stream, the integration stream, and does not allow users to create development streams. When developers join a single-stream project, they create a view attached to the integration stream.

You may want to use a single-stream project during the initial stage of development when several developers want to share code quickly. When the development effort expands and you need a parallel development environment, you can create a multiple-stream project based on the final baselines in the single-stream project.

The following are the main advantages of single-stream projects:

- Developers who work in dynamic views see each other's work as soon as they check in their files. Developers who work in snapshot views see each other's work as soon as they check in their files and update their views. In a multiple-stream project, developers see each other's changes only during deliver and rebase operations.
- Developers have a simplified work environment. Because all work is done on the integration stream, developers do not need to maintain a development

stream and two views, one attached to the development stream and one attached to the integration stream. In addition, developers do not need to perform deliver or rebase operations.

- Your role as integrator is simplified. Because developers work on the same stream and see each other's changes immediately, you do not need to create baselines frequently. In contrast to a multiple-stream project, developers do not depend on baselines to integrate their work. The primary purpose of baselines in a single-stream project is to identify major milestones.

The following are the main disadvantages of single-stream projects:

- Developers have limited support for sharing files simultaneously. Although multiple developers can check out an element in the same stream at the same time, only one developer can reserve the checkout. A *reserved checkout* guarantees the developer's right to check in a new version of the element. All other developers must check out the element as *unreserved*, which means that they cannot check in their versions until after the reserved checkout has been checked in or canceled. Developers with unreserved checkouts must merge their changes with the changes made by the reserved checkout.
- Because changes are shared as soon as developers check in their files, developers assume the full responsibility for testing their work and must be extremely vigilant to ensure that they do not introduce bugs to the project. In contrast, a multiple-stream project allows the integrator or a software quality engineering team to perform extensive testing of new baselines on a dedicated testing stream and to recommend baselines only after they pass those tests.
- Because changes are shared as soon as developers check in their files, developers might keep files checked out longer than they would in a multiple-stream project. If a view is lost, all changes made but not checked in that view are also lost. Therefore, have your Rational ClearCase administrator frequently back up views for single-stream projects.

Read-only streams

During the evolution of a project, you might need to provide some users with access to baselines while ensuring that they do not make any changes to components. You can address this requirement by creating a Read-Only development stream for those users. You cannot make baselines in Read-Only streams, nor can you create child streams beneath them. You can create *view-private* files, such as derived objects in Read-Only streams.

Common use cases for Read-Only streams include the following:

- Your quality engineering team needs to build and test a particular configuration.
- Your customer support team needs access to a library that was built in a previous release.
- Your release engineering team needs to create a release based on a combination of old and new baselines.

Specifying a baseline strategy

After you organize the project's components, determine your strategy for creating baselines of those components. The baseline strategy must define the following aspects of projects:

- A project baseline (see "Identifying a project baseline" on page 46)
- The use of pure composite baselines (see "Pure composite baselines" on page 47)
- When to create baselines (see "When to create baselines" on page 51)

- How to name baselines (see "Defining a baseline naming convention" on page 52)
- The set of promotion levels (see "Identifying promotion levels to reflect state of development" on page 52)
- How to test baselines (see "Planning how to test baselines" on page 52)

Identifying a project baseline

In your role as project integrator, you are responsible for telling developers which baselines to use when they join the project and when they rebase their development streams. You could keep track of a list of baselines, one for each component. However, a more efficient practice is to use a composite baseline to represent the project baseline (see Figure 24).

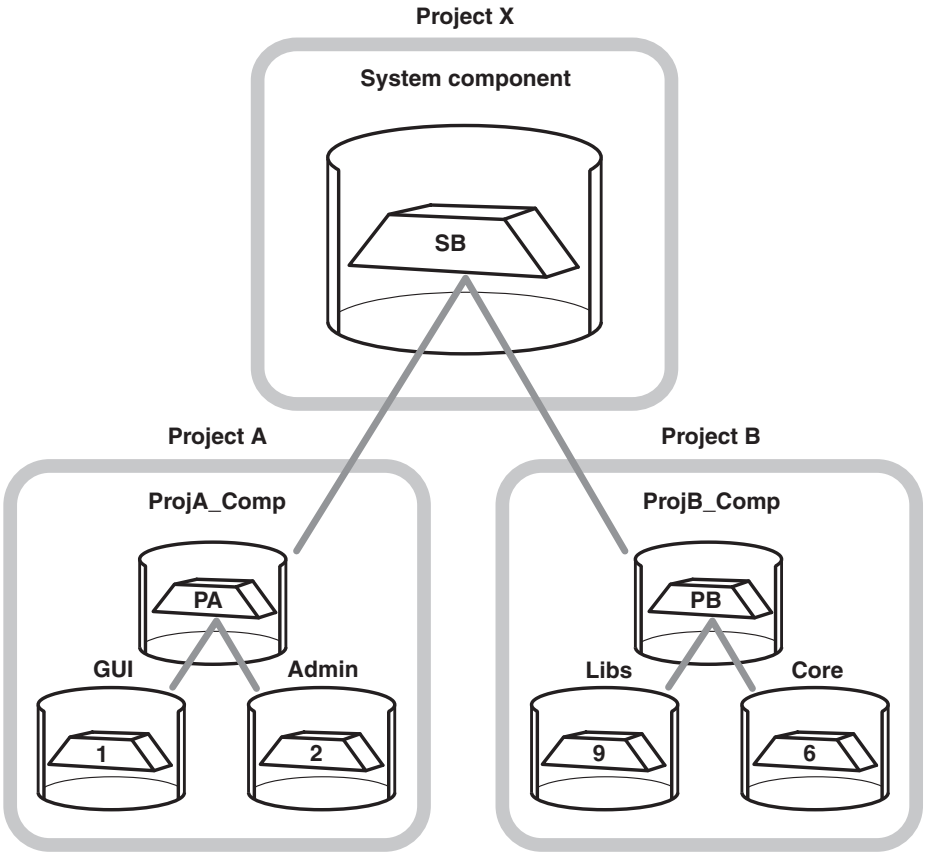


Figure 24. Using a system-level composite baseline

Project A uses a composite baseline, **PA**, to select baselines in the **GUI** and **Admin** components. **Project B** also uses a composite baseline, **PB**. Baselines that are selected by a composite baseline are referred to as *members*. A composite baseline is said to *depend* on the member baseline.

After you create a composite baseline to represent the project baseline, the next time you invoke the make baseline operation on the component that contains the project baseline, UCM performs the operation recursively. If a component that contributes to the composite baseline has changed since its latest baseline, UCM creates a new baseline in that component. For example, assume that developers made changes to files in the **GUI** component after the integrator created the **1** baseline. The next time you make a new project baseline, UCM creates a new

baseline in the **GUI** component that incorporates the changed files, and the new project baseline selects the new GUI baseline.

A composite baseline can select other composite baselines. Thus, member baselines may themselves be composite baselines. The chain of dependencies forms an acyclic directed graph, with no limit on the depth of the membership. Composite baselines do not necessarily have to model any source or build dependencies.

For example, if your system is so large that it consists of multiple projects, you may want to use a composite baseline to represent the system baseline. In Figure 24, **SB** is a composite baseline that selects the **PA** and **PB** baselines of **Project A** and **Project B**, respectively.

In addition to using a composite baseline to represent the project, you can use multiple composite baselines within the same project. When working with multiple composite baselines, you can encounter situations where two composite baselines select different baselines of the same component. When this happens, you need to resolve the conflict by choosing one of the member baselines. To avoid these conflicts, choose a simple baseline design, rather than one that uses a complex hierarchy of composite baselines. For information about baseline conflicts, see "Resolving baseline conflicts" on page 120.

Pure composite baselines

Like all baselines, a composite baseline must belong to a component. However, that component does not need to contain any of its own elements. For example, in Figure 24, the **System component**, **ProjA_Comp**, and **ProjB_Comp** components consist only of their composite baselines. When you create a component to be used solely for housing a composite baseline, you can specify an option that directs UCM to make the component without creating a root directory in a VOB. Such a component can never contain its own elements and its baseline is referred to as a *pure* composite baseline.

Dependency relationships in pure composite baselines

A pure composite baseline shows a loosely coupled dependency relationship between components (see Figure 25).

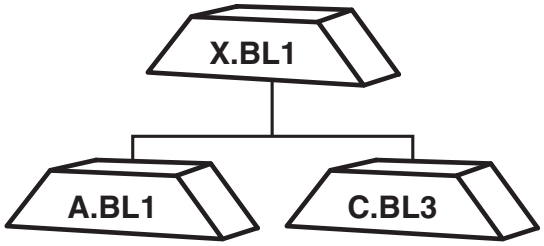


Figure 25. Loosely coupled relationship between baselines

The objects **A.BL1** and **C.BL1** are baselines on components **A** and **C** that are used to group directory and file elements. Baselines **A.BL1** and **C.BL1** are members of the composite baseline **X.BL1** on the component **X** that was created without a root directory in a VOB. Baseline **X.BL1** directly depends on baselines **A.BL1** and **C.BL1**. However, the dependency between **A.BL1** and **C.BL1** is incompletely expressed. The components have to be used together, but it is not clear whether component **A** depends on **C** or component **C** depends on **A**.

You cannot change the existing dependency relationships in a composite baseline. To change dependency relationships, you must create a new baseline, a descendant of the changing baseline. For the composite baseline, you can add dependency references to new components (to create a new dependency relationship) or can drop dependency references to existing components (to discontinue a dependency relationship). For either operation, a new composite baseline is created.

At any time, a composite baseline can have member baselines added or dropped. Although it appears that components are manipulated to create composite baselines, the dependency relationship is made between baselines and not components. One might say that some components are dependant on other components, but the dependency relationships have a limited scope. The dependency relationship may last for the life of a particular project, but it might be different for other projects. The dependency relationships change over time as components are added or dropped. So the relationship must be made between versions of components, the baselines, rather than between the components themselves.

Because a component without a VOB root directory has no elements (and, therefore, no associated code), a new composite baseline created on such a component can only indicate changes in the dependency relationship. A composite baseline on a such a component is an aggregation of baselines.

When you use a pure composite baseline, a new baseline on component **X** only means that there was some change in the membership of the baseline. Using a pure composite baseline provides greater configuration flexibility than using a composite baseline on a component that has a root directory in a VOB. Use pure composite baselines whenever possible when you configure composite baselines.

Dependency relationships in composite baselines of ordinary components

In a composite baseline of a component that has a root directory in a VOB (an *ordinary* component), a tightly coupled relationship exists between components as shown in Figure 26.

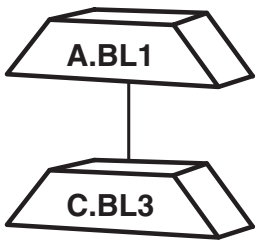


Figure 26. Tightly coupled relationship between baselines

A.BL1 is a baseline on component **A** that groups directory and file elements and is also a composite baseline that selects baseline **C.BL3** of component **C**. Baseline **C.BL3** is a member of composite baseline **A.BL1**. Composite baseline **A.BL1** depends on the member baseline **C.BL3**.

A change to the baseline in component **A** could be caused by a change in the configuration of component **A** (that is, a member being added or dropped) or by a new baseline in component **C**. This type of component arrangement is tightly coupled, for example, if code in component **A** depends on code in component **C**.

In the tightly coupled relationship, the composite baseline **A.BL1** fulfills two roles: selecting baselines from other components and identifying a set of versions in its component. Determining the reason for making a new descendant of this type of composite baseline is expensive (in terms of performance). Using a composite baseline that has tightly coupled relationships imposes configuration restrictions. For example, you may only be able to do an advanced rebase operation.

Making a new descendant baseline

Making a new descendant for a pure composite baseline indicates that the relationship between member baselines has changed. The loosely coupled relationship in a pure composite baseline (see Figure 25) indicates that the project integrator manages the dependency among the components outside of the Rational ClearCase environment. This form of dependence provides more choices of baselines to use in a project. Because the dependency relationship does not enforce the changing of component **A** if component **C** changes, the project integrator must be more careful in selecting baselines but has more flexibility in making changes.

Making a new descendant for a composite baseline of an ordinary component when the predecessor baseline changes can indicate one or both of the following meanings:

- Changes in the baseline dependency relationships
- Changes in the elements that make up the components

In a tightly coupled dependence (see Figure 26), a change in component **C** enforces changing component **A**.

Whether to use pure composite baselines

Many projects or project organizations need the flexibility provided by using pure composite baselines. Pure composite baselines are better in the following situations:

- If baseline conflicts occur.
- The project needs a lateral rebase operation to configure baselines from outside the project.

However, a project that does not include overlapping composite baselines will not have conflicts. Also, in development teams where projects are release-oriented, a project is likely to only need baselines from the previous project, and so the rebase flexibility is not needed.

To avoid using pure composite baselines, a project integrator can ensure that the subsystems stay synchronized with the shared components. When a new baseline is created on a shared component, all consuming projects would release a new baseline for their subsystem based on the new baseline. This frequent updating and rebasing prevent baseline conflicts. Such overhead is likely to be feasible only for projects with a small number of shared components.

Changing to a pure composite baseline

If a project has a composite baseline on an ordinary component, you can change the project to use a pure composite baseline (see Figure 27).

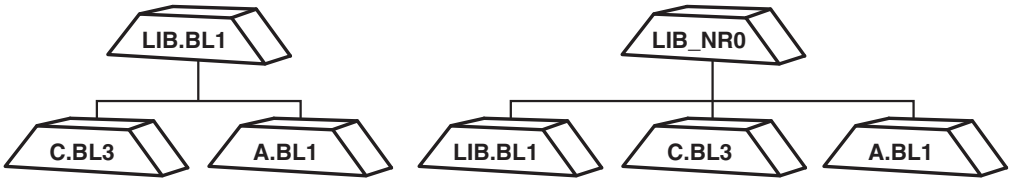


Figure 27. Changing a regular composite to a pure composite baseline

The project manager can create a component without a VOB root directory for each component that has a composite baseline and contains elements. For example, composite baseline **LIB.BL1** is for a component that has its own elements. The project manager drops the dependency references to components C and A from baseline **LIB.BL1**. The project manager creates a new component **LIB_NR** that does not have a root directory. Then a new composite baseline **LIB_NR0** can be made based on the rootless component. And you can add dependencies on the baseline **LIB.BL1** of the original component and on baselines **C.BL3** and **A.BL1** of the other components.

Creation of composite baseline descendants

If you introduce changes in a member component in a composite baseline, related baselines must be updated. The following are reasons to create a new baseline:

- Changes in the component since the last baseline was created
- Changes to the dependencies (a member is added or dropped)
- Replacement of member baselines by different baselines on the component

A change in one of the dependencies is propagated up to the root as new baselines are created to include the new member baselines. For example, there are changes in component **C**, and a new baseline is created on component **A** (see Figure 28).

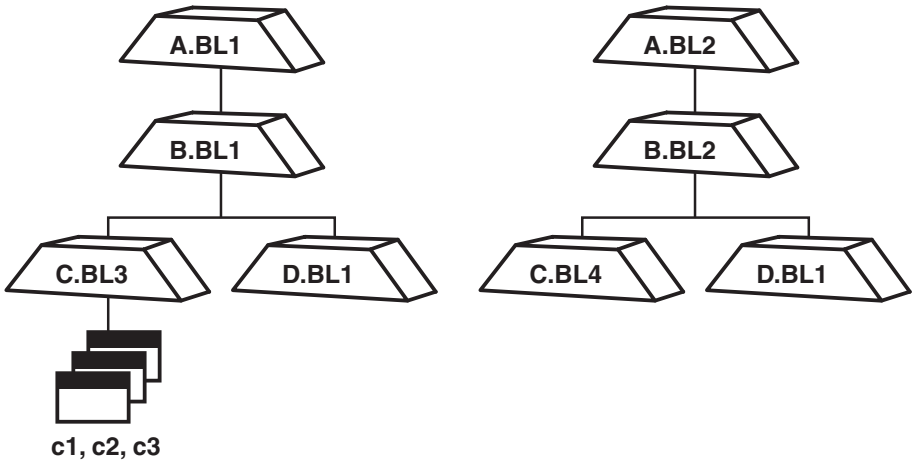


Figure 28. Creation of a composite baseline descendant

When you create a new baseline on component **A**, the work done on component **C** causes a new baseline, **C.BL4**, to be created to capture the changes. Because baseline **C.BL3** was replaced by **C.BL4**, and baseline **C.BL3** was a member of composite baseline **B.BL1**, a new baseline, **B.BL2**, on component **B** is made to record the relationship with baseline **C.BL4**. Because baseline **B.BL1** was replaced by baseline **B.BL2**, a new baseline **A.BL2** on component **A** is needed.

Unless you explicitly add dependencies to a composite baseline or drop dependencies from a composite baseline, when a baseline is created following on a composite baseline, the new baseline inherits the members of its predecessors. When a new descendant baseline is created, all dependencies of the composite baselines are checked and new baselines are created as needed.

A new baseline need only be created for the composite baseline, not for any member baselines of unchanged components. Unless a new baseline is needed for a project that is not using the composite baseline, you should not have to create a new baseline for component **D**. Baseline **B.BL2** inherits baseline **D.BL1** unchanged. A new baseline for component **D** would not be a member of the composite baseline **B.BL2**.

When to create baselines

At the beginning of a project, you must identify the baseline or baselines that represent the starting point for new development. As work on the project progresses, you need to create new baselines periodically.

Identifying the initial baseline

If your project represents a new version of an existing project, you probably want to start work from the latest recommended baselines of the existing components of the project. For example, if you are starting work on version 3.2 of the Transaction Builder project, identify the baselines that represent the released, or production, versions of its version 3.1 components. A convenient way to start a project with stable versions of components is to use a bootstrap project (see "Bootstrap projects" on page 146).

If you use pure composite baselines, create a bootstrap project with the initial baselines. Then, create your ongoing projects and configure them with the pure composite baselines from that bootstrap project.

If you are converting a base ClearCase configuration to a project, you can make baselines from existing labeled versions. Check whether the latest stable versions are labeled. If they are not, you need to create and apply the label type to the versions that you plan to include in your project. See "Making a baseline from a label" on page 98 for information about creating and applying a label type to versions.

Ongoing baselines

After developers start working on their streams in the new project and make changes, create baselines on the integration stream and on any feature-specific development streams on a frequent (nightly or weekly) basis. This practice has several benefits:

- Developers stay in sync with each other's work.

  It is critical to good configuration management that developers have private work areas where they can work on a set of files in isolation. Yet extended periods of isolation can cause problems. Developers are unaware of each other's work until you incorporate delivered changes into a new baseline, and they rebase their development streams.
- The amount of time required to merge versions is minimized.

  When developers rebase their development streams, they may need to resolve merge conflicts between files that the new baseline selects and the work in their private work areas. When you create baselines frequently, they contain fewer changes, and developers spend less time merging versions.

- Integration problems are identified early.

  When you create a baseline, you first build and test the project by incorporating the work delivered since the last baseline. By creating baselines frequently, you have more opportunities to discover any serious problems that a developer may introduce to the project inadvertently. By identifying a serious problem early, you can localize it and minimize the amount of work required to fix the problem.

If you are working in a single-stream project, you do not need to create baselines frequently. Developers see each other's changes as soon as they check in files; they do not rebase to the latest recommended baselines. The primary purpose of baselines in a single-stream project is to identify major project milestones, such as the end of an iteration or a beta release.

Defining a baseline naming convention

Because baselines are an important tool for managing a project, define a meaningful convention for naming them. You may want to include some or all of the following information in a baseline name:

- Project name
- Milestone or phase of development schedule
- Date created

For example: **V4.0TRANS_BL2_June12.**

UCM includes a set of templates that you can use to implement a baseline naming convention within a project. See "Setting a baseline naming template" on page 92 for details.

Identifying promotion levels to reflect state of development

A promotion level is an attribute of a baseline that you can use to indicate the quality or stability of the baseline. The following default promotion levels are provided:

- Rejected
- Initial
- Built
- Tested
- Released

You can use some or all of the default promotion levels, and you can define your own. The levels are ordered to reflect a progression from lowest to highest quality. You can use promotion levels to help you recommend baselines to developers. The Recommended Baselines window displays baselines that have a promotion level equal to or higher than the one you specify. You can use this feature to filter the list of baselines displayed in the window. Determine the set of promotion levels for your project and the criteria for setting each level.

Planning how to test baselines

Typically, software development teams perform several levels of testing. An initial test, known as a validation test, checks to see that the software builds without errors and appears to work as it should. A more comprehensive type of testing, such as regression testing, takes much longer and is usually performed by a team of software quality engineers.

When you make a new baseline, you need to lock the integration stream to prevent developers from delivering additional changes. This allows you to build and test a static set of files. Because validation tests are not exhaustive, you probably do not need to lock the integration stream for a long time. However, more extensive testing requires substantially more time.

Keeping the integration stream locked for a long time is not a good practice because it prevents developers from delivering completed work. One solution to this problem is to create a development stream to be used solely for extensive testing. After you create a new baseline that passes a validation test, your testing team can rebase the designated testing development stream to the new baseline. When the baseline passes the next level of testing, promote it. When you are confident that the baseline is stable, make it the recommended baseline so that developers can rebase their development streams to it.

For information on creating a testing development stream, see "Creating a development stream for testing baselines" on page 106. For information on testing baselines, see "Testing the baseline" on page 116.

Planning PVOBs

UCM objects such as projects, streams, activities, and change sets are stored in project VOBs (PVOBs). PVOBs can also function as administrative VOBs. You need to decide how many PVOBs to use for your system and whether to take advantage of the administrative capabilities of the PVOB.

Deciding how many PVOBs to use

Product Note: This section does not apply to Rational ClearCase LT because that product allows for only one PVOB per server.

Projects that use the same PVOB have access to the same set of components. If developers on different projects need to work on some of the same components, use one PVOB for those projects. For example, Figure 29 shows concurrent development of two versions of the **Webotrans** product. While most members of the team work on the 4.0 release in one project, a small group works on the 4.0.1 release in a separate project. Both projects use the same components, so they use one PVOB.

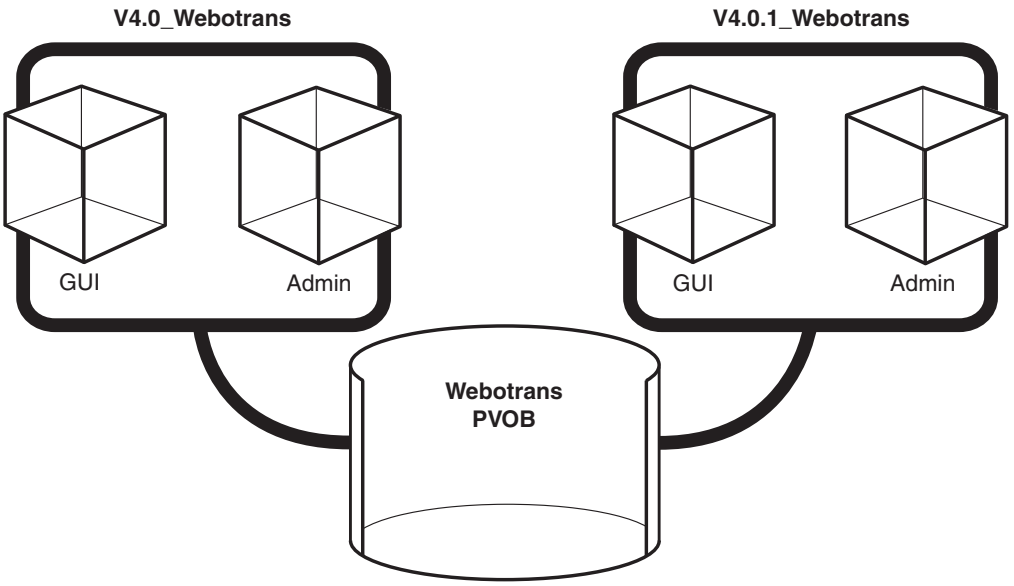


Figure 29. Related projects sharing one PVOB

Consider using multiple PVOBs only if your projects are so large that PVOB capacity becomes an issue. For more information, see "Using multiple PVOBs."

Understanding the role of the administrative VOB

An *administrative VOB* stores global type definitions. VOBs that are joined to the administrative VOB with **AdminVOB** hyperlinks share the same type definitions without having to define them in each VOB. For example, you can define element types, attribute types, hyperlink types, and so on in an administrative VOB. Any VOB linked to that administrative VOB can then use those type definitions to make elements, attributes, and hyperlinks.

If you currently use an administrative VOB, you can associate it with your PVOB by creating an **AdminVOB** hyperlink between the PVOB and the administrative VOB. On Windows computers, the VOB Creation Wizard creates the **AdminVOB** hyperlink for you. On UNIX® workstations, use the **cleartool mkhlink** command to create the **AdminVOB** hyperlink. Thereafter, when you create components, **AdminVOB** hyperlinks are created between the VOBs that store the components' root directories and the administrative VOB. These hyperlinks enable the components to use the administrative VOB's global type definitions.

If you do not currently use an administrative VOB, do not create one. When you create components, **AdminVOB** hyperlinks are made between the VOBs that store the component root directories and the PVOB, and the PVOB assumes the role of administrative VOB.

For details on administrative VOBs and global types, see the *IBM Rational ClearCase Administrator's Guide*.

Using multiple PVOBs

Although you can use only one PVOB for all your projects, your organization might have multiple PVOBs. In planning this configuration with multiple PVOBs, consider the following factors:

- "Multiple PVOBs and a common administrative VOB" on page 55

• "Multiple PVOBs and feature levels" on page 56

Multiple PVOBs and a common administrative VOB

If projects in one PVOB need to modify components in other PVOBs, your Rational ClearCase administrator needs to identify one PVOB to serve as a common administrative VOB for the PVOBs and the component VOBs. In Figure 30, **PVOB1** and **PVOB2** use **PVOB3** as their administrative VOB.

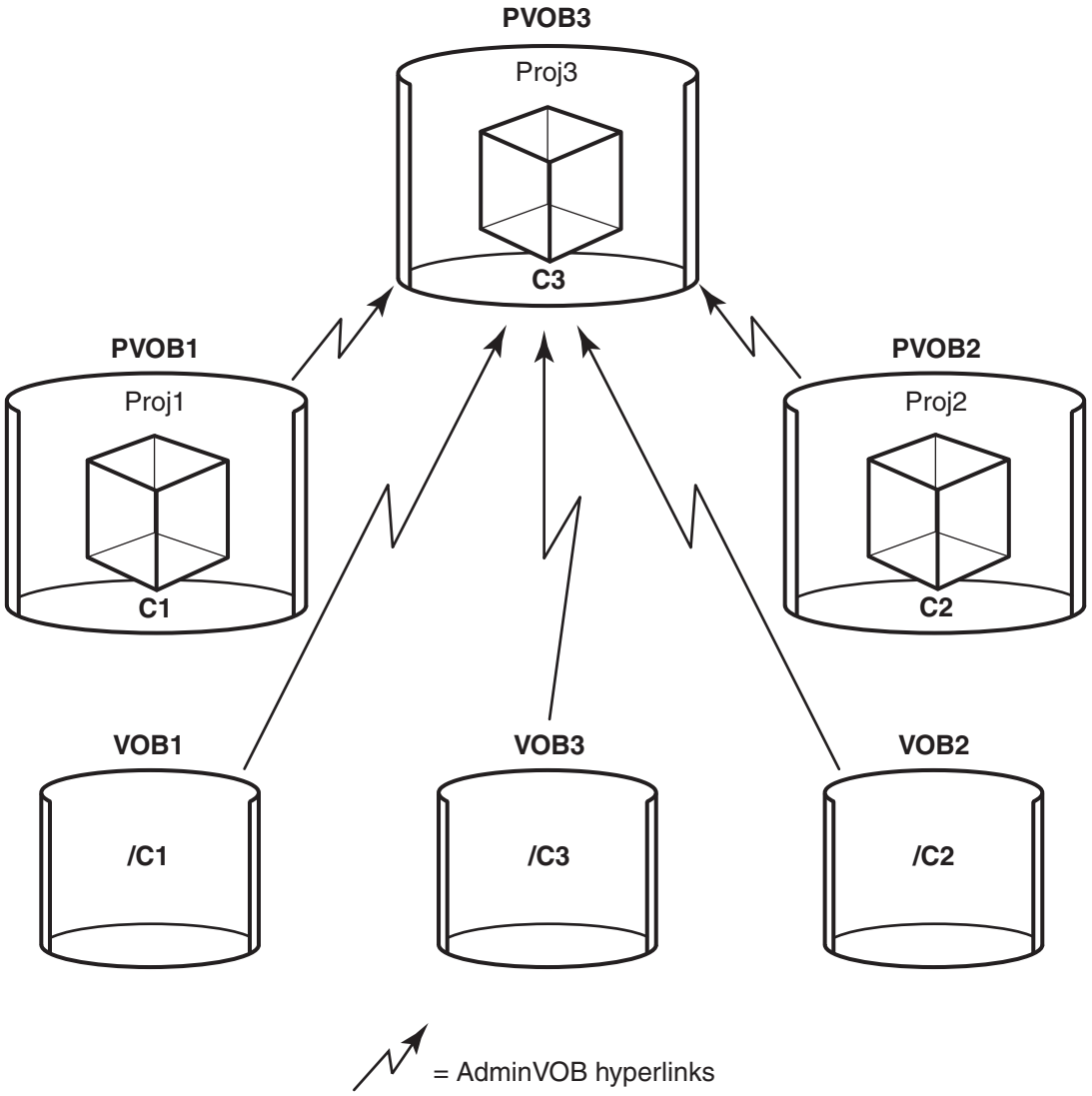


Figure 30. Using one PVOB as an administrative VOB for multiple PVOBs

The arrows from the PVOBs and the component VOBs represent **AdminVOB** hyperlinks to **PVOB3**. Because the component VOBs and the PVOBs share a common administrative VOB, all three projects can modify all three components.

For PVOBs that do not share a common administrative VOB, a project may select a component from another PVOB but the component will be Read-Only within that project.

In Figure 30, a PVOB serves as an administrative VOB.

As an alternative to using a PVOB as an administrative VOB, you can link PVOBs and component VOBs to an administrative VOB. This approach might be appropriate if your development team is moving from base ClearCase to UCM and you currently use an administrative VOB.

If you plan to use multiple PVOBs, create the PVOB that will serve as the administrative VOB first. When you create the other PVOBs, specify the first PVOB as the administrative VOB.

Multiple PVOBs and feature levels

For environments that implement multiple PVOBs, the first PVOB must be at feature level 3 or higher. Adhering to this rule avoids a situation that UCM software cannot detect and that could cause unknown problems. For example, if one of your PVOBs is at feature level 2 and you configure a new stream in a PVOB that is at feature level 2, you might have unknown problems. If the new stream is configured with a baseline from a PVOB that is at feature level 3 or later, the new stream will have configuration rules that can cause errors.

For instructions about raising the feature level of a VOB that is not replicated, see *IBM Rational ClearCase Administrator's Guide* and the **chflevel** reference page. For instructions about raising the feature level of replicated VOBs, see *IBM Rational ClearCase MultiSite Administrator's Guide*.

Identifying special element types

The use of element types lets each class of elements be handled differently. An *element type* is a class of file elements. Predefined element types, such as **file** and **text_file**, are included. You can define your own element types. When you create an element type for use in UCM projects, you can specify a **mergetype** attribute, which determines how deliver and rebase operations handle merging of files of that element type.

When a merge situation occurs during a deliver or rebase operation, an attempt is made to merge versions of the element. User interaction is required only if differences between the versions cannot be reconciled. For certain types of files, you may want to impose different merging behavior.

Using mergetype to manage merge behavior

You can use element types for some classes of files for which you want to define a merge behavior that differs from the behavior for predefined element types. For some types of files, you may want to merge versions manually rather than let them be merged automatically. One example is a Visual Basic form file, which is a generated text file. Visual Basic generates the form file based on the form that a developer creates in the Visual Basic GUI. Rather than let the form file be changed during a merge operation, you want to regenerate the form file from the Visual Basic GUI. For this type of file, the developer controls the contents of the file in the target view. The developer might want to copy the version in the development stream or generate a new version.

Some types of files never need to be merged. For these types of files, you may want to ensure that no one attempts to merge them accidentally. For example, the deployment, or staging, component contains the executable files that you ship to customers or install in-house. These files are not under development; they are the product of the development phase of the project cycle. During a deliver operation

(or a rebase operation), an attempt is made to merge these executable files to the target versions unless the files are of an element type for which different merge behavior is specified.

To define different merge behavior for special types of files in your environment, you can create an element type and specify one of the following mergetypes:

- **copy**

  During a merge or findmerge operation within a delivery or a rebase, a version whose element type has the mergetype **copy** is not merged. The source version is copied to the target version without user intervention.
- **never**

  A merge or findmerge operation ignores versions whose element type has **never** as a mergetype.
- **user**

  A merge or a findmerge operation performs trivial merges only. Nontrivial merges must be made manually. The graphic user interface (GUI) tools provide extra options for user mergetype to keep the target version, copy the source version (from the source stream), or back out of the deliver operation.

Note: If you fail to specify a mergetype of **copy**, **never**, or **user** for these element types, developers may encounter problems when they attempt to deliver work or rebase their streams. For example, default merge managers cannot handle data in these files. Developers create executable files when they build and test their work prior to delivering it. If these files are under version control as *derived objects*, they are included in the change set of the current activity.

For information about creating element types, see Chapter 15, "Using element types to customize file element processing," on page 229, and the **mkeltype** reference page in the *IBM Rational ClearCase Command Reference*.

Defining the scope of element types

When you define an element type, its scope can be ordinary or global. By default, the element type is ordinary; it is available only to the VOB in which you create it. If you create the element type in an administrative VOB and define its scope as global, other VOBs that have **AdminVOB** hyperlinks to that administrative VOB can use the element type. If you want to define an element type globally, and you do not currently use a separate administrative VOB, define the element type in the PVOB.

Planning how to use the UCM integration with Rational ClearQuest

Before you can set up the UCM integration with Rational ClearQuest, you need to make some decisions, which fall into two general categories:

- How to map PVOBs to Rational ClearQuest user databases
- Which schema to use for the Rational ClearQuest user databases

Mapping PVOBs to Rational ClearQuest user databases

You need to consider how to use PVOBs for projects that link to Rational ClearQuest user databases.

Rational ClearCase MultiSite requirement

If you use Rational ClearCase MultiSite, all PVOB replicas must have access to the Rational ClearQuest user database. If you have multiple PVOBs linked to either an administrative VOB or a PVOB acting as an administrative VOB, all of the PVOBs and administrative VOBs in the hierarchy must be replicated to all sites. For information on the use of Rational ClearQuest MultiSite, see "How the UCM integration with Rational ClearQuest is affected by Rational ClearQuest MultiSite" on page 104.

Integration requirement for Rational ClearQuest MultiSite

If your organization uses Rational ClearQuest MultiSite, you can register multiple replicas of the same database set (connection). Ensure that developers at their sites us the replica that accesses the user database at their local site. For information on the use of Rational ClearQuest MultiSite, see "How the UCM integration with Rational ClearQuest is affected by Rational ClearQuest MultiSite" on page 104.

Naming projects that are linked to same user database

Although UCM allows you to create projects with the same name in different PVOBs, you cannot link those projects to the same Rational ClearQuest user database (see Figure 31).

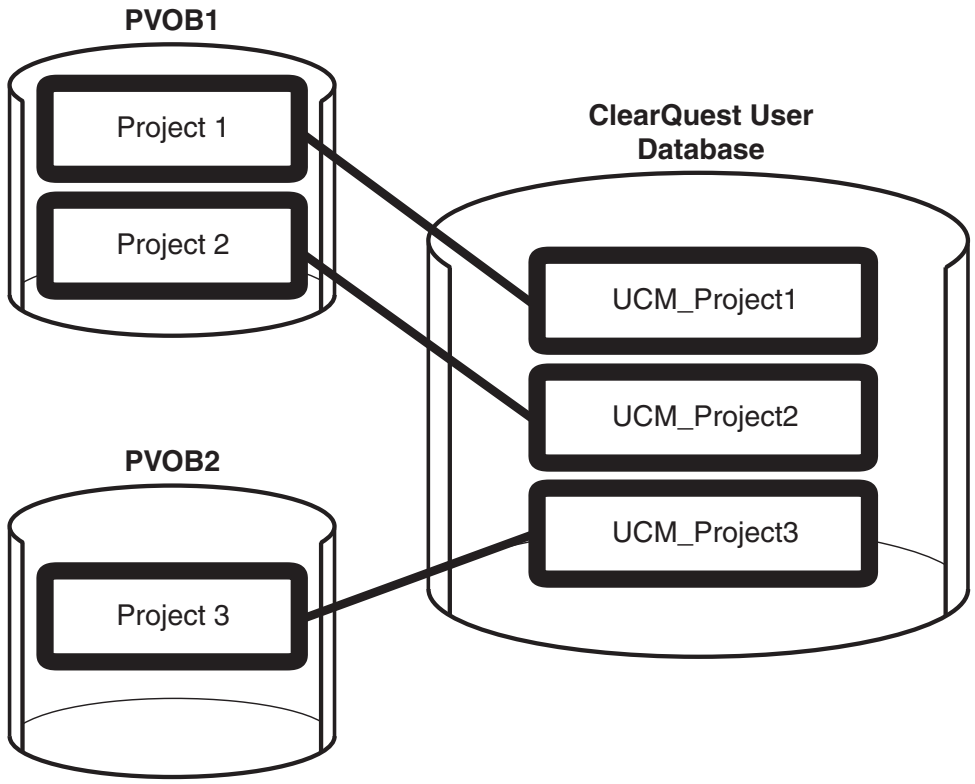


Figure 31. Multiple PVOBs linked to the same Rational ClearQuest user database

Give the projects unique names. For example, in Figure 31, if Project 3 were named either Project 1 or Project 2 (which is valid in the UCM environment), the generated name in the Rational ClearQuest user database would not be unique and would cause the software to run erroneously. This naming requirement states that the project names that appear in the Rational ClearQuest user database must be unique.

Use of multiple user databases

If some developers on your team work on multiple projects, you can store the schemas for the Rational ClearQuest user databases that are linked to those projects in one schema repository, as shown in Figure 32.

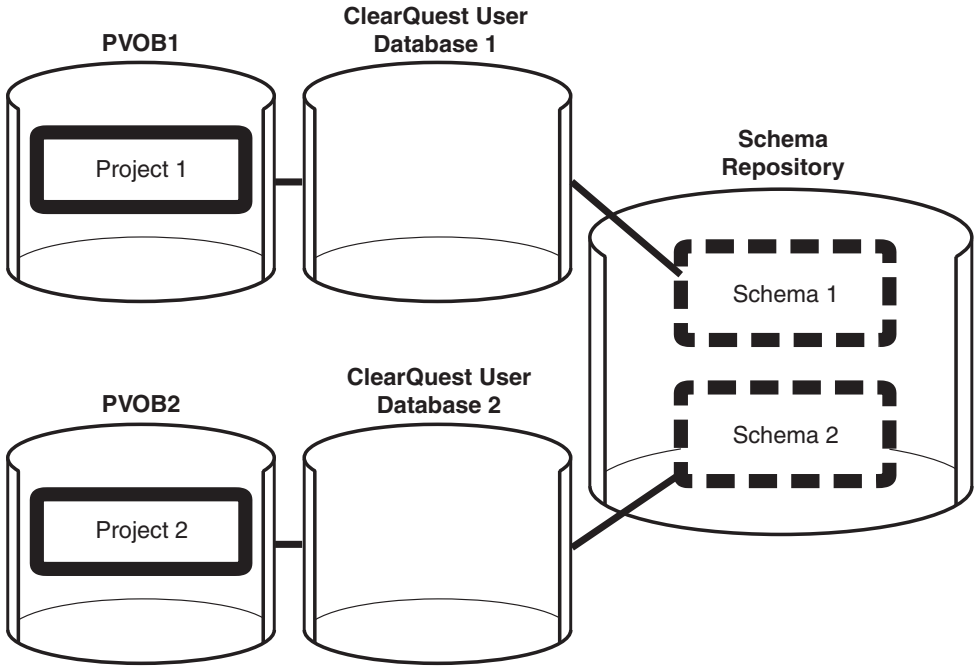


Figure 32. One schema repository for multiple Rational ClearQuest user databases

The databases that are named ClearQuest User Database 1 and ClearQuest User Database 2™ are linked to the same schema repository.

If you use multiple user databases, give each one a unique name. If a developer tries to access a user database whose name is not unique, the authentication can fail. In using the command line interface to be authenticated, a developer can specify only the name of the user database. (The graphic user interface requires that the database set (connection) be supplied.) If multiple user databases share the same name, the software cannot distinguish the difference in databases that have the same name.

Using a single schema repository allows developers to switch between projects easily. If you store the schemas in different schema repositories, developers must connect to each schema repository once when they switch projects. The user name and password are stored locally for each connection that they use. The project manager or developer can provide or update the credentials by using the **cmregister** command.

Deciding which schema to use

To use the UCM integration with Rational ClearQuest, you must create a new Rational ClearQuest user database or upgrade an existing Rational ClearQuest user database that is based on a UCM-enabled schema. A UCM-enabled schema meets the following requirements:

- The **UnifiedChangeManagement** package has been applied to the schema. A package contains metadata, such as records, fields, and states, that define

specific functionality. Applying a package to a schema provides a way to add functionality quickly so that you do not have to build the functionality from scratch.

- The **UnifiedChangeManagement** package has been applied to at least one record type. This package adds fields and scripts to the record type, and adds the **Unified Change Management** tab to the record type's forms.
- The **UCMPolicyScripts** package has been applied to the schema. This package contains the scripts for three Rational ClearQuest development policies that you can enforce.

Rational ClearQuest includes two predefined UCM-enabled schemas: **UnifiedChangeManagement** and **Enterprise**. You can perform the following actions with these schemas:

- Start using the integration readily by using one of the predefined schemas.
- Use the Rational ClearQuest Designer and the Rational ClearQuest Package Wizard to enable a custom schema or another predefined schema to work with UCM.
- Use one of the predefined UCM-enabled schemas as a starting point and modify it to suit your needs.

Overview of the UnifiedChangeManagement schema

The UnifiedChangeManagement schema includes the following record types:

BaseCMActivity
: This is a lightweight record type that you can use to store information about activities that do not require additional fields. You may want to use this record type as a starting point and then modify it to include additional fields and states.

Defect
: This record type is identical to the record type of the same name that is included in other predefined Rational ClearQuest schemas, with one exception: it is enabled to work with UCM. The Defect record type contains more fields and form tabs than the BaseCMActivity record type to allow you to record detailed information.

UCMUtilityActivity
: This record type is not intended for general use. The integration uses this record type when it needs to create records for itself, such as when you link a project that contains activities to a Rational ClearQuest user database. You cannot modify this record type.

Enabling a schema for UCM

If you decide not to use one of the predefined UCM-enabled schemas, do some additional work to enable your schema to work with UCM. Before you can do this, you need to answer the following questions:

- Which record types are you enabling for UCM? You do not need to enable all record types in your schema, but you can link only records of UCM-enabled record types to activities.
- For each UCM-enabled record type:
  - Which state type does each state map to? You must map each state to one of the four UCM state types: Waiting, Ready, Active, Complete. See "Setting state types" on page 78.
  - Which default actions are you using to transition records from one state to another? See "State transition default action requirements for record types" on page 79.

– Which policies do you want to enforce? The integration includes policies that you can set to enforce certain development practices. You can also edit the policy scripts to change the policies. See Chapter 4, “Setting policies,” on page 63 for details.

Chapter 4. Setting policies

UCM includes policies that you can set to enforce certain development practices within a project. The following types are described:

- "Components and baselines policies"
- "Default view types" on page 64
- "Permissions to modify projects and streams" on page 65
- "Policies for all deliver operations" on page 65
- "Policies for deliver operations to nondefault targets" on page 66

Some policies are available only if you enable the project to work with Rational ClearQuest. See "Policies for the UCM integration with Rational ClearQuest" on page 69.

In addition to the policies that UCM supplies, you can create your own policies by using triggers on UCM operations. For information on using triggers, see Chapter 8, "Using triggers to enforce UCM development policies," on page 129.

Components and baselines policies

Some policies are related to components and baselines.

Modifiable components

In most cases, you want components to be modifiable. For information on when to use read-only components, see "Identifying read-only components" on page 33.

Component modifiability and visibility

Component modifiability and visibility can affect the viability of alternate-target deliver operations for migrating changes between two streams in the same project or in different projects (see Figure 33).

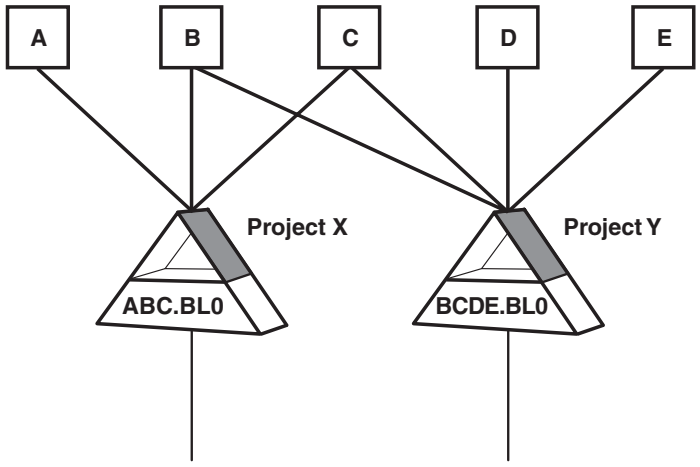


Figure 33. Component modifiability and visibility

From the source stream, an alternate target deliver operation can contain activities with changes in components that (in the target stream) are read-only or are not visible. This condition can occur for one of the following reasons:

- A limited set of components or different sets of components are configured in streams from the same project.
- Streams from different projects can have different modifiability.

  For example, in Figure 33, components **B** and **C** in project **X** can be modifiable while the same components in project **Y** are read-only.
- Streams from different projects can be configured with different sets of components.

  In project **X**, components **D** and **E** are not configured and therefore are not visible. Likewise in project **Y**, component **A** is not configured and is not visible.

By default, if one of these conditions occurs, the deliver operation is prohibited. The project manager can allow such deliver operations to proceed by setting a policy on the target project or stream (see "Require that all source components are visible in the target stream" on page 69).

Default promotion level for recommending baselines

Recommended baselines are typically the set of baselines that project team members use to rebase their development streams. In addition, when developers join the project, their development work areas are initialized with the recommended baselines. When you recommend baselines, the Recommend Baselines window lists the latest baselines that have promotion levels equal to or higher than the promotion level that you specify as the default promotion level for recommending baselines.

Default view types

When developers join a multiple-stream project, they use the Join Project Wizard to create their development views, integration views, and development streams. They use a development view that is attached to a development stream to work in isolation from the project team. They use an integration view that is attached to the parent stream of their development stream to build and test their work against the latest work delivered to the parent stream by other developers.

Two kinds of views are provided: dynamic and snapshot. Specify which type of view to use as the default for development and integration views. When developers join the project, they may choose to accept or reject the default view types. The Join Project Wizard uses the default values the first time that a developer creates views for a project. Thereafter, the wizard uses the developer's most recent selections as the default view types.

Product Note: Rational ClearCase LT supports only snapshot views.

Dynamic views use the Rational ClearCase multiversion file system (MVFS) to provide immediate, transparent access to files and directories stored in VOBs. On Windows systems, a dynamic view is mapped to a drive letter in Windows Explorer. *Snapshot views* copy files and directories from VOBs to a directory on your computer.

Use dynamic views as the default view type for integration views. Dynamic views ensure that when developers deliver work to the integration stream or feature-specific development stream, they build and test their work against the latest work that other developers have delivered since the last baseline was

created. Snapshot views require developers to copy the latest delivered files and directories to their computer (a *snapshot view update* operation), which they may forget to do.

Permissions to modify projects and streams

This section describes policies that control access to objects and determine who can modify the project and stream objects.

Allow all users to modify the project

By default, this policy is disabled, meaning that only the project owner, PVOB owner, or a privileged user can make changes to the project object. To allow all users to modify the project object, enable this policy.

Allow all users to modify the stream and its baselines

By default, this policy is disabled, meaning that only the stream owner, PVOB owner, or a privileged user can make changes to the stream or any baselines created in it. To allow all users to modify the stream and its baselines, enable this policy. You can set this policy to apply to all streams within the project or you can set it on a per-stream basis.

Policies for all deliver operations

Some policies affect all deliver operations. You can set these policies to apply to all streams within the project or you can set the policies on a per-stream basis. When a developer starts a deliver operation, UCM checks the policy settings on the target stream and the project. If the target stream policy setting is different than its project policy setting, the project setting takes precedence. For information on policies that apply only to deliver operations to nondefault targets, see "Policies for deliver operations to nondefault targets" on page 66.

Do not allow deliver to proceed with checkouts in the development stream

This policy prevents developers from delivering work to the target stream if some files remain checked out in the source stream. The policy can be set per project or per stream, for interproject and intraproject deliver operations.

If this policy is enabled, developers must check in all files in their source streams before delivering work. You may want to require developers to check in files to avoid the following situation:

1. A developer completes work on an activity, but forgets to check in all of the files associated with that activity.
2. The developer works on other activities.
3. Having completed several activities, the developer delivers them to the target stream. Because the files associated with the first activity are still checked out, they are not included in the deliver operation. The developer delivers older versions. Even though the developer may build and test the changes successfully in the development work area, the changes delivered to the target may fail because they do not include the checked-out files.

Rebase before delivery

This policy (Require development stream to be based on the project's recommended baseline(s) prior to delivery) requires developers to rebase their

source streams to the target stream current *recommended baselines* before they deliver work to the target stream. The policy can be set per project or per stream, for interproject and intraproject deliver operations.

The goal of this policy is to have developers build and test their work in their development work areas against the work included in the most recent stable baselines before they deliver to the target stream. This practice minimizes the amount of merging that developers must do when they perform deliver operations.

Policies for deliver operations to nondefault targets

Some policies apply only to deliver operations whose targets are not the default target streams. You can set these policies to apply to all streams within the project or you can set the policies on a per-stream basis. When a developer starts a deliver operation, UCM checks the policy settings on the target stream and the project. If the target stream's policy setting is different than its project's policy setting, the project setting takes precedence. For information on policies that apply to all deliver operations, see "Policies for all deliver operations" on page 65.

In a project, you can create a hierarchy of development streams. For details, see "Stream hierarchies" on page 35. Such a hierarchy as shown in Figure 34 allows you to designate a development stream as a shared area for developers working on a particular feature.

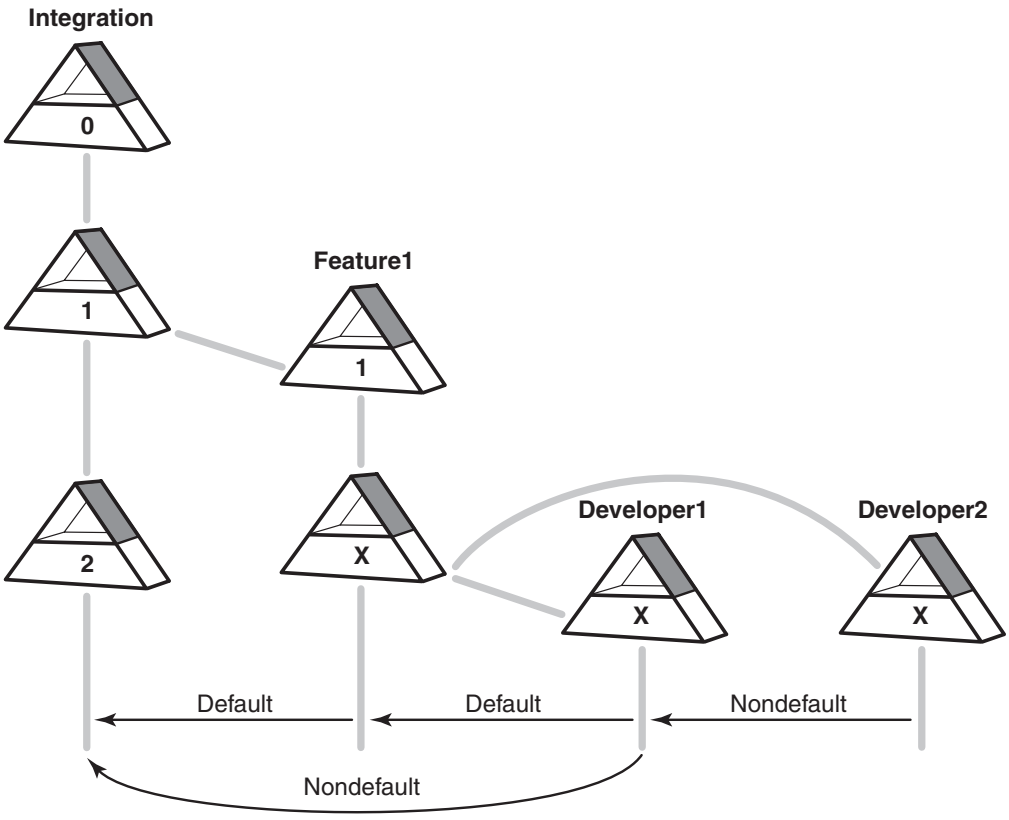


Figure 34. Default and nondefault deliver targets in a stream hierarchy

Developers who work on that feature deliver work to the feature-specific development stream **Feature1**. In Figure 34, the integration stream and the **Feature1** development stream are ancestors of the **Developer1** and **Developer2** development

streams. The streams **Feature1**, **Developer1**, and **Developer2** are descendants of the **integration** stream. The default target for a deliver operation from a development stream is the parent stream of that stream. Developers may also deliver to nondefault target streams. The arrows in Figure 34 illustrate default and nondefault deliver targets. The following policies apply only to such nondefault target streams:

- "Deliver changes from the foundation in addition to changes from the stream"
- "Allow deliveries that contain changes to missing or non-modifiable components" on page 68
- "Allow interproject deliver to project or stream" on page 69
- "Require that all source components are visible in the target stream" on page 69

Deliver changes from the foundation in addition to changes from the stream

Set this policy to control accepting changes that did not originate in the delivery stream. There are two versions of this policy: one for intraproject deliveries and one for interproject deliveries. The policy can be set per project or per stream.

UCM uses foundation baselines to configure a stream. A view attached to a stream selects the versions of elements identified by the stream foundation baselines plus the versions of elements associated with any activities created in the stream. For example, in Figure 35, **1** is the foundation baseline for the **Feature1** development stream. The **X** baseline is the foundation baseline for the **Developer1** development stream.

If the developer working in the **Developer1** stream delivers work to the integration stream, the deliver operation includes the activities created in the **Developer1** stream plus the files represented by the **X** foundation baseline. The integrator responsible for the integration stream may want to receive work that the developer working in the **Developer1** stream has completed; however, the integrator may be unaware that the deliver operation also contains changes made in the **X** baseline. You may want to set this policy to **Disabled** so that target streams do not accept deliver operations that contain changes in the source stream's foundation baselines.

If you set this policy to **Enabled**, the target stream accepts changes in the source stream that result from differences in the foundation baselines of the two streams in addition to changes in the source stream that the developer makes while working on assigned activities.

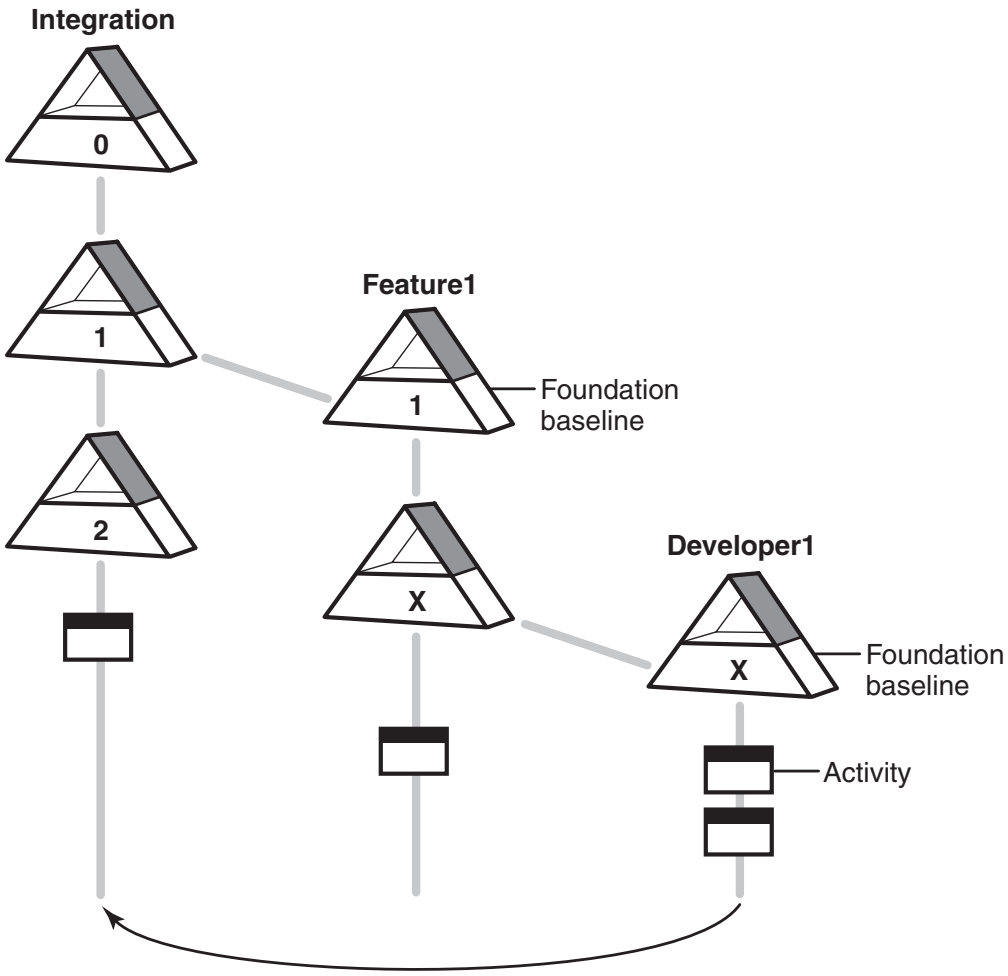


Figure 35. Delivering changes made in a foundation baseline

Allow deliveries that contain changes to missing or non-modifiable components

Set this policy to control whether streams accept deliveries that contain changes to components that are not modifiable in the project of the target stream. The policy can be set per project or per stream. There are two versions of this policy:

- For interproject deliveries–Allow the deliver even though target stream is missing components that are in the source stream
- For intraproject deliveries–Allow the deliver even though modifiable components in the source stream are non-modifiable in the target stream

For information on modifiable components, see "Modifiable components" on page 63.

If you set this policy to **enabled**, UCM allows the deliver operation, but the changes to any missing components or to any non-modifiable components are not included and no errors are generated for the presence of the changes.

Allow interproject deliver to project or stream

Set this policy to control whether streams accept deliveries from streams in other projects. The policy can be set per project or per stream for interproject deliveries. See Chapter 9, "Managing multiple projects," on page 141, for examples of when you may want to deliver work from one project to another.

Require that all source components are visible in the target stream

Set this policy to control whether streams accept deliveries that contain changes to components that are not in the target stream configuration. The policy can be set per project or per stream. For information about component visibility, see "Component modifiability and visibility" on page 63.

If you set this policy to **disabled**, UCM allows the deliver operation, but the changes to any missing components are not included.

This policy is ignored if interproject delivery is disabled.

Policies for the UCM integration with Rational ClearQuest

Some policies are available only when you enable the project to work with Rational ClearQuest. Some of the policies are customizable. In the Rational ClearQuest environment, scripts are used to implement the customizable policies. You can modify a policy behavior by editing its script. See "Customizing Rational ClearQuest project policies" on page 81.

These policies apply to the UCM package that is supported by the current version of Rational ClearQuest. If the user database uses an earlier UCM package version, some of the policies that are shown here may not be available.

The policies apply to the following usages:

- "For submitting records from a Rational ClearCase client"
- "For WorkOn" on page 70
- "For delivery" on page 70
- "For changing activities" on page 72

For submitting records from a Rational ClearCase client

These policies affect the configuration of the UCM project. They are not visible from Rational ClearQuest forms because they do not have corresponding Rational ClearQuest hooks.

Disallow submitting records from ClearCase client

Set this policy to prevent developers from creating new activity records in the project while they are working under source control. This policy is invoked when a developer attempts to create a new activity. If this policy is set, when developers work in source control, they cannot create new activities in which their changes are recorded. In the graphic user interface, the **New** button is disabled for checkout, checkin, cancel checkout and add to source control. The purpose is to restrict creation of activity records to users, for example, project managers, who have specific permission within the Rational ClearQuest schema.

If you disable this policy, developers can create new activity records in the Rational ClearQuest user database when they are working under source control. Also, you

can determine which record types are used in the Rational ClearQuest user database for new activity records that are created outside the Rational ClearQuest client (see "Allowed record types").

This policy is not customizable.

Allowed record types

If you disable the Disallow submitting records from ClearCase client policy (see "Disallow submitting records from ClearCase client" on page 69), you can specify the record types that are allowed for new activity records in the Rational ClearQuest user database from a Rational ClearCase client. When developers need to create an activity, they see only the record types that you specify in this policy.

By default, the record types that are enabled by the UCM package (except UCMUtilityActivity) are allowed.

This policy is not customizable.

For WorkOn

The policy described in "Perform ClearQuest action before work on" applies when the developer clicks **WorkOn** in the Rational ClearQuest record form.

Perform ClearQuest action before work on

This policy is invoked when a developer attempts to set an activity. The default policy script checks whether the developer's user name matches the name in the Rational ClearQuest record Owner field. If the names match, the developer can work on the activity. If the names do not match, the WorkOn fails.

The intent of this policy is to ensure that all criteria are met before a developer can start working on an activity. You may want to modify the policy to check for additional criteria.

For delivery

The policies described in this section apply when developers deliver their work in their project.

Perform ClearQuest action before delivery

This default policy script is a placeholder: it does nothing. This policy is invoked when a developer attempts to deliver an activity in a UCM-enabled project. You can edit the script to implement an approval process to control deliver operations. For example, you may want to add an **Approved** field to the record type of the activity and require that the project manager set it before allowing developers to deliver activities.

See "Policies and interproject deliveries" on page 73 for details about deliveries between two projects that are enabled for Rational ClearQuest.

Perform ClearQuest action after delivery

This policy is invoked at the end of a deliver operation for each activity included in the deliver operation. The default policy script is a placeholder: it does nothing.

You may want to edit this script to implement a post-delivery development practice. For example, you might want the script to send an e-mail message to all developers on the project telling them that a deliver operation has just finished.

See “Policies and interproject deliveries” on page 73 for details about deliveries between two projects that are enabled for Rational ClearQuest.

Transition to complete after delivery

This policy is invoked at the end of a deliver operation for each activity included in the deliver operation. The policy uses the default action of the activity to transition the activity to a Complete type state and unset the activity from its view. These actions prevent the checkouts of versions in the change set from being associated with the activity.

If the default action requires entries in certain fields of the activity record, and one of those fields is empty, the policy returns an error and leaves the deliver operation in an uncompleted state. This state prevents the developer from performing another deliver operation, but it does not affect the current one. It does not roll back changes made during the merging of versions.

To recover from an error, the developer needs to fill in the required fields in the activity record and resume the deliver operation. If the developer invoked the deliver operation from a graphic user interface (GUI), the integration displays the Rational ClearQuest record form so that the developer can fill in the fields.

This policy is not customizable.

See “Policies and interproject deliveries” on page 73 for details about deliveries between two projects that are enabled for Rational ClearQuest.

Transfer ClearQuest mastership before delivery

The Transition to Complete After Delivery project policy transitions activities to a Complete type state when a deliver operation completes successfully. For that policy to work correctly in a Rational ClearCase MultiSite environment, the activities being delivered must be mastered by the same replica that masters the target stream. To ensure that this is the case, you can set the Transfer Mastership Before Delivery policy.

The behavior of the Transfer Mastership Before Delivery policy depends on whether the deliver operation is local or remote. If the deliver operation is local, meaning that the target stream is mastered by the local PVOB replica, this policy causes the deliver operation to fail unless all activities being delivered are mastered locally.

A *remote deliver* operation is one for which the target stream is mastered by a remote PVOB replica. The developer starts the deliver operation, but the operation is left in a *posted* state. The integrator at the remote site completes the deliver operation.

For a remote deliver operation, the Transfer Mastership Before Delivery policy causes the following behavior:

- If all activities in the deliver operation are mastered by the remote replica, the deliver operation is allowed to proceed.
- If the deliver operation contains activities that are mastered by the local replica, Rational ClearCase MultiSite transfers mastership of those activities to the remote replica. To have Rational ClearCase MultiSite transfer mastership of those activities back to the local replica after the integrator at the remote site performs any required merges and completes the deliver operation, set the Transfer ClearCase Mastership After Delivery policy also.

- If the deliver operation contains activities that are mastered by a third replica, the deliver operation fails.

This policy is not customizable.

See "Policies and interproject deliveries" on page 73 for details about deliveries between two projects that are enabled for Rational ClearQuest.

Transfer ClearQuest mastership after delivery

Use this policy only in conjunction with the Transfer ClearQuest Mastership Before Delivery policy. The Transfer ClearQuest Mastership Before Delivery policy transfers mastership of activities involved in a remote deliver operation from the local replica to a remote replica. Set this policy if you want to transfer mastership of those activities back to the original (local) replica after the integrator at the remote site completes the deliver operation.

This policy is not customizable.

See "Policies and interproject deliveries" on page 73 for details about deliveries between two projects that are enabled for Rational ClearQuest.

For changing activities

The policies described in this section apply when developers work on their activities.

Perform ClearQuest action before changing activity

This default policy script is a placeholder: it does nothing. This policy is invoked when a developer attempts to finish an activity. The finish activity operation checks in all files that belong to the activity change set and performs Rational ClearQuest actions, such as modifying the state of the activity to a Complete type state, based on the policies that you set. When invoked in a single-stream project or on the integration stream of a multiple-stream project, the finish activity operation is similar to a deliver operation in a multiple-stream project. Both operations share changes with the rest of the team.

You can edit the script to implement an approval process to control finish activity operations. For example, you may want to add an **Approved** field to the record type of the activity and require that the project manager set it before allowing developers to finish activities.

Perform ClearQuest action after changing activity

This policy is invoked when a developer attempts to finish an activity. If the Perform ClearQuest Action Before Changing Activity policy is set, this policy is invoked first. The default policy script behaves as follows:

- For developers working in a single-stream project or on the integration stream of a multiple-stream project, allow the finish activity operation.
- For developers working on a development stream, check in all files that belong to the activity's change set but do not perform any Rational ClearQuest actions.

You may want to edit this script to implement a post-finish activity development practice. For example, you might want the script to send an e-mail message to all developers on the project telling them that a developer has just checked in files and finished an activity.

Transition to complete after changing activity

This policy is invoked at the end of a finish activity operation. The policy uses the activity's default action to transition the activity to a Complete type state. If the default action requires entries in certain fields of the activity record, and one of those fields is empty, the policy returns an error. To recover from an error, the developer needs to fill in the required fields in the activity record.

You may want to transition activities to a Complete type state depending on whether the developer works in an integration stream.

- To transition activities only for developers who work in a single-stream project or on the integration stream of a multiple-stream project, set this policy and the Perform ClearQuest Action After Changing Activity policy.
- To transition activities regardless of which stream the developer works on, set this policy and clear the Perform ClearQuest Action After Changing Activity policy.

This policy is not customizable.

Policies and interproject deliveries

With one exception, the integration does not invoke the following policies when you deliver from one project that is enabled for Rational ClearQuest to another that is also enabled for Rational ClearQuest:

- Perform ClearQuest Action Before Delivery
- Perform ClearQuest Action After Delivery
- Transition to Complete After Delivery
- Transfer ClearQuest Mastership Before Delivery
- Transfer ClearQuest Mastership After Delivery

The integration invokes the policies that are set for the project of the source stream only if the following conditions are true:

- The source and target streams are integration streams.
- The target stream is the default deliver target for the source stream.

Chapter 5. Setting up a Rational ClearQuest user database for UCM

This chapter describes how to set up a Rational ClearQuest user database for use with UCM. For information about the decisions that you need to make before you set up the integration, see "Planning how to use the UCM integration with Rational ClearQuest" on page 57.

About setting up a Rational ClearQuest user database

To use the UCM integration with Rational ClearQuest for your project, set up a Rational ClearQuest user database.

The steps to do the setup are typically completed by the Rational ClearQuest user database administrator or the schema designer. You have the following options.

- Take advantage of predefined schemas that Rational ClearQuest includes. These are ready for use with UCM (see "Using the predefined UCM-enabled schemas" on page 75).
- Enable a custom schema, or another predefined schema, to work with UCM. This allows you to use UCM with a current Rational ClearQuest configuration (see "Adding UCM support to an existing schema" on page 75). Because this integration is a dependent integration, you must add one or more packages in a specific order and perform additional configurations to the Rational ClearQuest user database.

Using the predefined UCM-enabled schemas

The easier way to set up a Rational ClearQuest user database for UCM is to use either the **UnifiedChangeManagement** or the **Enterprise** predefined UCM schema. Each schema already includes the record type, field, form, state, and other definitions necessary to work with a UCM project. Follow the procedure described in "To set up a Rational ClearQuest user database to work with UCM."

To set up a Rational ClearQuest user database to work with UCM

1. Create a user database that is associated with one of the predefined UCM-enabled schemas. In the Rational ClearQuest Designer, click **Database > New Database** to start the New Database Wizard.
2. Complete the steps in the wizard. Step 4 prompts you to select a schema to associate with the new database. Scroll the list of schema names and select either the **UnifiedChangeManagement** or the **Enterprise** schema.
3. Click **Finish**.

Adding UCM support to an existing schema

The predefined UCM schemas let you use the UCM integration with Rational ClearQuest right away (see "Using the predefined UCM-enabled schemas"). However, you may prefer to design a custom schema to track your project activities and change requests, or you may prefer to use a different predefined schema. For your schema to work with UCM, you need to apply several packages

in a prescribed order. These packages must be added in the order described for each step. Integrating your schema with UCM packages requires that the following actions be done in the order described:

1. Adding the AMStateTypes Package.
2. Setting the Default Actions for UCM.
3. Adding the UCMPolicyScripts Package.
4. Adding the UnifiedChangeManagement Package.
5. Adding the UCMProject Package.
6. Adding the BaseCMActivity Package (optional).
7. Saving the Schema Changes.
8. Configuring Rational ClearCase UCM.

Note: To avoid errors, you must install packages in the order described.

The AMStateTypes Package provides additional support for UCM and its state types. Installing this package requires that you map schema states to the following state types: Waiting, Ready, Active, and Complete. The package adds the am_statetype field to the enabled record type.

The UCMPolicyScripts Package adds three global scripts and does not add any record types.

The **UnifiedChangeManagement** package does the following:

- Adds the UCMUtilityActivity record type.
- Adds UCM_Project stateless record type.
- Adds UCM queries to the Rational ClearQuest client workspace under the **Public Queries** folder.
- Adds the ucm_base_ synchronize action to the enabled record type.

Although the **BaseCMActivity** package is not necessary, you may want to apply it to your schema. The **BaseCMActivity** package adds the BaseCMActivity record type to your schema. The BaseCMActivity record type is a lightweight activity record type. You may want to use the BaseCMActivity record type as a starting point and then modify it, for example, to include additional fields and states. If you want to rename the BaseCMActivity record type, be sure to do so before you create any records of that type.

To enable a schema to work with UCM

1. In the Rational ClearQuest Designer, ensure that the schema does not contain a record type named **UCM_Project**, which is a reserved name used by the UCM integration with Rational ClearQuest.
2. Ensure that the schema to which you are adding packages is checked in. To check in a schema, click **File > Check In**.
3. Click **Package > Package Wizard** to start the Package Wizard.
4. Add the latest **AMStateTypes** package to the schema. For information on applying packages, see *IBM Rational ClearQuest MultiSite Administrator's Guide*.

   The AMStateType package requires you to map state types (see "To map record states to state types" on page 77) and set default actions, if they have not already been defined.
5. Set the default actions for UCM (see "To set default actions for states" on page 80).
6. Add the **UCMPolicyScripts** package to the schema. For information on applying packages, see *IBM Rational ClearQuest MultiSite Administrator's Guide*.

7. Add the **UnifiedChangeManagement** package to the schema. For information on applying packages, see *IBM Rational ClearQuest MultiSite Administrator's Guide*.

   The **AMStateTypes** package is also applied.
8. In the second page of the wizard, select your schema. Click **Next**.
9. The third page of the wizard prompts you to specify the schema record types. Set the check boxes of the record types that you want to enable. Click **Finish**. All selected record types must meet the requirements listed in "Requirements for enabling custom record types" on page 78.
10. In the **Setup State Types** page, assign state types to the states for each record type. For information about assigning state types, see "Assigning state types to the states of a record type." For information about performing the task, see "To map record states to state types." Click **Finish**.
11. Set default actions for the states of each enabled record type (see "To set default actions for states" on page 80). Default actions are state transition actions that are taken when a developer begins to work on an activity or delivers an activity (see "State transition default action requirements for record types" on page 79).
12. In the Rational ClearQuest Designer workspace, navigate to the record type **Behaviors** (click *schema* **> Record Types > Record Type > States and Actions > Behaviors**). Double-click **Behaviors** to display the Behaviors grid.
    a. Verify that the **Headline** field is set to **Mandatory** for all states.
    b. Verify that the **Owner** field is set to Mandatory for all Active and Ready state types.
13. Validate the schema changes by clicking **File > Validate**. Fix any errors that are displayed, and then check in the schema by clicking **File > Check In**.
14. Do one of the following actions:
    - Click **Database > Upgrade Database** to upgrade the user database so that it is associated with the UCM-enabled version of the schema.
    - Create a new user database that is based on the UCM-enabled version of the schema.

Assigning state types to the states of a record type

For each record type that you choose to enable (see "To enable a schema to work with UCM" on page 76), you must map its states to state types that are defined in the **AMStateTypes** package. For example, when you apply the **UnifiedChangeManagement** package to the schema, the UCMUtiltityActivity record type is added. If you try to check in the schema with these changes, you see messages that describe validation errors. You need to map the states of the UCMUtiltityActivity record type to the states in the AMStatesTypes package. Likewise, if you apply the BaseCMActivity package to the schema, map the states of the BaseCMActivity record type to the state of the AMStatesTypes package.

You see these validation errors because the AMStatesTypes package is applied to the schema when you apply the **UnifiedChangeManagement** package. To eliminate the validation errors and be able to check in the schema, for each record type that is added, map its states to the state types of the AMStatesTypes package.

To map record states to state types

1. Run the Rational ClearQuest Designer.
2. Do one of the following:

- If you are running Package Wizard in the Rational ClearQuest Designer, advance to the **Setup State Types** page.
- If the **Setup State Types** page of the wizard does not appear, click **Package > Setup State Types**.

3. In the **Setup State Types** window, for each record type that is listed in **Record Type**, do the following:
   a. For each state in the **States** column, click in the adjacent cell under **State Type** to display the list of available state types.
   b. Select one of the entries.
   c. To display the states of another record type, click the arrow in the **Record Type** and select another of the available record types.

   See "Setting state types" on page 78 for a description of the four state types, and the rules for setting them.

Requirements for enabling custom record types

Before you can apply the **UnifiedChangeManagement** package to a custom record type (see "To enable a schema to work with UCM" on page 76), the record type must meet the following requirements:

- It contains a field named **Headline** defined as a SHORT_STRING, and a field named **Owner** defined as a REFERENCE to the **users** record type that is supplied with Rational ClearQuest. The **Headline** field must be at least 120 characters long.
- It does not contain fields with these names:
  - **ucm_vob_object**
  - **ucm_stream**
  - **ucm_stream_object**
  - **ucm_view**
  - **ucm_project**
- It contains an action named **Modify** of type Modify.
- It contains a state named **Submitted**.

Note: You can change the name of the state after you apply the **UnifiedChangeManagement** package.

Setting state types

The UCM integration with Rational ClearQuest uses a state transition model to help you monitor the progress of activities. To implement this model, the integration adds state types to UCM-enabled schemas. Table 2 lists and describes the four state types. You must assign each state to a state type (see "To map record states to state types" on page 77). You must have at least one state definition of state type Waiting, one of state type Ready, one of state type Active, and one of state type Complete.

Table 2. State Types in UCM-Enabled Schema

State type	Description
Waiting	The activity is not ready to be worked on, either because it has not been assigned or it has not satisfied a dependency.
Ready	The activity is ready to be worked on. It has been assigned, and all dependencies have been satisfied.
Active	The developer has started work on the activity but has not completed it.

Table 2. State Types in UCM-Enabled Schema (continued)

State type	Description
Complete	The developer has either worked on and completed the activity, or not worked on and abandoned the activity.

State transition default action requirements for record types

Record types can include numerous state definitions. However, UCM-enabled record types must have at least one path of transitions among state types as follows: Waiting to Ready to Active to Complete (see "Setting state types" on page 78). The transition from one state to the next must be made by a default action.

For example, Figure 36 shows the actions and default actions between the states defined in the UCM-enabled BaseCMActivity record type included in the predefined UCM schema. The states are **Submitted**, **Ready**, **Active**, and **Complete**. The corresponding state types appear to the right of the states.

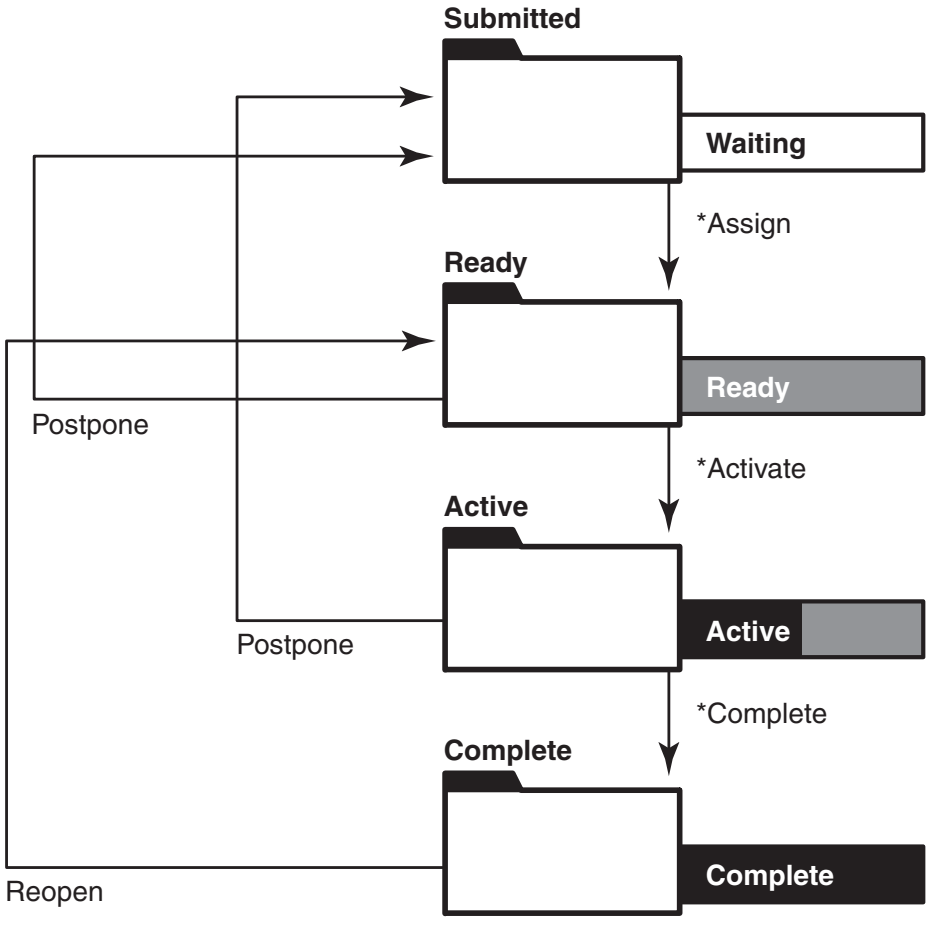


Figure 36. State transitions of UCM-enabled BaseCMActivity record type

In addition to this single path requirement, states must adhere to the following rules:

- All Waiting type states must have a default action that transitions to another Waiting type state or to either a Ready or Active type state.

- If a Ready type state has an action that transitions directly to a Waiting type state, that Waiting type state must have a default action that transitions directly to that Ready type state.
- All Ready type states must have a default action that transitions to another Ready type state or to an Active type state.
- All Ready type states must have at least one action that transitions directly to a Waiting type state.
- For the BaseCMActivity record type, its initial state must be a Waiting type.

To set default actions for states

1. In the Rational ClearQuest Designer workspace, click **Record Types > Activity > States and Actions** to navigate to the record type state transition matrix.
2. Double-click **State Transition Matrix** to display the matrix.
3. Right-click the state column heading, and click **Properties**.
4. Click the **Default Action** tab. Select the default action. See "State transition default action requirements for record types" on page 79 for default action requirements.

   Before you can set default actions, you may need to add some actions to the record type. To do so, double-click **Actions** to display the Actions grid, and then click **Edit > Add Action**.
5. Click **File > Check In** to check in the schema.

Upgrading your schema to the latest UCM package

If you have a UCM-enabled Rational ClearQuest schema from a previous release of Rational ClearQuest, you may want to upgrade that schema with the latest revision of the **UnifiedChangeManagement** package so that you can use new functionality.

To upgrade the schema

1. In the Rational ClearQuest Designer, click **Package > Upgrade Installed Packages** to start the Upgrade Installed Packages Wizard.
2. The first page of the wizard lists all schemas that have at least one package that needs to be upgraded. Select the schema that you want to upgrade, and click **Next**.
3. The second page of the wizard lists the packages that will be upgraded. Click **Upgrade** to accept the changes.
4. If the **UnifiedChangeManagement** package from which you are upgrading is earlier than revision 3.0, you need to assign states to state types for each UCM-enabled record type. For information about performing this operation, see "To map record states to state types" on page 77.
5. Validate the schema changes by clicking **File > Validate**. Fix any errors that are displayed, and then check in the schema by clicking **File > Check In**.
6. Upgrade the user database to associate it with the new version of the schema by clicking **Database > Upgrade Database**.

Customizing Rational ClearQuest project policies

To implement the project policies, the integration adds the following pairs of scripts to a UCM-enabled schema:

- **UCM_ChkBeforeDeliver** and **UCM_ChkBeforeDeliver_Def**
- **UCM_ChkBeforeWorkOn** and **UCM_ChkBeforeWorkOn_Def**
- **UCM_CQActAfterDeliver** and **UCM_CQActAfterDeliver_Def**
- **UCM_CQActBeforeChact** and **UCM_CQActBeforeChact_Def**
- **UCM_CQActAfterChact** and **UCM_CQActAfterChact_Def**

Each policy has two scripts: a base script and a default script. The default scripts have **_Def** appended to their names and are installed by the **UnifiedChangeManagement** package. The integration invokes the base scripts, which are installed by the **UCMPolicyScripts** package. The base scripts call the corresponding default scripts. You can modify the behavior of a policy by editing the base script (see "To modify the behavior of a policy").

Each script has a Visual Basic version and a Perl version. The Visual Basic scripts have a **UCM** prefix. The Perl scripts have a **UCU** prefix. For Rational ClearQuest clients on the Windows system, the integration uses the Visual Basic scripts. For Rational ClearQuest clients on Linux® and the UNIX system, the integration uses the Perl scripts. If you modify a policy behavior and your environment includes Rational ClearQuest clients on different types of platforms, be sure to make the same changes in both the Visual Basic and Perl versions of the policy script. Otherwise, the policy will behave differently for Rational ClearQuest clients on Linux and the UNIX system and the Windows system.

For descriptions of these policies, see "Policies for the UCM integration with Rational ClearQuest" on page 69.

To modify the behavior of a policy

1. Remove the call to the default script from the base script.
2. Add logic for the new behavior to the base script.

Adhere to the rules stated in the base script.

Associating child activity records with a parent activity record

As project manager, you may assign activities for large tasks to developers. When the developers research their activities, they may determine that they need to perform several separate activities to complete one large activity. For example, an "Add customer verification functionality" activity may require significant work in multiple product components, for example, the graphic user interface (GUI), the command-line interface, and a library. To more accurately track the progress of the activity, you can decompose it into three separate activities.

By using the parent and child controls in Rational ClearQuest, you can accomplish this decomposition and tie the child activities back to the parent activity.

Using parent and child controls

In the Rational ClearQuest interface, you use controls to display fields in record forms. A parent and child control, when used with a reference or reference list field, lets you link related records. By adding a parent and child control to the

record form of a UCM-enabled record type, you can provide the developers on your team with the ability to decompose a parent activity into several child activities.

To have the state of the parent activity changed to Complete when all child activities have been completed, you need to write a hook. See *IBM Rational ClearQuest MultiSite Administrator's Guide* for an example of such a hook.

Creating users and adding credentials

Before you can assign activities to the developers on your project team, you must create in a Rational ClearQuest user database user account profiles for each developer. See *IBM Rational ClearQuest MultiSite Administrator's Guide* and the Rational ClearQuest Designer Help for details on creating user profiles. You must also add credentials that allow users to be logged in to the Rational ClearQuest user databases that they need to access.

To create Rational ClearQuest user account profiles

1. In Rational ClearQuest Designer, click **Tools > User Administration**.
2. Click **User Action > Add User**.
3. Complete the Add User window.

Creating and maintaining credentials for Rational ClearQuest database sets

The UCM integration with Rational ClearQuest supports multiple database sets (connections). Each connection for each user database on a system requires credentials: the user name and password for access to the Rational ClearQuest user database. If a user of the integration performs an action that requires the login to a different database, the integration accesses stored credentials and attempts to perform the login. You or the developer can register the credentials.

The usage of credentials differs between the Rational ClearCase command line interface (CLI) and the graphic user interface (GUI). For a CLI user, if the credentials are not registered, the user sees a message requesting that the credentials be created. For a GUI user, a login window is displayed for the credentials to be supplied. The GUI stores the credentials for that user database after a successful login. After credentials are registered, the user does not have to enter credentials to subsequently use the integration on that system for that database set.

In environments where multiple database sets are used with the integration, the user may need to be logged in to a different user database. Credentials are required for a change in login in the following usages:

- The project manager enables a UCM project to use the integration.
- A user displays or sets project policies related to the integration.
- A developer works on an activity in a project that is enabled for the integration.
- A user displays properties of a Rational ClearQuest activity record that has contributions from a project that is connected to another Rational ClearQuest user database.
- A developer who accesses multiple UCM projects that are connected to different Rational ClearQuest database sets does one of the following operations:
  - Starts a delivery in one project while working in another project.

  - Delivers changes in a stream in one project to a stream in another project.
- A developer transitions a Rational ClearQuest activity record.

Create and maintain the credentials on the user's system for each database set. The credential information is stored on the system so that the user can be logged in by the integration server when a different user database is accessed. If no credentials exist for a user when a connection to another user database is required, an error message is generated.

Use the **crmregister** command for the following purposes:

- Create a new entry or overwrite an existing entry. For example:

```
crmregister add -database MY_DB -connection 07.00
-user jsmith -password mypassword
```

- Delete an entry or all entries. For example:

```
crmregister remove -database MY_DB
crmregister remove -all
```

- Modify the specified fields for a specified user database. For example:

```
crmregister replace -database MY_DB -password mynewpassword
```

Setting the environment (Linux and the UNIX system)

Before you can enable a UCM project to work with a Rational ClearQuest user database, you must define the environment variables as shown in Table 3. Developers who want to use the integration must also define these variables on their machines.

The Rational ClearQuest installation directory includes a C shell script, **cq_setup.csh**, which you can run to set the environment variables for you. For example:

```
% source cquest–home–dir/cq_setup.csh
```

Table 3. Environment variables required for integration

Variable	Setting
$CQ_HOME	*cquest–home–dir*
$LD_LIBRARY_PATH $SHLIB_PATH (on HP-UX)	Must include: *cquest–home–dir*/shlib *cquest–home–dir*/*architecture*/shlib
$SQUID_DBSET	If you have multiple Rational ClearQuest schema repositories, set the environment variable to the name of the schema repository to use.

Chapter 6. Setting up the project

This chapter describes how to set up a UCM project.

About setting up the project

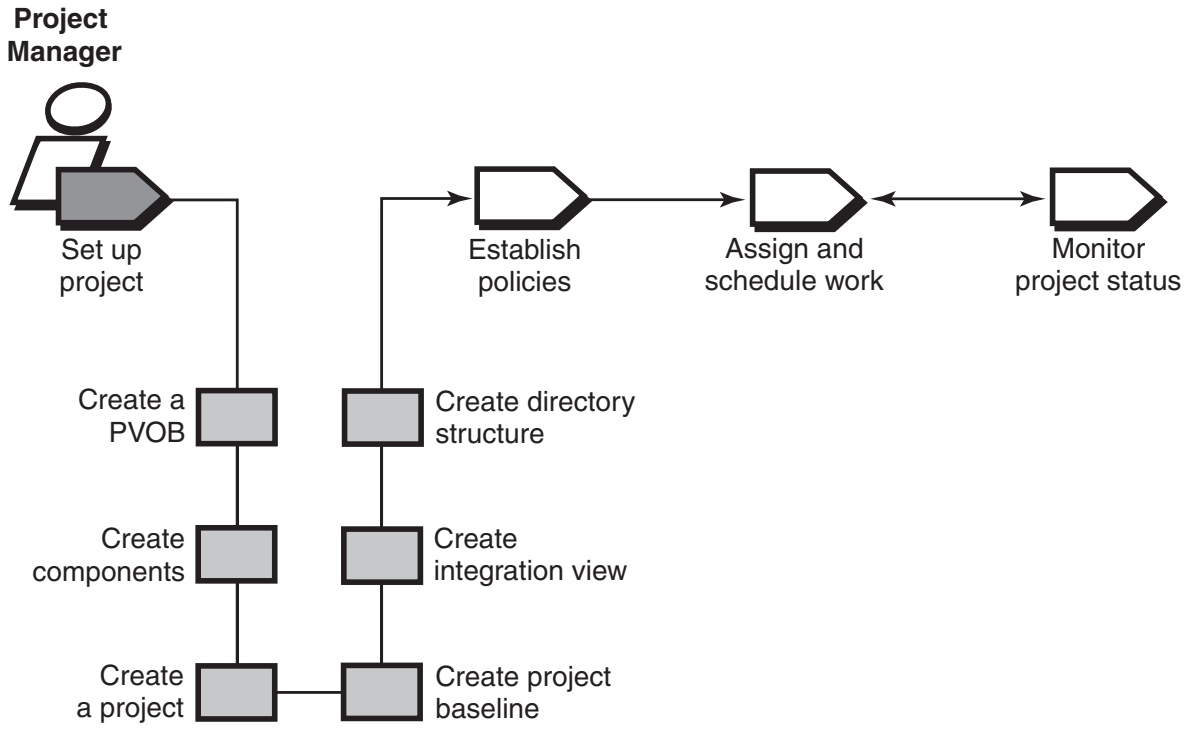
You set up a project so that a team of developers can work in the Unified Change Management (UCM) environment. Before you set up a project, be sure to plan the project. See Chapter 3, "Planning the project," on page 29 for information on what to include in a configuration management plan.

In setting up a project, you may encounter the following scenarios:

- Creating a project from scratch
- Creating a project based on an existing base ClearCase configuration
- Creating a project based on an existing project
- Enabling a project to use the UCM integration with Rational ClearQuest
- Working with Rational Suite
- Creating a development stream reserved for testing new baselines
- Creating a feature-specific development stream

If you work in a multiple project environment, see Chapter 9, "Managing multiple projects," on page 141.

Creating a project from scratch



You can create and set up a new project that is not based on an existing project or on an existing set of VOBs.

Creating the project VOB

In setting up a project from scratch, create a project VOB (PVOB). Creating the PVOB differs between the Windows system and Linux or the UNIX system. After you create the PVOB, you can create components.

To create a PVOB (the Windows system)

Product Note: This task does not apply to Rational ClearCase LT users. The Rational ClearCase administrator creates the PVOB during the installation.

1. Start the VOB Creation Wizard (see "To start the VOB Creation Wizard (the Windows system)" on page 86).
2. In Step 1 of the VOB Creation Wizard, enter a name for the PVOB. Enter a comment to describe the purpose of the PVOB. Leave the **This VOB will contain UCM components** check box clear. Although you can use one VOB as the PVOB and a component, do not do so unless your project is very small and you anticipate that it will remain small. Set **Create as a UCM project VOB**.
3. In Step 2, specify the PVOB storage directory. A PVOB storage directory is a directory tree that serves as the repository for the PVOB contents. A PVOB storage directory contains the same subdirectories as a VOB storage directory. (For details about VOB storage directory structure, see the *IBM Rational ClearCase Administrator's Guide*.) You can choose one of the recommended locations or enter the universal naming convention (UNC) path of a different location. Click **Browse** to search the network for shared resource locations.
4. Step 3 prompts you to choose an administrative VOB to be associated with the PVOB. Because you are creating a project from scratch and do not currently use an administrative VOB, scroll to the top of the list and select **none**. When you create components, **AdminVOB** hyperlinks are made between the components and the PVOB, and the PVOB assumes the role of administrative VOB.

   If you are creating multiple PVOBs and anticipate that projects in those PVOBs may need to modify some of the same components, choose one PVOB to act as the administrative PVOB and create it first. When you create the other PVOBs, use this step in the wizard to specify the PVOB that will serve as administrative VOB. When you create components, **AdminVOB** hyperlinks are made between the components and the PVOB that serves as the administrative VOB. See "Planning PVOBs" on page 53 for details about using multiple PVOBs.

To start the VOB Creation Wizard (the Windows system)

On the Rational ClearCase server host, click **Start > Programs > IBM Rational > IBM Rational ClearCase > Administration > Create VOB**. The VOB Creation Wizard is displayed.

To create a PVOB (Linux and the UNIX system)

Product Note: This task does not apply to Rational ClearCase LT users. The Rational ClearCase administrator creates the PVOB during the installation.

1. Issue the **cleartool mkvob** command. For example:

```
cleartool mkvob –tag /vobs/myproj2_pvob –nc –ucmproject \
/usr/vobstore/myproj2_pvob.vbs
```

The **–ucmproject** option indicates that you are creating a PVOB instead of a VOB. The /usr/vobstore/myproj2_pvob.vbs path specifies the location of the PVOB storage directory. A PVOB storage directory is a directory tree that serves as the repository for the PVOB contents. A PVOB storage directory contains the same subdirectories as a VOB storage directory. For details about VOB storage directory structure, see the *IBM Rational ClearCase Administrator's Guide*.

If you are in an MVFS environment and developers use dynamic views, perform the following two steps.

2. Create the PVOB mount point to match the PVOB tag. For example:

   ```
   mkdir /vobs/myproj2_pvob
   ```

3. Mount the PVOB. For example:

   ```
   cleartool mount /vobs/myproj2_pvob
   ```

The PVOB assumes the role of administrative VOB. When you create components, **AdminVOB** hyperlinks are automatically made between the components and the PVOB.

If you are creating multiple PVOBs and anticipate that projects in those PVOBs may need to modify some of the same components, choose one PVOB to act as the administrative PVOB and create it first. When you create the other PVOBs, use the **cleartool mkhlink** command to create an **AdminVOB** hyperlink between each PVOB and the PVOB that acts as the administrative VOB. For more information about using multiple PVOBs, see "Planning PVOBs" on page 53.

When you create components, **AdminVOB** hyperlinks are made between the components and the PVOB that serves as the administrative VOB.

Creating components for storing baseline dependencies

After you create the PVOB (see "Creating the project VOB" on page 86), you can create components whose sole function is to store baseline dependencies. If you create components without a VOB root directory, nobody can create elements in the components. A component that has no VOB root directory cannot store its own elements. Although you can store baseline dependencies and elements in the same component, it is cleaner to dedicate components for storing baseline dependencies. For the most configuration flexibility, use pure composite baselines in the project. To use pure composite baselines, create components without a VOB root directory. For more information on pure composite baselines, see "Pure composite baselines" on page 47.

You can also use a pure composite baseline to represent the project configuration. Use one top-level pure composite baseline that selects the baselines of all components in the project, either directly or indirectly through other composite baselines. Using a pure composite baseline to represent the project is easier than keeping track of a set of baselines, one for each component. For more information on identifying a project baseline, see "Identifying a project baseline" on page 46.

After you create components without a VOB root directory, you can create components for storing elements (see "Creating components for storing elements" on page 88).

To create a component without a VOB root directory

1. Start the Project Explorer (see "To start Project Explorer" on page 88).

2. The left pane of the Project Explorer lists folders for all PVOBs in the local Rational ClearCase domain. Each PVOB has its own root folder. The root folder is created using the name of the PVOB. Navigate to the PVOB that you created.
3. Locate a folder called **Components**, which contains entries for each component in the PVOB. Right-click the **Components** folder and click **New > Component Without a VOB**.
4. In the Create Component Without a VOB window, enter a name and description for the component. Click **OK**.

To start Project Explorer

The Project Explorer is the graphical user interface (GUI) through which you create, manage, and view information about projects.

On a Windows system, do one of the following:

- In the shortcut pane of Rational ClearCase Explorer, click **UCM** and then click **Project Explorer**.
- Click **Start > Programs > IBM Rational > IBM Rational ClearCase > Project Explorer**.

On Linux and the UNIX system, at a shell prompt, enter **clearprojexp**.

Creating components for storing elements

You create ordinary components for storing the files that your team develops and the directories in which those files are cataloged.

Product Note: The process for creating components that store elements is slightly different for Rational ClearCase and Rational ClearCase LT.

When you create an ordinary component, you must specify the VOB that stores the component directory tree. You can store multiple components in a VOB, or you can create a VOB that stores one component. See "Deciding how many VOBs to use" on page 31 for details about using one VOB to store multiple components.

When you create an ordinary component, it includes an initial baseline with a name in the following format:

*component-name*_INITIAL

This baseline selects the /main/0 version of the root directory of the component. It serves as the starting point for successive baselines of the component.

To create a multiple-component VOB (Windows)

1. Start the VOB Creation Wizard (see "To start the VOB Creation Wizard (the Windows system)" on page 86).
2. Enter a name for the VOB. Enter a comment to describe the purpose of the VOB. Set **This VOB will contain UCM components**.
3. Set **Allow this VOB to contain multiple components** and **Seed the VOB with these components**. Select a view from the **View** list, and click **Add**.

   In the Add Component window, enter the component name and root directory, and click **OK**. The component appears in the list in the wizard. Click **Add** to create additional components. The component name must be unique within its PVOB.

4. Specify where to store the VOB. You can choose one of the recommended locations or enter the UNC path of a different location. Click **Browse** to search the network for shared resource locations.
5. Identify the PVOB that will store the project information about the components. Click the arrow to see the list of available PVOBs. Select the PVOB that you previously created (see "Creating the project VOB" on page 86).

To create a multiple-component VOB in Rational ClearCase LT (Windows)

On the Rational ClearCase LT server host, when the VOB Creation Wizard runs (see "To start the VOB Creation Wizard (the Windows system)" on page 86), follow these steps:

1. Enter a name for the VOB. Enter a comment to describe the purpose of the VOB.
2. Set **Allow this VOB to contain multiple components** and **Seed the VOB with these components**. Select a view from the **View** list, and click **Add**.

   Enter the component name and root directory in the Add Component window, and click **OK**. The component appears in the list in the wizard. Click **Add** to create additional components. The component name must be unique within its PVOB.
3. Specify where to store the VOB. This page of the wizard lists the VOB storage locations created by your Rational ClearCase administrator. If only one VOB storage location exists, the VOB Creation Wizard skips this step and uses that VOB storage location.

To create a multiple-component VOB (Linux and the UNIX system)

1. Use the **cleartool mkvob** command. For example:

   ```
   cleartool mkvob –nc –tag /vobs/testvob13 /usr/vobstore/testvob13.vbs
   ```

   If you are in an MVFS environment and developers use dynamic views, perform the following two steps.
2. Create the VOB mount point to match the VOB tag. For example:

   ```
   mkdir /vobs/testvob13
   ```
3. Mount the VOB. For example:

   ```
   cleartool mount /vobs/testvob13
   ```

To create a multiple-component VOB in Rational ClearCase LT (Linux and the UNIX system)

On the Rational ClearCase LT server host, use the **cleartool mkvob** command. For example:

```
cleartool mkvob –nc -tag /testvob13 -stgloc vobstore
```

To create a component and store it in the VOB

1. In Rational ClearCase Project Explorer, right-click the **Components** folder and click **New > Component in a VOB**.
2. In the Create a Component in a VOB window, from the **VOB** list, select the VOB that will contain the component.
3. Enter a name for the component and the component root directory. Click **OK**.

To create one component per VOB (Windows)

1. Start the VOB Creation Wizard (see "To start the VOB Creation Wizard (the Windows system)" on page 86).

2. Enter a name for the component. The component name must be unique within its PVOB. Enter a comment to describe the purpose of the component. Set **This VOB will contain UCM components**.
3. Set **Create VOB as a single VOB-level component**.
4. Specify where to store the component. You can choose one of the recommended locations or enter the UNC path of a different location. Click **Browse** to search the network for shared resource locations.
5. Identify the PVOB that will store the project information about the component. Click the arrow to see the list of available PVOBs. Select the PVOB that you previously created (see "Creating the project VOB" on page 86).

   The component is created with an initial baseline that points to the `\main\0` version of the component root directory.

To create a VOB and one component in Rational ClearCase LT (Windows)

1. On the Rational ClearCase LT server host, click **Start > Programs > IBM Rational > IBM Rational ClearCase LT > ClearCase Create VOB**. The VOB Creation Wizard appears.
2. Enter a name for the component. The component name must be unique within its PVOB. Enter a comment to describe the purpose of the component.
3. Set **Create VOB as a single VOB-level component**.
4. Select one of the available storage locations for the VOB storage directory. This page of the wizard lists the VOB storage locations created by your Rational ClearCase administrator. If only one VOB storage location exists, the VOB Creation Wizard skips this step and uses that VOB storage location.

To create one component per VOB (Linux and the UNIX system)

1. Make a view by using the **cleartool mkview** command. For a dynamic view, also issue the **cleartool setview** command. For example:

   ```
   cleartool mkview –tag myview /net/host2/view_store/myview.vws
   cleartool setview myview
   ```

2. Use the **cleartool mkvob** command to create a VOB. For example:

   ```
   cleartool mkvob –nc –tag /vobs/testvob1 /usr/vobstore/testvob1.vbs
   ```

3. If you are in an MVFS environment and developers use dynamic views, perform the following steps.
   a. Create the VOB mount point to match the VOB tag. For example:

      ```
      mkdir /vobs/testvob1
      ```

   b. Mount the VOB. For example:

      ```
      cleartool mount /vobs/testvob1
      ```

4. Do one of the following steps to create the component:
   - Issue the **cleartool mkcomp** command. For example:

     ```
     cleartool mkcomp –nc –root /vobs/testvob1 testcomp1@/vobs/myproj2_pvob
     ```

     In this example, the **mkcomp** command creates a component named **testcomp1** based on the VOB named **testvob1**. Although this example uses different names for the VOB and component, you can use the same name for both. The component name must be unique within its PVOB. The VOB and PVOB must be mounted before you issue the command. All projects that use the **myproj2_pvob** PVOB can access the **testcomp1** component.
   - Convert an existing VOB into a component by using the Rational ClearCase Project Explorer. See "To make a VOB into a component" on page 97 for more information.

To create a component in Rational ClearCase LT (Linux and the UNIX system)

1. Make a view and change to the directory. For example:

```
cleartool mkview –stgloc dev_views ~/chris_snap_view
cd ~/chris_snap_view
```

2. Use the **cleartool mkvob** command to create a VOB. For example:

```
cleartool mkvob –nc –tag /testvob1 –stgloc vobstore
```

3. Issue the **cleartool mkcomp** command. For example:

```
cleartool mkcomp –nc –root testvob1 testcomp1@/myproj2_pvob
```

Creating the project

You can create a project by using the Project Explorer and the New Project Wizard. For information on creating a project from the command-line interface (CLI), see the **cleartool mkproject**, **mkstream**, and **mkfolder** reference pages.

To create a project

1. Start Project Explorer (see "To start Project Explorer" on page 88).
2. The left pane of the Project Explorer lists root folders for all PVOBs in the local Rational ClearCase domain.

   Product Note: On the Rational ClearCase LT server, there is only one PVOB.

   Each PVOB has its own root folder. The root folder is created using the name of the PVOB. The folder **Components** contains entries for each component in the PVOB. Folders can contain projects and other folders. Select the root folder for the PVOB that you want to use for storing project information.
3. Click **File > New > Folder** to create a project folder. You do not need to create a project folder, but it is a good idea. As the number of projects grows, project folders are helpful in organizing related projects.
4. In the left pane, select the project folder or root folder. Click **File > New > Project**. The New Project Wizard appears.
5. In the New Project Wizard, enter a descriptive name for the project and provide a comment to describe the purpose of this project.
   - Enter a name for the project integration stream or accept the default name (*project-name*_**Integration**).
   - Select the type of project to create. A traditional parallel development project lets users create multiple streams so that developers can have private and shared work areas. A single-stream project contains only one stream, the integration stream. Users cannot create development streams in a single-stream project. See "Choosing a stream strategy" on page 34 for information about single-stream and multiple-stream projects.
6. Indicate whether you want to create the project based on an existing project. Because you are creating a project from scratch, click **No**.
7. Choose the baselines that the project will use. These baselines are either the foundation baselines upon which all work within the project is built or the baselines from which other projects start.
   a. Click **Add** to open the Add Baseline window. In the **Component** list, select one of the components that you previously created.
      - On Windows systems, click **Change > All Streams**.
      - On Linux and the UNIX system, click the arrow at the end of the **From Stream** field and set **All Streams**.

      The component initial baseline appears in the **Baselines** list.

b. Select the baseline.
c. Set **Allow project to modify the component** unless you want the component to be read-only. (See "Identifying read-only components" on page 33 for information on when you may want to use read-only components.)
d. Click **OK**. The baseline now appears in the list. Continue to use the Add Baseline window until the project contains its full set of foundation baselines, including the baseline for the component that stores the project composite baseline.

8. Specify the development policies to enforce for this project. Set the check boxes for the policies that you want to enforce. See Chapter 4, "Setting policies," on page 63 for information about each policy.
9. Indicate whether to configure the project to work with the UCM integration with Rational ClearQuest. To enable the project to work with Rational ClearQuest, click **Yes, use the following ClearQuest connection** and, in **ClearQuest Link**, select the database set (connection) and Rational ClearQuest user database that you have set up to use with this project. You are asked to authenticate with your Rational ClearQuest user name and password. See "Enabling use of the UCM integration with Rational ClearQuest" on page 100 for details about the integration.

   You can click **Policies** to set UCM policies related to the integration or access the Policies page of the project to set them after the project is created.

Setting a baseline naming template

UCM lets you define a template for implementing a baseline naming convention within a project. The template can include any of the following tokens:

- Project
- Component
- Stream
- Date
- Time
- User
- Host
- Basename

Basename refers to a name that you specify.

If you do not specify a baseline naming template, the basename is used to name new baselines. When necessary, a numeric identifier is appended to the baseline name to make it unique.

During deliver operations, a baseline is created in the source stream. When naming this baseline, the following format is used in place of the basename token:

deliverbl.*source-stream-name.date.unique-identifier*

For information about using a baseline naming convention, see "Defining a baseline naming convention" on page 52.

To set a baseline naming template: Use the **–blname_template** option with the **cleartool mkproject** or **chproject** command to set a template. For example:

```
cleartool chproject -blname_template project,component,date mck_proj1
```

This example sets a template that uses the project name, component name, and date in all baseline names created in the **mck_proj1** project. Use commas to separate the tokens in the command-line entry. When you create baselines, the

commas are replaced with underscores. See *IBM Rational ClearCase Command Reference* for details about using **chproject** and **mkproject**.

Defining promotion levels

Five baseline promotion levels are provided. You can keep some or all of them, and you can define your own promotion levels. Use the Project Explorer to define the promotion levels. For information on promotion levels, see "Identifying promotion levels to reflect state of development" on page 52.

To define the promotion levels that your project uses:

1. In the Project Explorer, select the PVOB root folder that contains your project, and then click **Tools > Define Promotion Level**. All projects that use that PVOB have access to the same set of promotion levels.
2. In the Define Promotion Levels window, to remove an existing promotion level, select it and click **Remove**. To change the order of promotion levels, select a promotion level and use the **Move Up** or **Move Down** buttons.
3. To add a new promotion level, click **Add**. The Add Promotion Level window is displayed. Enter the name of the new promotion level and click **OK**. The new promotion level appears in the list of promotion levels in the Define Promotion Levels window. Move it to the desired place in the order.
4. When you finalize the set and order of promotion levels, select one to be the initial promotion level for new baselines. The initial promotion level is the level assigned by default when you create a baseline.

For information on defining promotion levels from the CLI, see the **cleartool setplevel** reference page.

Creating an integration view

When you create a project, the project integration stream is created for you. To see elements in the project and make changes to the project shared elements, you need an *integration view* which is attached to the project integration stream. Use the Project Explorer to create an integration view. The following kinds of views are supported:

- Dynamic views, which use the multiversion file system (MVFS) to provide immediate, transparent access to files and directories stored in VOBs. On Windows systems, a dynamic view is mapped to a drive letter in Windows Explorer.
- Snapshot views, which copy files and directories from VOBs to a directory on your computer.

  Product Note: Rational ClearCase LT supports only snapshot views.

If the integration view is a dynamic view, you ensure that you always see the correct version of files and directories that developers deliver to the integration stream. With a snapshot view, you have to perform an *update* operation to copy the latest delivered files and directories to your computer. For more information about dynamic and snapshot views, see *Developing Software* online help.

To create an integration view

1. In the Project Explorer, navigate to the integration stream by moving down the object hierarchy:
   a. Root folder
   b. Project folder
   c. Project

d. Stream

2. Select the integration stream and click **File > New > View**.

   On the Windows system, the View Creation Wizard is displayed. On Linux and the UNIX system, the Create View window is displayed.

3. Accept the default values to create an integration view attached to the integration stream. By default, the View Creation Wizard and the Create View window use this convention for the integration view name:

   *username_project-name*_int

Creating and setting an activity in the integration stream (Linux and the UNIX system only)

Before you can add elements to the integration stream, you need to create and set an activity.

To create and set an activity (Linux and the UNIX system)

1. Set your integration view if it is a dynamic view. For example:

   ```
   cleartool setview kmt_Integration
   ```

   If your integration view is a snapshot view, change directory to it.

2. Issue the **cleartool mkactivity** command. For example:

   ```
   cleartool mkactivity –headline "Create Directories"
   create_directories
   ```

   The Rational ClearCase GUI tools use the name specified with **–headline** to identify the activity. The last argument, **create_directories**, is the activity-selector. Use the activity-selector when you issue **cleartool** commands.

3. By default, when you make an activity with the **cleartool mkactivity** command, your view is set to that activity. Your view is not set to an activity if you create multiple activities in the same command line or if you specify a stream with the **–in** option. If you need to set your integration view to the activity, use the **cleartool setactivity** command. For example:

   ```
   cleartool setactivity create_directories
   ```

Creating the directory structure

If you create the project from scratch, you need to create the directory elements within the project components. This action implements the directory structure that you define during the planning phase. See “Defining the directory structure” on page 33.

To add a directory element to a component (the Windows system)

1. In Windows Explorer, navigate to the integration view. Double-click the component to display its contents. If the component is in a VOB that you created to store multiple components, the component appears as a folder under the VOB.
2. Create a folder.
3. Right-click the folder and click **ClearCase > Add to Source Control**.
4. When prompted, specify an activity to be associated with the addition of the new directory element.

For more information about creating directory and file elements, see *Developing Software* online help and the **mkelem** reference page.

To add a directory element to a component (Linux and the UNIX system)

1. With your integration view set to an activity (see "Creating and setting an activity in the integration stream (Linux and the UNIX system only)" on page 94), navigate to the component. If the component is in a VOB that you created to store multiple components, the component appears as a directory under the VOB. For example:

   ```
   cd /vobs/testvob13/libs
   ```

2. Check out the component root directory. For example:

   ```
   cleartool co –nc .
   ```

3. Issue the **cleartool mkelem** command. For example:

   ```
   cleartool mkelem –nc –eltype directory design
   ```

   This example creates a directory element called **design**. By default, the **mkelem** command leaves the element checked out. To add elements, such as subdirectories, to the directory element, you must leave the directory element checked out.

4. When you finish adding elements to the new directory, check it in. For example:

   ```
   cleartool ci –nc design
   ```

5. When you finish creating directory elements, check in the component root directory. For example:

   ```
   cleartool ci –nc .
   ```

For more information about creating directory and file elements, see *Developing Software* online help and the **mkelem** reference page.

Importing directories and files from outside Rational ClearCase version control

If you have a large number of files and directories that you want to place under Rational ClearCase version control, you can speed the process by using the **clearexport** and **clearimport** command-line utilities. These utilities allow you to migrate an existing set of directories and files to a Rational ClearCase repository from another version control software system, such as SourceSafe, RCS, or PVCS. You have the following options:

- Migrate source files directly into a component (see "To migrate source files into a component" on page 95).
- Use **clearexport** and **clearimport** on VOBs, and then convert the VOBs to components. For details on converting VOBs into components, see "Creating a project based on an existing Rational ClearCase configuration" on page 97.
- Migrate directories and flat files that are not currently under any version control.

  Use the **clearfsimport** command-line utility. Run **clearfsimport** from within a UCM view to import directories and files directly onto a stream. You can then create a baseline in the stream without having to label the versions. See the **clearfsimport** reference page for details.
- On Windows systems, use the Import Wizard, a graphic user interface (GUI) that you can use as an alternative to the **clearexport** and **clearimport** commands.

For details on using **clearexport** and **clearimport**, see the *IBM Rational ClearCase Administrator's Guide* and the **clearexport** and **clearimport** reference pages.

To migrate source files into a component

1. Run **clearexport** to generate a data file from your source files.

2. Create and set a non-UCM view. On Windows systems, use the View Creation Wizard. To start the View Creation Wizard, from Rational ClearCase Explorer click **Base ClearCase > Create View**. On Linux and the UNIX system, use the **cleartool mkview** and **setview** commands.
3. In the view context, run **clearimport** to populate the component with the files and directories from the data file.
4. In the component, create a baseline from a labeled set of versions. If the versions that you want to include in the baseline are not labeled, create a label type and apply it to the versions.

Making baselines of newly populated components

After you create the directory structure and import files (see "Creating the directory structure" on page 94 and "Importing directories and files from outside Rational ClearCase version control" on page 95), create new baselines that select those directory and file elements for each of the components to which you added elements. For more information, see "Creating a new baseline" on page 114.

If you use pure composite baselines, use these new baselines to create the dependency relationships for the composite baselines that you want to be consumed in the project. For more information, see "Creating the dependency relationships for composite baselines in the project."

Creating the dependency relationships for composite baselines in the project

In the components that you created without a VOB root directory (see "Creating components for storing baseline dependencies" on page 87), add as member baselines the newly created baselines for the components in your project that group files and directories. This creates composite baselines for the project. When a member baseline is added to a component, a dependency reference is added to that component and a new composite baseline is made. If you are making pure composite baselines that select other pure composite baselines, start making the composite baselines at the lowest level in the hierarchy that you are defining.

A sound strategy is to maintain the pure composite baselines in a bootstrap project.

You can create a pure composite baseline that represents the project by selecting the lower level pure composite baselines or the latest baseline from each component that the project will use.

To create a composite baseline

1. In the Project Explorer, right-click the project integration stream or feature-specific development stream and click **Edit Baseline Dependencies**.
2. The Edit Baseline Dependencies window displays a list of all components that the project uses. Identify the component that you created without a VOB root directory and to which you will add the member baselines (see "To create a component without a VOB root directory" on page 87). Drag the other components onto the component that will contain the baseline dependencies.
3. Click **OK**. The Create Baseline Dependencies window is displayed.
4. Enter the name in the **Base Name** (Windows) or **Baseline Title** (Linux and the UNIX system) field that you want to use for the baselines that UCM creates for these components. If you have a baseline naming template set for the project, the **Template Name** field shows the name that will be used. If the template

does not include the Basename token, or if no template is set, the **Base Name** or **Baseline Title** field does not appear in the Create Baseline Dependencies window.

5. Click **OK**.

Recommending a baseline for new components

Recommend a new baseline that selects those directory and file elements to be used in the project. If you use pure composite baselines, this can be one baseline that you want all developers to use. Developers who join the project at the project level or at the feature-specific development stream populate their development streams with the versions that are identified by the recommended baseline. For more information, see "Recommending the baseline" on page 118.

Creating a project based on an existing Rational ClearCase configuration

If you have existing VOBs, you can convert them or their directories into components so that you can include them in projects. You can set up a project based on existing VOBs.

Creating the PVOB from an existing Rational ClearCase configuration

On the Windows system, use the VOB Creation Wizard to create the PVOB (see "To create a PVOB (the Windows system)" on page 86). In Step 3, if you currently use an administrative VOB, select it in the list. An **AdminVOB** hyperlink is made between the PVOB and the administrative VOB. When you create components, they use the existing administrative VOB. If you do not currently use an administrative VOB, select **none**.

On Linux and the UNIX system, use the **cleartool mkvob** command (see "To create a PVOB (Linux and the UNIX system)" on page 86). If you currently use an administrative VOB, use the **cleartool mkhlink** command to create an **AdminVOB** hyperlink between the PVOB and the administrative VOB. When you create components, they then use the existing administrative VOB.

If the project uses pure composite baselines, create components without a VOB root directory. For more information, see "Creating components for storing baseline dependencies" on page 87.

Making components from existing VOBs

You can do any of the following:

- Make a VOB into a component
- Make a directory in a VOB into a component

You may want to organize the contents of a VOB into multiple components.

To make a VOB into a component

1. In the Project Explorer, select the PVOB. Do one of the following:
   - On the Windows system, click **Tools > Import > VOB as Component**.
   - On Linux and the UNIX system, click **Tools > Import > Import VOB**.

   The Import VOB window is displayed.

2. In the **Available VOBs** list, select the VOB that you want to make into a component. Click **Add** to move the VOB to the **VOBs to Import** list. You can add more VOBs to the **VOBs to Import** list. If you change your mind, you can select a VOB in the **VOBs to Import** list and click **Remove** to move it back to the **Available VOBs** list.
3. When you are finished, click **Import**.

To make a directory tree within a VOB into a component

1. In the Project Explorer, right-click the PVOB folder and click **Import > VOB Directory as Component**.
2. In the Import VOB Directory as Component window, select a view from the **View** list; select the VOB that contains the directory from the **VOB** list; select the directory from the **Root Directory** list; and specify a name for the component.

   The new component contains the directory and all its subdirectories and files. The component root directory must be at or directly below the VOB root directory. If the component root directory is at the VOB root directory, that VOB cannot store multiple components.

Making a baseline from a label

After you convert an existing VOB or one of its directory trees into a component, to access the directories and files in that component, you must create a baseline from the set of versions identified by a label type.

To create a baseline by label type

1. Create and apply a label type.

   On the Windows system, if the set of versions that you want to use are not already labeled, use the Apply Label Wizard to make and apply a label type. To start the Apply Label Wizard, do one of the following steps:

   - Click **Start > Programs > IBM Rational > IBM Rational ClearCase > Apply Label Wizard**
   - Enter **clearapplywizard** at the command prompt

   On Linux and the UNIX system, if the set of versions that you want to use are not already labeled, use the **cleartool mklbtype** and **mklabel** commands to create and apply a label type. For example:

   ```
   % cleartool mklbtype –c "label for release 2" REL2
   Created label type "REL2".

   % cleartool mklabel -recurse REL2 .
   Created label "REL2" on "." version "/main/5".
   Created label "REL2" on "./src" version "/main/6".
   Created lable "REL2" on "./src/Makefile" version "/main/2".
   ```

   The **–recurse** option causes the label to be applied to all versions at or below the current working directory.
2. In the Project Explorer, select the PVOB. Do one of the following:

   On Linux and the UNIX system, click **Tools > Import > Import Label**. Step 1 of the Import Label Wizard appears.

   On the Windows system, click **Tools > Import > Label as Baseline**.
3. In the **Available Components** list, select the component that contains the label from which you want to create a baseline. Click **Add** to move that component to the **Selected Components** list. If you change your mind, select a component in the **Selected Components** list and click **Remove** to move the component back to the **Available Components** list.

4. In Step 2 on page 98, select the label type that you want to import, and enter the name of the baseline that you want to create for the versions identified by that label type. Then select the baseline promotion level.

   Note: You cannot import a label type from a global label type definition.

Creating the project

Use the New Project Wizard to create the project. For more information, see "Creating the project" on page 91.

Finishing the project configuration

To finish configuring the project that is based on an existing Rational ClearCase configuration, perform the following operations:

- Create an integration view. For more information, see "Creating an integration view" on page 93.
- If the project is using pure composite baselines, create the dependency relationships for those baselines. For more information, see "Creating the dependency relationships for composite baselines in the project" on page 96.
- Recommend a baseline that developers use in the project. For more information, see "Recommending a baseline for new components" on page 97.

Creating a project based on an existing project

As you create new projects, you may need to create new instances of existing projects. For example, suppose you have released version 3.0 of the Webotrans project and are planning for version 3.1. You anticipate that version 3.1 will use the same components as version 3.0. Therefore, you want to use the latest baselines in the version 3.0 components as the foundation baselines for version 3.1 development. A good strategy is to use a bootstrap project. See "Bootstrap projects" on page 146.

Capturing final baselines in a composite baseline

If the existing project contains numerous components, you may want to create a pure composite baseline that selects the final baselines of those components before you create the new project. This composite baseline serves as a single starting point for teams that want to start their work from the final approved baselines of the existing project.

To create a pure composite baseline from existing approved baselines

1. Create a component that does not have a root directory in a VOB. See "Creating components for storing baseline dependencies" on page 87.
2. Add the initial baseline of the component to the integration stream.
3. In the component, create a composite baseline that selects baselines of all other components in the project. See "Creating the dependency relationships for composite baselines in the project" on page 96.
4. Recommend the composite baseline. See "Recommending the baseline" on page 118.

Creating the project from another project

If your project is a new instance of an existing project and uses the same components as the existing project, do not create a new PVOB for this project.

Continue to use the existing PVOB. You can create the project based on the existing PVOB (see "To create a project based on an existing project" on page 100).

To create a project based on an existing project

1. Start the New Project Wizard to create the project (see "Creating the project" on page 91).
2. In Step 2 of the wizard, set **Yes** to indicate that the project begins from the baselines in an existing project. Then navigate to the project that contains those baselines. For example, the new project is based on the baselines in the **OM_proj1.0_Integration** stream.
3. Step 3 lists the latest baselines in the project that you select in Step 2. If you created a pure composite baseline to capture the final approved baselines in the existing project, select it. You can add baselines from components that are not part of the existing project by clicking **Add** to open the Add Baseline window. Similarly, you can remove a baseline by selecting it and clicking **Remove**.
4. Finish the remaining steps in the wizard (see "Creating the project" on page 91).

Creating an integration view

When you create a new project, a new integration stream is created for you. Therefore, you need to create a new integration view to access elements in the integration stream. Create an integration view as described in "Creating an integration view" on page 93.

Enabling use of the UCM integration with Rational ClearQuest

Before you can connect a project to a Rational ClearQuest user database, you must set up the database to use a UCM-enabled schema and have the required credentials to access the user database. See Chapter 5, "Setting up a Rational ClearQuest user database for UCM," on page 75. After you set up the Rational ClearQuest user database, you can enable the project for use with a Rational ClearQuest user database.

To enable a project to work with a Rational ClearQuest user database

For a current project:

1. In the left pane of the Project Explorer, right-click the project and click **Properties** to display its property sheet.
2. Click the **ClearQuest** tab and then set **Project is ClearQuest-enabled**.

   On Windows systems, **Link to the ClearQuest User Database** is seeded with a name. The Rational ClearQuest Schema Repository window appears with the connection corresponding to that user database selected.
3. Select the user database that you want to link to the project. The first time that you enable a project, the Rational ClearQuest Login window is displayed.
4. Enter your credentials (user name, password, and the name of the Rational ClearQuest user database to which you are linking the project). For information on credentials, see "Creating users and adding credentials" on page 82. Click **Next**. And click **OK**.

5. To set the Rational ClearQuest development policies that you want to enforce, click **Policies** and **ClearQuest**. See "Policies for the UCM integration with Rational ClearQuest" on page 69 for a description of these policies. Click **OK** when you are finished.

For a new project:

If you are creating a new project, you can enable the project to work with Rational ClearQuest by setting **Yes, use the following ClearQuest connections** and selecting the connection and user database in Step 5 of the New Project Wizard. For information about the procedure to create a new project, see "Creating the project" on page 91.

Changing the project to a different Rational ClearQuest user database

After you enable a UCM project to work with a Rational ClearQuest user database, you may decide to link the project to a different user database. If no activities have been created, you can switch databases by selecting a different one on the **ClearQuest** tab of the project property sheet.

Migrating activities

If your project contains activities when you enable it to work with a Rational ClearQuest user database, the UCM integration with Rational ClearQuest creates records for each of those activities by using the **UCMUtilityActivity** record type. To store all of your project activities in records of some other record type, enable the project when you create it, before team members create any activities. After the migration is complete, any new activities that you create can link to records of any UCM-enabled record type.

Setting project policies

A UCM-enabled schema includes policies that you can enable from either Rational ClearCase or Rational ClearQuest control.

To set policies in Rational ClearCase control

On the **ClearQuest** page of the project Policies window, set the check boxes next to the policies to enable them. Clear the check box to disable the related policy.

To start a Rational ClearQuest client

1. Do one of the following:
   - To start the Rational ClearQuest Web client, from your internet browser, use the URL that your project manager supplies. The Rational ClearQuest Web server displays a Login window.
   - To start the Rational ClearQuest for Windows client, on the Windows system, click **Start > Programs > IBM Rational > IBM Rational ClearQuest > Rational ClearQuest**. The client displays a Login window.
2. Enter the credentials that are registered on your system. For more information about credentials, see "Creating users and adding credentials" on page 82.

To set policies from the Rational ClearQuest client

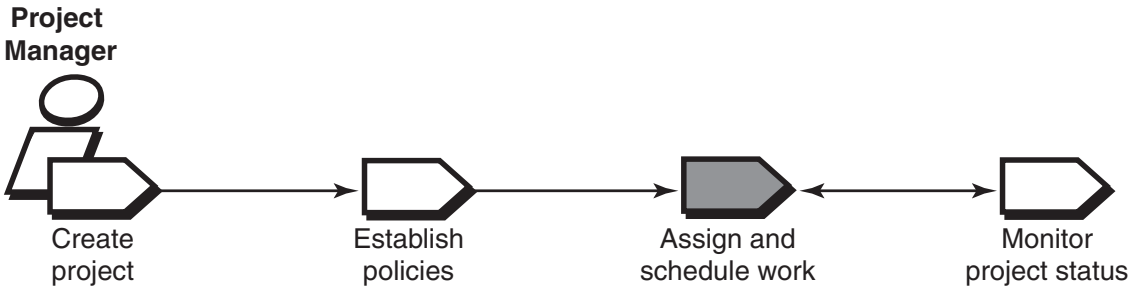
1. Run the Rational ClearQuest client (see "To start a Rational ClearQuest client" on page 101).
2. In the Rational ClearQuest client workspace, navigate to and double-click the **UCMProjects** query.

The query displays all UCM-enabled projects that are associated with the current Rational ClearQuest user database.

3. Select a project from the **Results** set. The project form appears.
4. On the form, click **Actions** and select **Modify**. Set the check boxes for the policies to be enabled.

Because not all policies are stored in the Rational ClearQuest user database for a project, you may have to set some policies from Rational ClearCase control. For descriptions of the policies, see "Policies for the UCM integration with Rational ClearQuest" on page 69.

Assigning activities



After you establish policies, you create activities so that work can be assigned and scheduled.

To create and assign activities in a Rational ClearQuest user database

1. Start the Rational ClearQuest client (see "To start a Rational ClearQuest client" on page 101).
2. Log on to the user database connected to the project.
3. Click **Actions > New**. The Choose a record type window is displayed. Select a UCM-enabled record type, and click **OK**.
4. The **Submit** form appears. Fill in the boxes on each tab. When you finish filling in the boxes, click **OK**. The record is created and placed in a **Submitted** type state.
5. Run a query and select the record. For example, if the record type is **Defect**, you can run the **All Defects** query.
6. Click **Actions > Assign**, and select the owner from the **Owner** list. Click **Apply**.

User account profiles must exist in a Rational ClearQuest user database for the developers to whom you assign activities. See "Creating users and adding credentials" on page 82 for information about creating user account profiles.

Disabling the link between a project and a Rational ClearQuest user database

There may be times when you want to disable the link between a project and a Rational ClearQuest user database.

To disable the project and user database link

1. On the **ClearQuest** tab of the project property sheet, clear **Project is ClearQuest-enabled**.
2. Click **OK** on the **ClearQuest** tab. The integration disables the link between the project and the Rational ClearQuest user database. The integration also removes any existing links between activities and their corresponding Rational ClearQuest records.
3. Display the project property sheet again, set **Project is ClearQuest-enabled**, and select another user database if you want to link the project to a different user database.

   Tip: If you select the same user database that you just unlinked, the integration creates new Rational ClearQuest records for the project activities; it does not link the activities to the Rational ClearQuest records with which they were previously linked.

Fixing projects that contain linked and unlinked activities

After you enable a project to work with Rational ClearQuest, some of the project activities can remain unlinked to Rational ClearQuest records. Similarly, when you disable the link between a project and a Rational ClearQuest user database, some activities may remain linked. The following scenarios can cause your project to be in this inconsistent state:

- A network failure or a general system crash occurs during the enabling or disabling operation and interrupts the activity migration.
- The Rational ClearQuest user database can become corrupted, forcing you to restore a backed-up version of the user database. That version of the user database is out of sync with the PVOB that contains the project that is linked to the user database.
- You use the **cleartool** command-line interface to rename an activity or a project. When you rename an activity or project from the command-line interface, the UCM integration with Rational ClearQuest does not update the corresponding user database record with the name change. As a result, the Rational ClearCase and Rational ClearQuest objects are not synchronized.
- The project PVOB is in a Rational ClearCase MultiSite configuration, and unlinked activities were added by a Rational ClearCase MultiSite synchronization operation to the local PVOB project, which is enabled to work with Rational ClearQuest.

Detecting unlinked activities

If a developer attempts to take an action, such as modifying an unlinked activity in an enabled project, the integration displays an error and disallows the action.

Correcting unlinked activities

If the problem is the result of one of the most likely scenarios (see "Fixing projects that contain linked and unlinked activities" on page 103), use the **cleartool checkvob** command with the **–ucm** option to restore the project to a consistent state. See *IBM Rational ClearCase Administrator's Guide* and *IBM Rational ClearCase Command Reference* for details about using this command.

If the problem is caused by Rational ClearCase MultiSite, at the remote site, link the unlinked activities (see "To link unlinked activities at a remote site" on page 103).

To link unlinked activities at a remote site:

1. In the Project Explorer, display the project property sheet, and click the **ClearQuest** tab.
2. Click **Link all unlinked activities mastered at this replica**. The integration checks all of the project activities and links any that are unlinked. The integration then displays the following summary information:
   - Number of activities that had to be linked.
   - Number of activities that were previously linked.
   - Number of activities that could not be linked because they are not mastered in the current PVOB replica. In this case, the integration also displays a list of replicas on which you must run the **Link all unlinked activities mastered at this replica** operation again to correct the problem.
3. At each replica on the list described in Step 2, repeat Step 1 and Step 2.

How the UCM integration with Rational ClearQuest is affected by Rational ClearQuest MultiSite

If you use Rational ClearCase MultiSite to replicate the PVOB and ClearQuest MultiSite to replicate the Rational ClearQuest user database and schema repository involved in the UCM integration with Rational ClearQuest, you need to be aware of several requirements.

Replica and naming requirements

When you set up the UCM integration with Rational ClearQuest, you establish a link between a project and a Rational ClearQuest user database. If you use Rational ClearCase MultiSite, the following requirements apply:

- Each site that has a PVOB replica that contains a linked project must have a replica of the Rational ClearQuest user database to which the project is linked and the user database schema repository.
- Similarly, each site that contains a linked Rational ClearQuest user database replica must contain a replica of the PVOB that contains the project to which the user database is linked.
- The name of the Rational ClearQuest replica must match the name of the PVOB replica at the same site.

Transferring mastership of the project

Before you enable a project to work with Rational ClearQuest, your current PVOB replica must master the project. If your replica does not master the project, transfer mastership of the project by using the **multitool chmaster** command at the replica that masters the project.

When you enable the project to work with Rational ClearQuest, the UCM integration with Rational ClearQuest creates a corresponding project record in the Rational ClearQuest user database (see "Mapping PVOBs to Rational ClearQuest user databases" on page 57). The integration assigns mastership of that record to the current replica of the Rational ClearQuest user database. If a project record with the same name as the project exists in the Rational ClearQuest user database when you enable the project, and that project record is not mastered by your current replica, you must transfer mastership of the project record to your current replica.

Linking activities to Rational ClearQuest records

If a project contains activities, when you enable that project to work with Rational ClearQuest, the UCM integration with Rational ClearQuest creates corresponding Rational ClearQuest records for the activities and links the records to the activities. The integration cannot link activities that are mastered by remote replicas. See "Correcting unlinked activities" on page 103 for information about linking activities that are mastered by a remote replica.

Changing project policy settings

Before you can change a project policy setting from within Rational ClearQuest control, the Rational ClearQuest project record must be mastered. Similarly, before you can change a project policy settings from within Rational ClearCase control, the project object must be mastered. After you change a project policy setting in the current replica, the new settings do not take effect in streams in sibling replicas until you synchronize the current replica with those replicas. See the *IBM Rational ClearCase MultiSite Administrator's Guide* for information about synchronizing replicas.

Changing the project name

The integration links a project name to the name field in the corresponding Rational ClearQuest project record (see "Naming projects that are linked to same user database" on page 58). If you change the project name in the Rational ClearCase graphic user interface (GUI), the integration makes the same change to the name field in the corresponding Rational ClearQuest project record. Similarly, if you change the name in the Rational ClearQuest user database, the integration makes the same change to the project name in the Rational ClearCase repository. Before you can change the project name in a Rational ClearCase MultiSite environment, the project record and the project object must both be mastered.

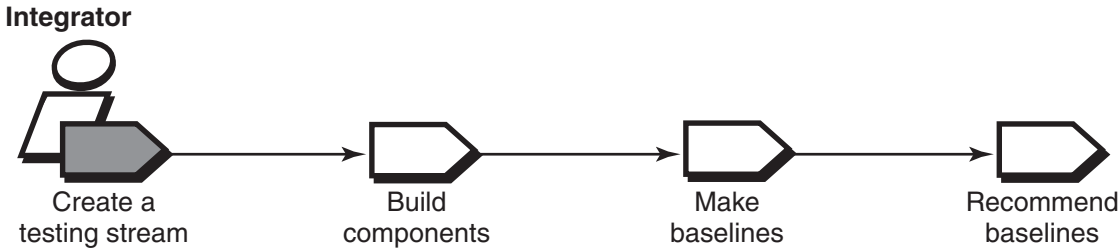
Restriction: Change the project name only by using a GUI, such as Project Explorer. If you change the project name by using the command-line interface, the integration does not make the same change to the corresponding project record.

Note also that the project name cannot be the same as a folder name or a stream name.

Working with IBM Rational Suite (Windows)

If you are using UCM with IBM Rational Suite, you can store Rational RequisitePro projects, Rational Rose and XDE™ models, and Rational Test datastores in UCM components and include them in baselines. To enable this integration, use the Rational Administrator GUI to create and configure a Rational project. A Rational project associates your UCM project with a RequisitePro® project, Rose models, and Rational Test datastores.

Creating a development stream for testing baselines



When you make a new baseline, lock the integration stream so that you can build and test a static set of files. Otherwise, developers can inadvertently cause confusion by delivering changes while you are building and testing. Locking the integration stream for a short period of time is acceptable; locking the integration stream for several days can result in a backlog of completed but undelivered activities.

To avoid locking out developers for a long period of time, you may want to create a development stream and use it for extensive testing of baselines (see "To create a development stream" on page 106). If your project uses feature-specific development streams, you may want to create a testing stream for each feature-specific development stream so that you can test the baselines created in those streams.

If the development stream is configured to be read-only, you can build and test the new baselines, and developers can deliver changes to the integration stream without being concerned about interfering with the building and testing process. For information on testing baselines, see "Testing the baseline" on page 116.

To create a development stream

1. In Project Explorer, right-click the integration stream, and click **Create Child Stream** from the pop-up menu. The Create a Development Stream window appears.
2. If you want to disallow changes to be made in the testing stream, set **Make Stream read only**. If you set this option, you cannot fix defects discovered in the baseline in this stream. Instead, the developers responsible for the defects would need to make the fixes in their development streams and deliver them to the feature-specific development stream.
3. By default, the set of *recommended baselines* is used when creating a development stream. Because the new baseline has not been tested extensively, you probably have not yet promoted it to the level associated with recommended baselines. To create the development stream with baselines other than the recommended baselines, click **Advanced Options**. The Change Baseline window appears.
4. Select the component that contains the baseline that you want to test. Click **Change**. A second Change Baseline window appears, listing all baselines for the component.
5. Select the baseline that you want to test, and click **OK**. If you need to test the baseline of another component, select it in the first Change Baseline window and repeat the process. When you are finished, click **OK** in the first Change Baseline window.

6. In the Create a Development Stream window, set **Prompt me to create a View for this stream**. Click **OK**.

   The View Creation Wizard (the Windows system) or Create View window (Linux and the UNIX system) appears.
7. Complete the steps of the View Creation Wizard or the fields of the Create View window to create a view for the development stream.

Creating a feature-specific development stream

Feature-specific development streams allow you to isolate work.

About creating feature-specific development streams

The basic UCM process uses the integration stream as the project sole shared work area. You may choose to organize your project into small teams of developers where each team develops a specific feature. This type of organization is supported by feature-specific development streams.

Create a development stream to serve as the shared work area for each team of developers. The developers who work on that feature create their own development streams based on the recommended baselines in the feature-specific development stream. See "Choosing a stream strategy" on page 34 for additional information about feature-specific development streams.

To create a feature-specific development stream

1. In Project Explorer, right-click the parent stream, and select **Create Child Stream** from the pop-up menu. The Create a Development Stream window appears.
2. By default, the set of recommended baselines is used when creating a development stream. To create the development stream with baselines other than the recommended baselines, click **Advanced Options** and select the baselines from the Change Baseline window.
3. In the Create a Development Stream window enter a name and description for the new stream. Set **Prompt me to create a View for this stream**. Click **OK**.

   On the Windows system, the View Creation Wizard is displayed.

   On Linux and the UNIX system, the Create View window is displayed.
4. Complete the steps of the View Creation Wizard or the Create View window to create a view for the development stream.
5. In Project Explorer, right-click the feature-specific development stream, and select **Recommend Baselines**.
6. In the Recommended Baselines window, click **Add** to display the Add Baseline window. Select the baselines that you want to recommend to developers who will work on this feature. When developers create their own development streams, those streams will be based on the recommended baselines. When you finish selecting the baselines, click **OK** in the Recommended Baselines window.

Chapter 7. Managing the UCM project

This chapter describes tasks involved in maintaining a UCM project.

About managing a project

After you create and set up a project, developers join the project, work on activities, and deliver completed activities to the integration stream or feature-specific development stream. In your role as integrator, you need to maintain the project so that developers do not get out of sync with each other's work. The following sections describe project maintenance tasks:

Adding components

Over time, the scope of your project typically broadens, and you may need to add components to a stream and to projects (see "To add a component to a stream" on page 110). Adding a component to a stream requires that you rebase to the baseline of the new component after the component is added.

By default, a component is added to the project as read-only. To allow developers to deliver changes for that component, make the component modifiable (see "To make a component modifiable within the project" on page 110).

Before you can access the component that you added to a stream from a view that is attached to the stream, you must synchronize the view with the new configuration (see "To synchronize a view with a new configuration" on page 110).

To enable a child stream to access a modifiable component that you added to a parent stream, you must do the following tasks:

- Synchronize the child stream with the new set of modifiable components in the project (see "To synchronize a child stream with project modifiable components" on page 110).
- Synchronize the child stream view with the new configuration of the parent stream (see "To synchronize a child stream view with new parent stream configuration" on page 111).

If snapshot views are attached to a stream to which you added a component, you need to edit the view load rules to include the components that you add to the stream (see "To edit the view load rules" on page 111). The load rules of a snapshot view specify which components are loaded into the view. In addition, you need to know whether any developers working on the project use snapshot views for their development views. When a developer who uses a snapshot view

rebases to a baseline that contains a new component, the snapshot view config spec is updated, but the view load rules are not updated. When you add a component, take the following actions for developers who use snapshot views:

- Notify the developers that they need to rebase their development streams to the baseline of the newly added component.
- Instruct the developers to update the load rules for their development views to load the newly added component.

To add a component to a stream

1. Start Project Explorer (see "To start Project Explorer" on page 88).
2. In the right pane of the Project Explorer, right-click the stream and click **Properties** to open the stream Properties window.
3. Click the **Configuration** tab, and then click **Add**. The Add Baseline window opens.
4. Do one of the following steps:
   - On Linux and the UNIX system, click the arrow at the end of the **From Stream** box and either select a stream from the tree hierarchy or click **All Streams**.
   - On the Windows system, click **Change > All Streams** or **Change > Browse** to select the stream that contains the component baseline you want to add.
5. In the **Component** list, select the component that you want to add. The component baselines appear in the Baselines list.
6. In the **Baselines** list, select the baseline that you want to add to the project.
7. Click **OK**. The Add Baseline window closes, and the baseline that you chose appears on the **Configuration** page.
8. Click **OK** to close the stream Properties window.

   The Rebase Stream Preview window opens. To modify the stream configuration to include the new foundation baseline, UCM needs to rebase the stream.
9. Click **OK** in the Rebase Stream Preview window.
10. Click **Complete** to finish the rebase operation.

To make a component modifiable within the project

1. In the Project Explorer, select the project, and click **File > Policies**.
2. In the **Components** tab, click the check box next to the component.
3. Click **OK**.

To synchronize a view with a new configuration

1. In the Project Explorer, select the stream that contains the component that you added, and click **File > Properties**.
2. Click the **Views** tab. Select the view and click **Properties**.
3. On the **General** tab, click **Synchronize with stream**.

To synchronize a child stream with project modifiable components

1. In the Project Explorer, select the child stream and click **File > Properties**.
2. On the **General** tab, click **Synchronize with project**.

To synchronize a child stream view with new parent stream configuration

1. In the Project Explorer, select the child stream and click **File > Properties**.
2. Click the **Views** tab. Select the view and click **Properties**.
3. On the **General** tab, click **Synchronize with stream**.

To edit the view load rules

1. In the Project Explorer, select the stream to which you added a component, and click **File > Properties** to display the stream property sheet.
2. In the property sheet, click the **Load Rules** tab.
3. Select the component or components that you added to the stream.
4. Click **Add**. Click **OK** to close the property sheet.

Element relocation

After you create components and add them to a UCM configuration, you should not change the configuration. Change flow and integration in UCM depend on stable components. If you absolutely must relocate directory and file elements, you can run a UCM-supplied script mkelem_cpver.pl. The script must be run in a strictly-controlled situation to prevent undesired configuration changes. You must use ratlperl to run the script. The script does the following operations:

- Copies directory and file elements (one source version to one target version) within a VOB or between VOBs. New elements are created in the target directory with the view's version of the original element from the source directory.
- Uncatalogs the elements in the source directory.
- Preserves the history of the source elements to allow work to continue in other streams.

The script does not do the following operations:

- Retain source history in the target version. (A comment attached to the target version records the path of the source version for historical purposes.)
- Roll back when errors are encountered. (No support is provided for undoing the operation. The project manager must fix any errors.)
- Integrate changed content in other streams. If you deliver or rebase content between streams, be careful to distinguish between the previous and current names of the relocated elements.

To relocate elements

1. In a shell or command prompt window, change directory to a view that is attached to a stream that contains the components whose elements you want to relocate.
2. If the view is a snapshot view, update it.
3. Set the view to an activity that you reserve for this operation.
4. Run the script and specify the source path and target path for the relocation.

For example:

```
cleartool mkact -head "Move XML utility code to libks" my-app-xml-move
```

On the UNIX system:

```
rational-home-dir/common/bin/ratlperl ccase-home-dir/etc/utils/mkelem_cpver.pl \
/vobs/app/xml /vobs/lib/ks
```

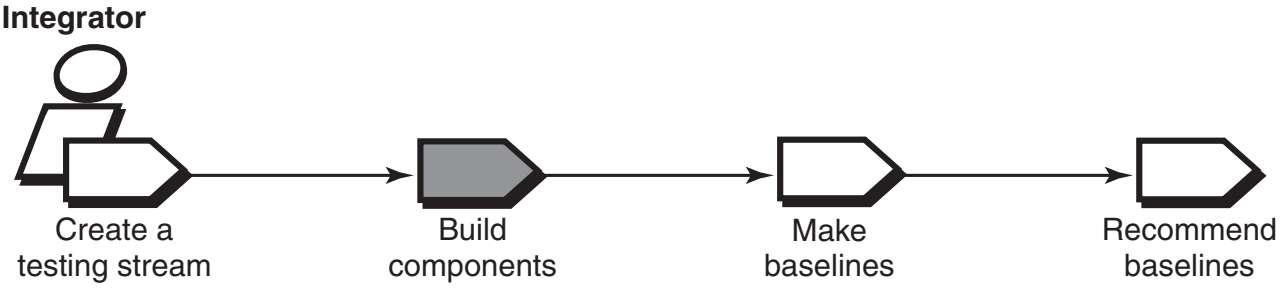
On Linux and the Windows system:

```
rational-home-dir/common/ratlperl.exe ccase-home-dir/etc/utils/mkelem_cpver.pl ^
/vobs/app/xml /vobs/lib/ks
```

The xml directory and all its contents are copied to the new location under the ks directory. If the script encounters a hard link, it creates a separate element.

After the relocation, if developers try to deliver or rebase operations that involve a relocated element, they see warning messages that the element is not visible. Any changes that are made to the source elements must be manually merged from the source location to the target elements in the new location.

Building components



A responsibility of the project integrator is to build and test components to ensure that the changes to the software are stable and meet the goals of the project before the changes are made available in new baselines.

About building components

Before you make new baselines in a stream, build the components by using the current baselines and any work that developers have delivered to the stream since you created the current baselines. If the build succeeds, you can make baselines that select the latest delivered work. Building components involves the following tasks:

- Locking the stream
- Finding remote deliver operations
- Completing remote deliver operations
- Undoing bad deliver operations
- Building and testing the components

Locking the shared stream

Before you build components in the integration stream or feature-specific development stream, lock the stream to prevent developers from delivering work. This ensures that you are dealing with a static set of files.

Note: It is possible that a developer could be in the process of completing a deliver operation when you lock the stream. This scenario could result in some files associated with an activity not being checked in, which, in turn,

could break your build operations and produce bad baselines. You may want to create a script that checks for deliveries in progress, and run the script before you lock the stream.

To lock a stream

1. In the Project Explorer, select the stream.
2. Click **File > Properties** to display the stream property window.
3. Click the **Lock** tab.
4. Click **Locked** and then click **OK**.

Finding work that is ready to be delivered

Before you build components, you may need to complete some deliver operations. In most cases, developers complete their deliver operations. However, in a Rational ClearCase MultiSite configuration in which the target stream is mastered at a different replica than the developer's source stream, the developer cannot complete deliver operations. When such a stream mastership situation is detected, the deliver operation is made a *remote deliver* operation.

In a remote deliver operation, the deliver operation starts but is left in the posted state. It is up to you, as integrator, to find and complete deliver operations in the posted state (see "To find all deliver operations that are in the posted state"). Developers who have deliver operations in the posted state cannot deliver from or rebase their source development streams until you complete their deliver operations (see "To complete remote deliver operations for a development stream") or cancel them (see "Undoing a deliver operation").

Product Note: Rational ClearCase LT does not support Rational ClearCase MultiSite.

To find all deliver operations that are in the posted state

1. In the Project Explorer, select the project.
2. Click **Tools > Find Posted Deliveries**. If the project contains posted deliveries, the Find Posted Deliveries window appears and lists all streams within the project that contain deliver operations in the posted state. For each posted deliver operation, the window shows the source stream and the target stream.
3. To find posted deliver operations for a specific target stream, select the stream and click **Tools > Find Posted Deliveries**. The Find Posted Deliveries window lists the source streams that have posted deliver operations for the target stream. The Find Posted Deliveries window lists posted deliver operations only for source streams that are direct children of the target stream.

To complete remote deliver operations for a development stream

1. Find the list of streams that contain deliver operations in the posted state (see "To find all deliver operations that are in the posted state" on page 113).
2. In the Find Posted Deliveries window, select the development stream from the list.
3. Click **Deliver**. The Deliver window opens. Click **Resume** to resume the deliver operation. Click **Cancel** to cancel the deliver operation. See *Developing Software* online help for details on completing the deliver operation.

Undoing a deliver operation

At any time before developers complete the deliver operation, they can back out of it and undo any changes made; but if they check in their versions to the

integration view, they cannot undo the changes easily. When this happens, you may need to remove the checked-in versions by using the **cleartool rmver –xhlink** command.

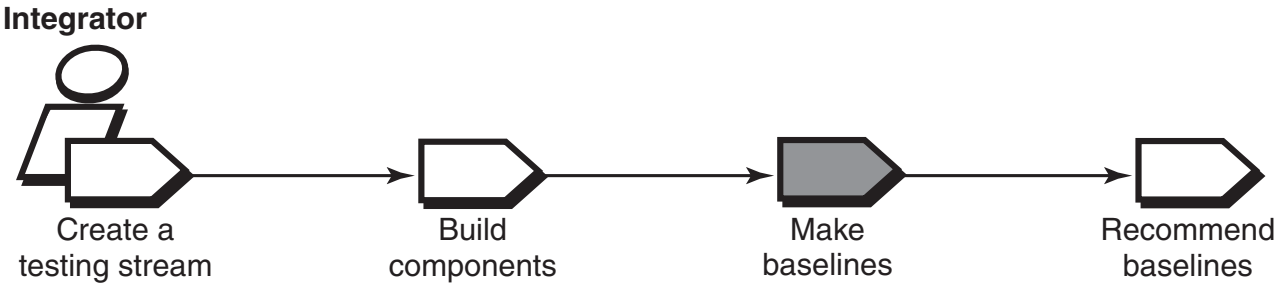
Warning: The **rmver** command erases part of your organizational development history, and it may have unintended consequences. Therefore, be very conservative in using this command, especially with the **–xhlink** option. See the **rmver** reference page in the *IBM Rational ClearCase Command Reference* for details.

Removing a version does not guarantee that the change is really gone. If a successor version was created or if the version was merged before you removed the version, the change still exists. You may need to check out the file, edit it to remove the change, and check the file back in.

Building and testing the components

After you lock the stream and complete any outstanding deliver operations, you are ready to build and test the project executable files to make sure that the changes delivered by developers since the last baseline do not contain any bugs. For information on performing builds, see *IBM Rational ClearCase Guide to Building Software*. Because you lock the stream when you build and test in it, it is better to use a separate development stream for extensive testing of new baselines. For information about using a development stream for testing new baselines, see "Testing the baseline" on page 116. Perform only quick validation tests in the current stream so that it is not locked for an extended period of time.

Creating a new baseline



Project integrators are responsible for making baselines before they recommend them.

About making a baseline

As developers deliver work to the integration stream or feature-specific development stream, it is important that you make new baselines frequently to record the changes. Developers can then rebase to the new baselines and stay current with each other's changes. Before you make the baseline, make sure that the stream is still locked so that developers cannot deliver work to the stream.

By default, all activities modified since the last baseline was made are included in the new baseline. There might be times when you want to create a baseline that includes only certain activities. You can also make a baseline for one specific component rather than all components in the stream.

You need to choose the type of baseline to create.

- An *incremental baseline* is a baseline that is created by recording the last full baseline and those versions that have changed since the last full baseline was created.
- A *full baseline* is a baseline that is created by recording all versions below the component root directory.

Generally, incremental baselines are faster to create than full baselines; however, the contents of a full baseline can be searched faster than the contents of an incremental baseline can.

While you are making the baseline, you should lock the stream (see "Locking the shared stream" on page 112). After you create a baseline, unlock the integration or feature-specific development stream so that developers can resume delivering work to the stream.

To make a baseline

1. Lock the stream to prevent developers from delivering work while you create the baseline (see "To lock a stream" on page 113). Developers can continue to work on activities in their development streams.
2. Verify the stability of the project by testing its components.
3. Make the baseline. Do one of the following:
   - Make baselines for all components in the stream (see "To make new baselines for all components in the stream" on page 115).
   - Make a baseline for certain activities (see "To make a baseline for a set of activities" on page 116).
   - Make a baseline for one specific component (see "To make a baseline of one component" on page 116).
4. Unlock the stream so that developers can deliver work (see "To unlock the stream" on page 116).

For information on baselines, see "Specifying a baseline strategy" on page 45.

To make new baselines for all components in the stream

1. Ensure that the stream is locked (see "To lock a stream" on page 113).
2. In the Project Explorer, select the integration stream or feature-specific development stream in which you want to make the baseline.
3. Click **Tools > Make Baseline**. The Make Baseline window opens. The **Project/Stream** field shows the object selector of the stream that you selected.
4. The **Template Name** field shows the name that will be used for the new baseline if a baseline naming template is set for the project. For information about baseline naming templates, see "Setting a baseline naming template" on page 92).

   Enter a name in the **Base Name** (Windows systems) or **Baseline Title** (Linux and the UNIX system) field only if the project does not have a baseline naming template set or the template includes the basename token. Otherwise, the **Base Name** or **Baseline Title** field does not appear in the Make Baseline window.
5. Choose the type of baseline to create (see "About making a baseline" on page 114).
6. In **View Context**, specify a view in which to perform the operation. Choose a view that is attached to the stream in which you want to make the baseline.

To make a baseline for a set of activities

1. Ensure that the stream is locked (see "To lock a stream" on page 113).
2. In the Project Explorer, select the integration stream or feature-specific development stream in which you want to make the baseline.
3. Click **Tools > Make Baseline**. The Make Baseline window opens.
4. If you have a baseline naming template set for the project, the **Template Name** field shows the name that will be used for the new baseline. Enter a name in the **Base Name** (Windows systems) or **Baseline Title** (the UNIX system) field only if the project does not have a baseline naming template set or the template includes the basename token. Otherwise, the **Base Name** or **Baseline Title** field does not appear in the Make Baseline window. For information about baseline naming templates, see "Setting a baseline naming template" on page 92).
5. Click **Activities** in the Make Baseline window, and select the activities that you want to go into the baseline.
6. Click **General** and select the type of baseline to create.
7. Specify a view in which to perform the operation. Choose a view that is attached to the stream where you want to make the baseline.

To make a baseline of one component

1. Ensure that the stream is locked (see "To lock a stream" on page 113).
2. In the Project Explorer, select the stream in which you want to create a new baseline. Click **File > Properties** to display the stream property window.
3. Click the **Baselines** tab. Select a component, and click **Make Baseline**.
4. Fill in the fields of the Make Baseline window, then click **OK**.

To unlock the stream

1. In the Project Explorer, select the stream.
2. Click **File > Properties** to display the property sheet of the stream.
3. Click the **Lock** tab.
4. Click **Unlocked** and then click **OK**.

Testing the baseline

To avoid locking the integration stream or feature-specific development stream for an extended period of time, use a separate development stream for performing extensive testing, such as system, regression, and acceptance tests, on new baselines. See "Creating a development stream for testing baselines" on page 106 for information about creating a development stream.

To test in a separate development stream

To use a test stream to stabilize code, perform the steps shown in Figure 37.

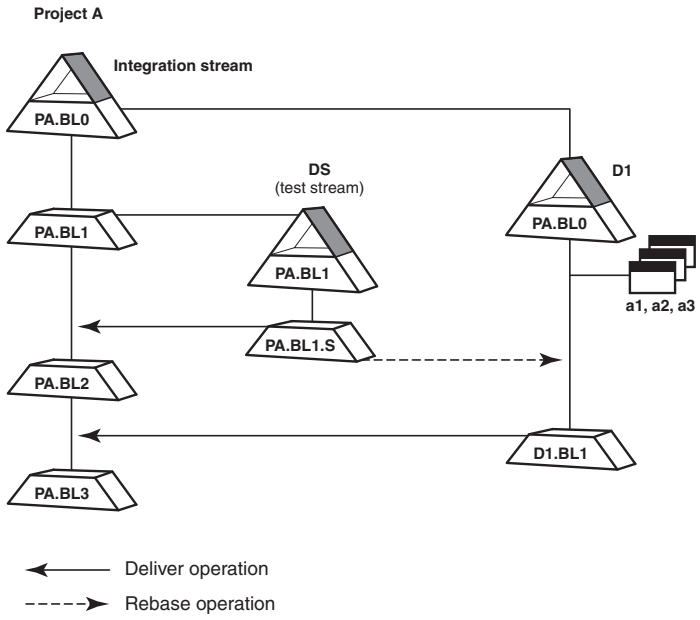


Figure 37. A test stream to stabilize a baseline

The steps are:

1. Make a baseline that contains the changes to be tested (see **PA.BL1** in Figure 37).
2. Do one of the following:
   - Create a development stream dedicated to stabilization (see stream **DS** in Figure 37). For information about creating a development stream, see "To create a development stream" on page 106.
   - Rebase a dedicated development stream to the baseline that you made (see "Rebasing the test development stream" on page 117).
3. Use as the foundation baseline of the test stream the baseline that you created (see **PA.BL1** in Figure 37).
4. Control the changes that are being made in the stabilization stream **DS**. Other work from development streams in the project can be delivered to the integration stream without affecting the stabilization stream. Fixes implemented in the stabilization stream are isolated from activities delivered to the integration stream.
5. When the code in the test stream is stable, make a baseline in the test stream (see **PA.BL1.S** in Figure 37).
6. Deliver the baseline **PA.BL1.S** to the integration stream.
7. In the integration stream, recommend the baseline from the stabilization stream so that development streams can rebase to it. (This is an example of an advance rebase operation; see "Advance rebase operations" on page 21.)

Rebasing the test development stream

After you create a new baseline and verify that it builds and passes an initial validation test, rebase the development stream to the new baseline. For information about the rules for rebasing a stream, see "Summary of rules for rebasing a stream" on page 23. When you finish rebasing the development stream, you are ready to begin testing the new baselines.

To rebase the development stream

1. In the Project Explorer, select the development stream and click **Tools > Rebase Stream**. The Rebase Stream Preview window opens.
2. By default, your development stream rebases to the *recommended baselines*. Because the new baseline has not been tested extensively, you probably have not yet promoted it to the level associated with recommended baselines. To rebase to the baseline, or baselines, you want to test, click **Advanced**. The Change Rebase Configuration window opens.
3. Select a component that contains a baseline you want to test. Click **Change**. The Change Baseline window opens, listing all baselines for the component.
4. Select the baseline that you want to test, and click **OK**.
5. Select another component in the Change Rebase Configuration window and repeat the process. When you finish selecting baselines, click **OK** to close the Change Rebase Configuration window.
6. Click **OK** in the Rebase Stream Preview window to continue the rebase operation. See the Help or *Developing Software* online help for details on rebasing a development stream.

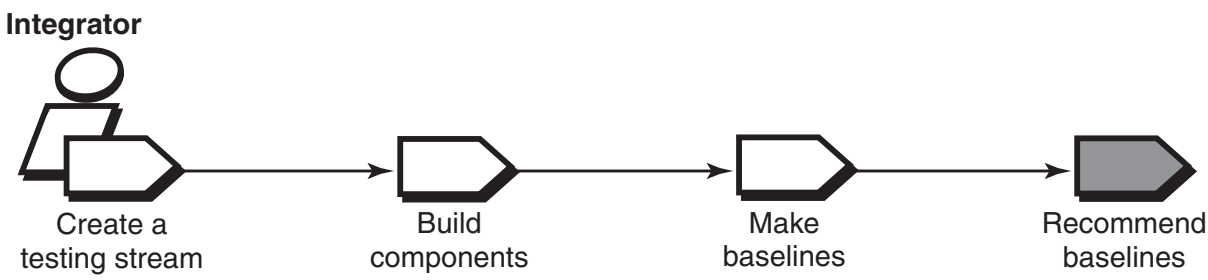
Fixing problems in baselines

If you discover a problem with a baseline while testing it, fix the affected files and deliver the changes to the integration stream.

To fix a problem in a new baseline

1. From the development view attached to the development stream, specify an activity and check out the files that you need to fix.
2. Make the necessary changes to the files and check them in.
3. Build and test the changes in the development view.
4. When you are confident that the changes work, make a new baseline that incorporates the changes in the development stream.
5. Deliver the new baseline to the integration or feature-specific development stream. When you deliver the new baseline to the integration or feature-specific development stream, you merge changes with work that developers have delivered since the last baseline was created. For information about delivering baselines, see “Delivering work from an integration stream to another project” on page 151.
6. Change the set of recommended baselines for the integration stream or feature-specific development stream to include the new baseline that you made in the testing stream. For details about recommending a baseline in another stream, see “Recommending the baseline” on page 118.

Recommending the baseline



As work on your project progresses and the quality and stability of the components improve, you make new baselines and want to make the work available to the team. Do the following:

- Change the baseline promotion level attribute to reflect a level of testing that the baseline has passed.
- Recommend baselines that have passed extensive testing.

To change a baseline promotion level

1. Access the stream that contains the baseline.

   On the Windows system, in the Project Explorer, right-click the stream and click **Properties**.

   On Linux and the UNIX system, in the Project Explorer, select the stream. Click **File > Properties**.

   The stream Properties window opens.
2. Click the **Baselines** tab.
3. In the **Components** list, select the component that contains the baseline that you want to promote. In the **Baselines** list, select the baseline. Click **Properties**. The baseline Properties window opens.
4. Click the arrow in the **Promotion Level** list to display all available promotion levels. Select the new promotion level.

To recommend a baseline or set of baselines

1. In the Project Explorer, select the stream. Click **Tools > Recommend Baselines**.
2. In the Recommended Baselines window, you can filter the list of baselines displayed by selecting a promotion level and clicking **Seed List**. The window then displays only baselines at or above the selected promotion level.
3. To remove a baseline from the list, select it and click **Remove**. To add a baseline, click **Add** and select the baseline in the Add Baseline window.
4. To recommend a different baseline of a component, select the baseline and click **Change**. In the Change Baseline window, select the baseline that you want to recommend. To select a baseline in another stream, such as a testing stream, click **Change** and navigate to the stream in the Change Baseline window.

   You can recommend a baseline for a stream if the baseline is from the stream or the stream's foundation.

   For a baseline that is not from the stream or from the foundation set of the stream, the following rules apply:

   - The baseline must be an ancestor of the foundation baseline of the stream and must have been created on the same stream as the foundation baseline.
   - The baseline must be contained in the stream, which means the baseline has been delivered to the stream, or the stream has rebased to the baseline or to one of its descendants.
   - The baseline must contain the current recommended baseline, which means it must be a descendant of the current recommended baseline.

   You are not required to recommend a baseline for every component in the configuration of the stream.
5. When you finalize your list of recommended baselines, click **OK** in the Recommended Baselines window.

Resolving baseline conflicts

If your project uses composite baselines, you may encounter a situation where you must resolve a conflict in a stream configuration between two different baselines of the same component. The following conflicts can occur during operations that involve baselines:

- Making a baseline
- Adding a baseline to a stream configuration
- Recommending a baseline
- Rebasing a stream

For information on composite baselines, see "Identifying a project baseline" on page 46.

Conflicts between a composite baseline and an ordinary baseline

A composite baseline can conflict with an ordinary baseline. For example, assume that a stream configuration includes a composite baseline that selects baseline **BL4** of component **A**, and that the composite baseline is the recommended baseline. After testing a new baseline, **BL5**, of component **A**, you decide to recommend it. By doing so, you override the member baseline, **BL4**, selected by the composite baseline. The Recommended Baselines window identifies **BL5** as an override and **BL4** as overridden. UCM uses the same override and overridden identifiers in other GUIs.

Conflicts between composite baselines

A conflict can occur when a stream configuration includes multiple composite baselines where each composite baseline selects a baseline of the same component. A stream cannot select two different baselines of the same component. If you attempt to perform an operation that would cause this situation, UCM recognizes the conflict and forces you to resolve it before completing the operation.

Composite baselines promote component reuse by making it easier to include large components and subsystems into a project. As the number of shared components rises, a higher probability exists that different subsystems will have a baseline conflict. Higher-level projects that use lower-level subsystems are increasingly likely to include in their foundation sets composite baselines that have members in the same component. Conflicts arise when the members are not the same baseline for a particular component. For example, suppose **AC.BL1** and **BCD.BL1** are composite baselines that each select baselines of component **C** (see Figure 38).

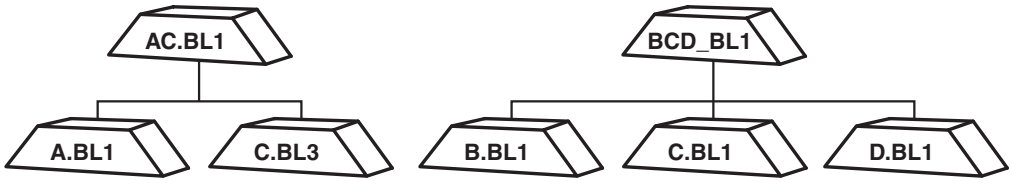


Figure 38. Composite baselines with the same component

Baseline **C.BL3** a member of composite baseline **AC.BL1** and baseline **C.BL1** a member of composite baseline **BCD.BL1**.

If baselines **AC.BL1** and **BCD.BL1** are configured in project **Y**, it is unclear to Rational ClearCase which baseline on component **C** to use, baseline **C.BL1** or **C.BL3** (see Figure 39).

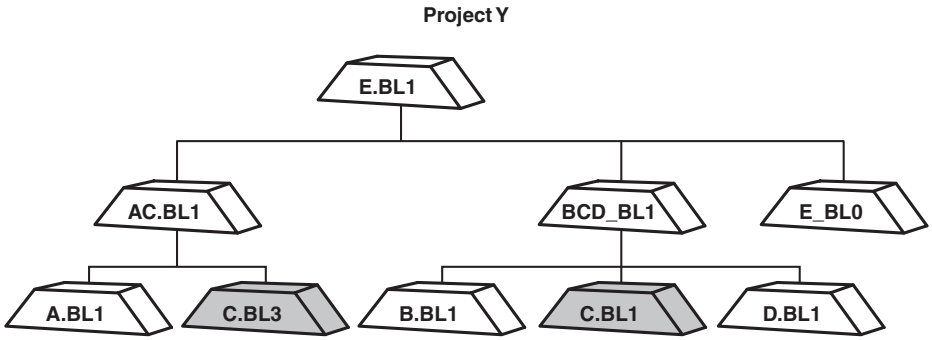


Figure 39. Composite baselines with a conflict

In the Rational ClearCase environment, a view must have an unambiguous rule for selecting versions of an element. In UCM, a stream can only use one baseline to select the versions in a component. Rebase operations and baseline recommendations that would result in conflicts are blocked.

To resolve the conflict, you are forced to explicitly specify a baseline for the component in question. This chosen baseline is said to *override* the members of the composite baselines in conflict. A baseline that you explicitly specify as an override baseline in the foundation set of a stream, regardless of whether it resolves a conflict, overrides any baseline of that component that is implied by a composite baseline.

To resolve the conflict on component **C** shown in Figure 39, the project integrator chooses baseline **Cx.BL3** (see Figure 40).

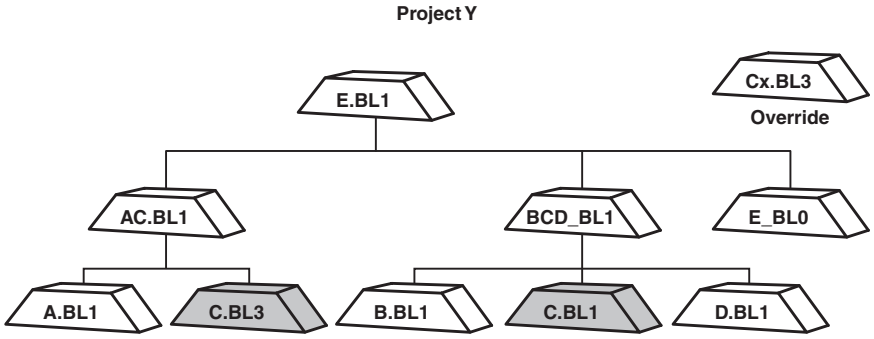


Figure 40. Composite baselines with an override baseline

Because an override was selected, the baseline **C.BL3** in composite baseline **AC.BL1** and the baseline **C.BL1** in composite baseline **BCD.BL1** are ignored. Baseline **Cx.BL3** is used to select versions in component **C**.

Tip: The override applies only for the baseline for component **C** in the foundation baselines of the stream, but the composite baseline itself remains the same.

You can choose as the override any baseline of the component involved in the conflict. The overriding baseline does not have to be one of the conflicting baselines. The project integrator can select a baseline that is compatible with the

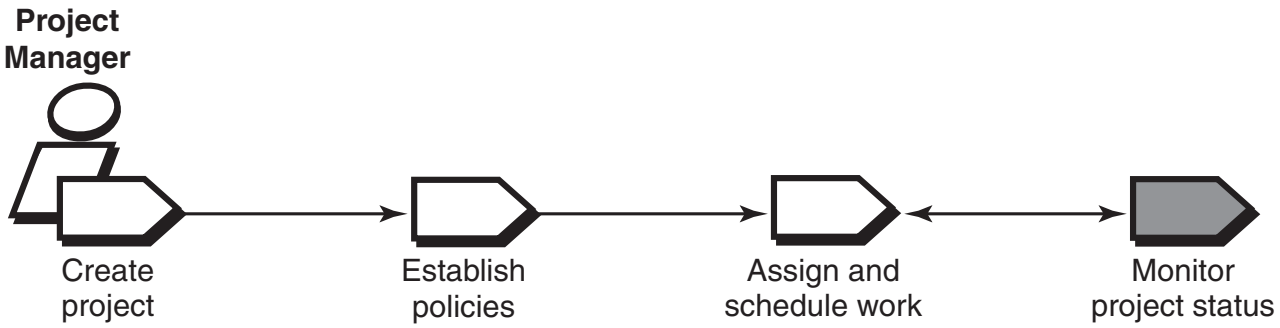
other baselines in the baseline set of the project. In Figure 40, the project integrator could have chosen baseline **C.BL1** as the override, instead of baseline **Cx.BL3**. However, the integrator must ensure that the versions selected by composite baseline **AC.BL1** are compatible with the versions selected by baseline **C.BL1**. With the selection of the override **Cx.BL3**, the AC and BCD subsystems need to be checked to ensure that they are compatible with baseline **Cx.BL3**.

Baseline overrides stay in effect until you do one of the following:

- Explicitly remove the overriding baseline from the foundation set (for example, with **cleartool rebase –dbaseline**).
- Replace completely the foundation set. This happens when you rebase to the recommended baselines of the parent stream (**cleartool rebase –recommended**).

The decision to select a baseline override is solely for the project integrator. It is not a decision that can be automated. Each project team has to determine the correct override in each instance of a conflict.

Monitoring project status



Several tools are provided to help you track the progress of your project. You can do the following operations:

- View baseline histories
- Compare baselines
- Query a Rational ClearQuest user database
- Generate reports (Windows only)

Viewing baseline histories

On the Windows system, the Component Tree Browser displays the baseline history of a component. On the UNIX system, the **cleartool lscomp** command lists information about a component, including its baselines.

To view baseline history (the Windows system)

1. In the Project Explorer, navigate to the component whose baseline history you want to see.
2. Right-click the component and click **Browse Baselines**.

The Component Tree Browser opens and shows the lines of development for the component and each stream that uses the component. You can see the initial baseline that was created when the project manager created the component and the first baseline that the integrator created after creating the component. Also shown are baselines that are created in the development stream during deliver operations

and integration arrows that represent deliver operations, for example, from the development stream to the integration stream .

To view baseline history (Linux and the UNIX system)

From a shell prompt, use the **cleartool lscomp -tree** command and specify the component. For example:

```
cleartool lscomp -tree guivob@/vobs_guipvob
```

Baselines and streams in the specified component are listed. The format is similar to that of the **lsvtree** command. For more information, see the **lscomp** reference page.

Comparing baselines

You can display the differences between two baselines graphically or from a command shell.

To compare baselines in Component Tree Browser (Windows only)

1. View the baseline history (see "To view baseline history (the Windows system)" on page 122).
2. In the Component Tree Browser, select a baseline by clicking its icon. Do one of the following:
   - To compare two baselines, click **Tools > Compare > with Another Baseline**. Click the second baseline icon.
   - To compare a baseline with its immediate predecessor, click **Tools > Compare > with Previous Baseline**.

The Compare Baselines window opens. For more information, see "About the Compare Baselines window."

To compare two baselines

Do one of the following:

- Use the **cleardiffbl** command and specify two baselines. For example:

  ```
  cleardiffbl OM_proj2.0_Integration_08_12_01 OM_proj2.0_09_06_01
  ```

  The Compare Baselines window opens.
- Open the Compare Baselines window from within the baseline Property window.
  1. In Project Explorer, select the integration stream, and click **File > Properties** to display the integration stream Property window.
  2. Click the **Baselines** tab and then select the component that contains the baseline you want to compare.
  3. Select the baseline; then right-click it and click **Compare with Previous Baseline** or **Compare with Another Baseline**.

  The Compare Baselines window opens.

For more information, see "About the Compare Baselines window."

About the Compare Baselines window

The Compare Baselines window shows the results of a comparison of two baselines in the following pages:

Members
Shows the baselines that contribute to each baseline of a composite baseline

Activities
Lists the activities (if any) that contribute to the baseline. A baseline could contain no activities if it or its member baselines are the initial baselines of the component.

Versions
Lists the change sets associated with the activities in the baseline.

Querying Rational ClearQuest user databases

If you use the UCM integration with Rational ClearQuest, you can use Rational ClearQuest queries to retrieve information about the state of your project. When you create a new Rational ClearQuest user database or upgrade an existing Rational ClearQuest user database to use a UCM-enabled schema, the integration installs some queries in subfolders of the Public Queries folder in the user database workspace. These queries make it easy for developers to see which activities are assigned to them and for project managers to see which activities are active in a particular project. Table 4 lists and describes the queries.

Table 4. Queries in a UCM-enabled schema

Query	Description
ActiveForProject	For one or more specified projects, selects all activities in an active state type.
ActiveForStream	For one or more specified streams, selects all activities in an active state type.
ActiveForUser	For one or more specified developers, selects all assigned activities in an active state type.
AllActivitiesInStream	For one or more specified streams, selects all activities.
MyCompletedWork	Selects all activities in a completed type state for the developer running the query.
MyToDoList	Selects all activities in an active or ready state type assigned to the developer running the query.
UCMProjects	Selects all projects linked to the Rational ClearQuest user database.
UCMCustomQuery1	This query is not intended to be used directly by users; the integration uses it. When a developer checks out or checks in a file, or adds a file to source control and is prompted to select an activity, the integration calls this query to display the list of activities available in the stream associated with the developer's view. You can customize this query on a per-developer basis by copying the query from the **Public Queries** folder to the developer's **Personal Queries** folder and using the Query editor.

You can also create your own queries by clicking **Query > New Query** within the Rational ClearQuest client. In the Choose a record type window that opens, select **All_UCM_Activities** if you want the query to search all UCM-enabled record types.

Using Rational ClearCase Reports (Windows systems only)

The Rational ClearCase Reports applications (Report Builder and Report Viewer) allow you to generate and view reports specific to your project environment. Use the Report Builder to select and define report parameters. Use the Report Viewer to see the report output.

Product Note: To start the Rational ClearCase Report Builder:

- On a system that runs Rational ClearCase, click **Start > Programs > IBM Rational > IBM Rational ClearCase > Administration > Report Builder**.
- On a system that runs the ClearCase LT server, click **Start > Programs > IBM Rational > IBM Rational ClearCase LT > Administration > Report Builder**.

The Report Builder categorizes its reports based on object types, such as UCM projects and streams. When you select a category in the left pane, the Report Builder lists the reports available for that category in the upper right pane. When you select a report, the Report Builder prompts you for parameters in the lower right pane.

For details on using the Report Builder and the Report Viewer, see Help.

Rational ClearCase Reports includes a set of hooks into the Report Builder and Report Viewer applications. These hooks, known as report procedures, implement all the operations necessary to generate and view a specific report. The Rational ClearCase Reports Programming Interface allows you to customize report procedures. For details on doing so, see Appendix C, "Customizing Rational ClearCase Reports," on page 265.

Cleaning up the project

When your team finishes work on a project and releases or deploys the new software, you should clean up the project environment before creating the next version of the project. Cleaning up involves removing any unused objects, and locking and hiding the project and its streams. This process reduces clutter and makes it easier to navigate in the Project Explorer.

Removing unused objects

During the life of the project, you or a developer might create an object and then decide not to use it. Perhaps you decide to use a different naming convention, and you create a new object instead of renaming the existing one. To avoid confusion and reduce clutter, remove these unused objects.

About deleting projects

Note: By design, you cannot delete a project in which a delivery or a rebase operation has been performed or from which a delivery to another project has been performed.

You can delete a project only if it does not contain any streams. When you create a project with the Project Creation Wizard, the wizard also creates an integration stream. Therefore, you can delete a project only if you created it with the **cleartool mkproject** command, or if you first delete the integration stream. For more information on removing projects, see the **rmproject** reference page in the *IBM Rational ClearCase Command Reference*. To remove an unused project, see "To delete an unused object."

About deleting streams

You can delete a development stream or an integration stream only if all of the following conditions are true:

- The stream contains no activities.
- No baselines have been created in the stream.
- No views are attached to the stream.

In addition, you cannot delete an integration stream if the project contains any development streams. For more information on removing streams, see the **rmstream** reference page in the *IBM Rational ClearCase Command Reference*. To remove an unused stream, see "To delete an unused object."

About deleting components

You can delete a component only if all of the following conditions are true:

- No baselines of the component other than its initial baseline exist.
- The initial baseline of the component does not serve as a foundation baseline for another stream.

For more information about removing components, see the **rmcomp** reference page in the *IBM Rational ClearCase Command Reference*. To remove an unused component, see "To delete an unused object."

About deleting baselines

You can delete a baseline only if all of the following conditions are true:

- The baseline does not serve as a foundation baseline.
- The baseline is not a component initial baseline.
- A stream has not made changes to the baseline.
- The baseline is not used as the basis for an incremental baseline.

For more information about removing baselines, see the **rmbl** reference page in the *IBM Rational ClearCase Command Reference*. To remove an unused baseline, see "To delete an unused object."

About deleting activities

You can delete an activity only if both of the following conditions are true:

- The activity has no versions in its change set.
- No view is currently set to the activity.

For more information about removing activities, see the **rmactivity** reference page in the *IBM Rational ClearCase Command Reference*. To remove an unused activity, see "To delete an unused object."

To delete an unused object

To delete an unused object, perform the following steps:

1. Ensure that the requirements for the type of object are satisfied.

- For a project, see “About deleting projects” on page 125.
- For a stream, see “About deleting streams” on page 126.
- For a component, see “About deleting components” on page 126.
- For a baseline, see “About deleting baselines” on page 126.
- For an activity, see “About deleting activities” on page 126.

2. Select the object in the Project Explorer, and click **File > Delete**.

   To delete a baseline (see “About deleting baselines” on page 126), use the **cleartool rmbl** command.

Locking and making obsolete the project and streams

To prevent a project or a stream from appearing in the Project Explorer, lock the object and use the obsolete option. The obsolete option hides the object.

To lock and hide an object

1. In the Project Explorer, select the stream or project that you want to hide, and click **File > Properties** to display its property sheet.
2. Click the **Lock** tab, and select **Obsolete**. Click **OK**.

To see objects that are obsolete

In the Project Explorer, click **View > Show Obsolete Items**.

Chapter 8. Using triggers to enforce UCM development policies

UCM provides a group of development policies that you can easily set in a project by using the graphic user interface (GUI) or command line interface (CLI) and additionally supports triggers to enforce policies. For information about UCM policies, see Chapter 4, "Setting policies," on page 63.

Overview of triggers

A *trigger* is a monitor that causes one or more procedures or actions to be run whenever a certain Rational ClearCase operation is performed. Typically, the trigger runs a Perl, batch, or shell script. You can use triggers to restrict operations to specific users and to specify the conditions under which they can perform those operations.

In addition, you can use triggers on certain UCM operations to enforce customized development policies for your project team. You can create triggers and use them to implement various development policies in UCM projects. For additional information about trigger usage, see the **cleartool mktrigger** and **mktrtype** reference pages.

Supported triggers

You can use triggers with the following UCM operations:

- **chbl**
- **chfolder**
- **chproject**
- **chstream**
- **deliver**
- **mkactivity**
- **mkbl**
- **mkcomp**
- **mkfolder**
- **mkproject**
- **mkstream**
- **rebase**
- **rmbl**
- **rmcomp**
- **rmfolder**
- **rmproject**
- **rmstream**
- **setactivity**
- **setplevel**

You can also use triggers with the following Rational ClearCase operations on UCM objects:

- **lock**
- **unlock**

You can define trigger types that can be set on lock and unlock operations and can restrict them to some individually named UCM objects or all UCM objects (activities, baselines, components, folders, projects, and streams).

Preoperation and postoperation triggers

Triggers fall into one of two categories. Preoperation triggers *fire*, or run their corresponding procedures, before an operation takes place. Postoperation triggers fire after an operation occurs. When a user enters a Rational ClearCase command, the presence of preoperation triggers on that command are checked. If a trigger is associated with the command, the trigger procedure is fired. If the trigger procedure finishes with a failure status, the operation requested by the user is disallowed. If the trigger procedure finishes with a success status, the operation is performed.

Use preoperation triggers to prevent users from performing operations unless certain conditions apply. Use postoperation triggers to perform actions after an operation completes. For example, you may want to place a postoperation trigger on the deliver operation to notify team members whenever a developer delivers work to the project's integration stream.

Scope of triggers

A *trigger type* defines a trigger for use within a VOB or PVOB. When you create a trigger type, with the **cleartool mktrtype** command, you specify the scope to be one of the following:

- An *element trigger type* applies to one or more elements. You attach an instance of the trigger type to one or more elements by using the **cleartool mktrigger** command.
- An *all-element trigger type* applies to all elements in a VOB.
- A *type trigger type* applies to type objects, such as attributes types, in a VOB.
- A *UCM trigger type* applies to a UCM object, such as a stream or a project, in a PVOB.
- An *all-UCM-object trigger type* applies to all UCM objects in a PVOB.

Using attributes with triggers

As you design triggers to enforce development policies, you may find it useful to use attributes. An *attribute* is a name/value pair. An *attribute type* defines an attribute. You can apply an attribute to an object, such as a stream or an activity, or to a version of an element. In your trigger scripts, you can test the value of an attribute to determine whether to fire the trigger. For example, you could define an attribute type called **TESTED** and attach a **TESTED** attribute to elements to indicate whether they had been tested. Acceptable values would be **Yes** and **No**.

When to use Rational ClearQuest scripts instead of UCM triggers

There are several use cases for UCM triggers. If your UCM project is enabled to work with Rational ClearQuest, you can set the following policies, which are described in "Policies for the UCM integration with Rational ClearQuest" on page 69:

- For submitting records from the Rational ClearCase client
  - Disallow Submitting Records from the ClearCase Client
  - Allowed Record Types
- For WorkOn

  - Perform ClearQuest Action Before Work On
- For delivery
  - Perform ClearQuest Action Before Delivery
  - Transfer ClearQuest Mastership Before Delivery
  - Perform ClearQuest Action After Delivery
  - Transition to Complete After Delivery
  - Transfer ClearQuest Mastership After Delivery
- For changing an activity
  - Perform ClearQuest Action Before Changing Activity
  - Perform ClearQuest Action After Changing Activity
  - Transition to Complete After Changing Activity

Some of these policies have Rational ClearQuest global hook scripts associated with them, which you can edit or replace in Rational ClearQuest Designer to customize the policy for your environment. You can also write your own Rational ClearQuest hooks to enforce development policies. In general, if the policy you want to enforce involves a Rational ClearQuest action, use one of the Rational ClearQuest policies previously mentioned or use Rational ClearQuest hooks. If the policy you want to enforce involves a Rational ClearCase action, use UCM triggers.

Some operations might have Rational ClearCase triggers and Rational ClearQuest hooks associated with them. For example, you might define a trigger that sends e-mail to team members when a developer completes a deliver operation, and you might have the Perform ClearQuest Activity After Delivery policy enabled. Under Rational ClearCase and Rational ClearQuest control, triggers, hooks, and UCM operations are run in the following order:

- Rational ClearCase preoperation trigger
- Rational ClearQuest preoperation hook
- UCM action
- Rational ClearQuest postoperation hook
- Rational ClearQuest transition activity hook
- Rational ClearCase postoperation trigger

You can use the Rational ClearQuest API to write code that runs in the Rational ClearQuest environment. For example, you can modify records that users submit or validate the records before they are committed to the user database. For code examples that work with **cqperl** on Linux and the UNIX system or that use CAL methods in Rational ClearQuest hook scripts on the Windows system, see *IBM Rational ClearQuest API Reference*.

Sharing triggers among different types of platform

You can define a trigger that fires correctly depending on the type of platform on which it runs (Linux, the UNIX system, and Windows computers). The following techniques are available:

- "Using different paths or different scripts" on page 132
- "Using the same script" on page 132

With one technique, you use different paths or different scripts; with the other technique, you use the same script for all platforms. For more information about sharing triggers, see "Tips for sharing scripts" on page 132.

Using different paths or different scripts

To define a trigger that fires on Linux and the UNIX system; the Windows system; or both types of platform, and that uses different paths to point to the trigger scripts, use the **mktrtype** command with the **–execunix** and **–execwin** options. These options behave the same as **–exec** when the trigger fires on the appropriate platform (Linux and the UNIX system for **–execunix** or the Windows system for **–execwin**). On the inappropriate type of platform, the related script does not run.

This technique allows a single trigger type to use different paths for the scripts or to use completely different scripts on Linux or the UNIX system and the Windows computer. For example:

```
cleartool mktrtype –element –all –nc –preop checkin
–execunix /public/scripts/precheckin.sh
–execwin \\neon\scripts\precheckin.bat
pre_ci_trig
```

Tip: The command line example is broken across lines to make the example easier to read. You must enter the command on one line.

On Linux or the UNIX system, only the script precheckin.sh runs. On the Windows system, only precheckin.bat runs.

To prevent users on a new platform from bypassing the trigger process, triggers that specify only **–execunix** always fail on the Windows system. Likewise, triggers that specify only **–execwin** fail on Linux and the UNIX system.

Using the same script

To use the same trigger script on Linux, the UNIX system, and the Windows system, use a batch command interpreter that runs on all operating systems. For this purpose, the **ratlperl** program is included in the Rational ClearCase configuration. You can use this version of Perl on the Windows system, Linux, and the UNIX system. The commands **Perl** on Linux and the UNIX system and **ccperl** on the Windows system are wrapper programs that run **ratlperl**.

The following **mktrtype** command creates sample trigger type **pre_ci_trig** and names precheckin.pl as the executable trigger script.

```
cleartool mktrtype –element –all –nc –preop checkin  \
–execunix 'Perl /public/scripts/precheckin.pl' \
–execwin 'ccperl \\neon\scripts\precheckin.pl' \
pre_ci_trig
```

Note: In your scripts, you can run **ratlperl** directly. Ensure that you include the following default paths to execute the scripts successfully:

- On Linux and the UNIX system: /opt/rational/common/
- On the Windows system: <*install_location*>\Rational\Common\

The value *install_location* is the root folder in which you installed Rational ClearCase.

Tips for sharing scripts

- To tailor script execution for each operating system, use environment variables in Perl scripts.
- To collect or display information interactively, use the **clearprompt** command.

- For more information on using the **–execunix** and **–execwin** options, see the **mktrtype** reference page.

Enforce serial deliver operations

Because UCM allows multiple developers to concurrently deliver work to the same integration stream, conflicts can occur if two or more developers attempt to deliver changes to the same element. If one developer's deliver operation has an element checked out, the second developer cannot deliver changes to that element until the first deliver operation is completed or canceled. The second deliver operation attempts to check out all elements other than the checked-out one, but it does not proceed to the merge phase of the operation. The second developer must either wait for the first deliver operation to finish or undo the second deliver operation.

You may want to implement a development policy that eliminates the confusion that concurrent deliveries can cause developers. The following sections show Perl scripts that prevent multiple developers from delivering work to the same integration stream concurrently:

- Script 1 creates the trigger types and an attribute type.
- Script 2 is the preoperation trigger action that fires at the start of a deliver operation.
- Script 3 is the postoperation trigger action that fires at the end of a deliver operation.

For information about sharing scripts, see "Sharing triggers among different types of platform" on page 131.

Delivery setup script

This setup script creates a preoperation trigger type, a postoperation trigger type, and an attribute type. The preoperation trigger action fires when a deliver operation starts, as represented by the **deliver_start** operation kind (*opkind*). The postoperation trigger action fires when a deliver operation is canceled or completed, as represented by the **deliver_cancel** and **deliver_complete** opkinds, respectively.

The script runs on both Linux or the UNIX system and the Windows system. Because the command-line syntax to run the preoperation and postoperation scripts on Windows differs slightly depending on whether the PVOB resides on Windows, Linux, or the UNIX system, the setup script uses an **IF ELSE** Boolean expression to set the appropriate PVOB tag.

The **mktrtype** command uses the **–ucmobject** and **–all** options to specify that the trigger type applies to all UCM objects in the PVOB, but the **–stream** option restricts the scope to one integration stream.

The **mkattype** command creates an attribute type called **deliver_in_progress**, which the preoperation and postoperation scripts use to indicate whether a developer is delivering work to the integration stream.

```
use Config;

my $PVOBTAG;
my $PREOPCMDW;
my $POSTOPCMDW;

$PREOPCMDW = '-execwin "ccperl
\\\\pluto\\c$\\ucmscripts\\ex1_preop.pl"';
```

```
$POSTOPCMDW = '-execwin "ccperl
\\\\pluto\\c$\\ucmscripts\\ex1_postop.pl"';
if ($Config{'osname'} eq 'MSWin32') {
   $PVOBTAG = '\cyclone-pvob';
}
else {
   $PVOBTAG = '/pvobs/cyclone-pvob';
}

my $PREOPCMDU = '';
my $POSTOPCMDU = '';
my $STREAM = "cc5testproj_Integration\@$PVOBTAG";
my $PREOPTRTYPE = "trtype:ex1_preop\@$PVOBTAG";
my $POSTOPTRTYPE = "trtype:ex1_postop\@$PVOBTAG";
my $CANXTRTYPE = "trtype:ex1_cancel\@$PVOBTAG";
my $ATTYPE = "attype:deliver_in_progress\@$PVOBTAG";

print $PVOBTAG . "\n";
print $STREAM . "\n";
print $PREOPTRTYPE . "\n";
print $POSTOPTRTYPE . "\n";
print $CANXTRTYPE . "\n";
print $ATTYPE . "\n";

print `cleartool mktrtype -ucmobject -all -preop deliver_start
$PREOPCMDU $PREOPCMDW -stream $STREAM -nc $PREOPTRTYPE`;
print `cleartool mktrtype -ucmobject -all -postop deliver_complete
$POSTOPCMDU $POSTOPCMDW -stream $STREAM -nc $POSTOPTRTYPE`;
print `cleartool mktrtype -ucmobject -all -postop deliver_cancel
$POSTOPCMDU $POSTOPCMDW -stream $STREAM -nc $CANXTRTYPE`;
print `cleartool mkattype -vtype integer -default 1 -nc $ATTYPE`;
```

Delivery preoperation trigger script

This preoperation trigger action fires when a developer begins to deliver work to the specified integration stream. The script attempts to attach an attribute of type **deliver_in_progress** to the integration stream. If another developer is in the process of delivering work to the same stream, the **mkattr** command fails and the script displays a message suggesting that the developer try again later. Otherwise, the **mkattr** command succeeds and prevents other developers from delivering to the integration stream until the current deliver operation finishes.

```
use Config;

my $PVOBTAG;
my $tempfile;
my $exit_value;

if ($Config{'osname'} eq 'MSWin32') {
   $PVOBTAG = '\cyclone-pvob';
   $tempfile = $ENV{TMP}."\\expreop.".$ENV{"CLEARCASE_PPID"}.".txt";
}
else {
   $PVOBTAG = '';
}

my $STREAM = "stream:".$ENV{"CLEARCASE_STREAM"};
my $ATTYPE = "attype:deliver_in_progress\@$PVOBTAG";

print $STREAM."\n";
print $ATTYPE."\n";

my $cmdline;
my $cmdoutput;

# Test to see if the deliver in progress attribute is
# applied to the target stream.
```

```
$msg = 'cleartool describe -fmt "%a" $STREAM';

print $msg."\n";

if ($ENV{"CLEARCASE_CMDLINE"} eq "") {
   open(TEMPFH, "> $tempfile");
   print TEMPFH $msg."\n";
   close(TEMPFH);
   $cmdline = "clearprompt text -outfile $tempfile -multi_line -dfile
\"$tempfile\" -prompt \"Describe Results\"";
   $cmdoutput = '$cmdline';
}

if (index($msg, "deliver_in_progress") >=0) {
   print "***\n";
   print "*** A deliver operation is already in progress. Please
try again later.\n";
   print "***\n";
   exit 1;
}
$cmdline = "cleartool mkattr -default $ATTYPE $STREAM";

print $cmdline."\n";
$msg = '$cmdline';

$exit_value = $? >> 8;

if (!($exit_value eq 0)) {
   print "***\n";
   print "*** A deliver operation was started just before yours.\n";
   print "*** That deliver operation is already in progress. Please
try again later.\n";
   print "***\n";
exit 1;
}
exit 0;
```

Delivery postoperation trigger script

This postoperation trigger action fires when a developer cancels or completes a deliver operation to the specified integration stream. This script removes the **deliver_in_progress** attribute that the preoperation script attaches to the integration stream at the start of the deliver operation. After the attribute is removed, another developer can deliver work to the integration stream.

```
# perl script that fires on deliver_complete or deliver_cancel postop
trigger.
use Config;

# define platform-dependent arguments.
my $PVOBTAG;
if   ($Config{'osname'} eq 'MSWin32') {
   $PVOBTAG = '\cyclone-pvob';
}
else{
   $PVOBTAG = '';
}
my $STREAM = "stream:".$ENV{"CLEARCASE_STREAM"};
my $ATTYPE = "attype:deliver_in_progress\@$PVOBTAG";

# remove the attribute to allow deliveries.
print `cleartool rmattr -nc $ATTYPE $STREAM`;
```

Send mail to developers on deliver operations

To improve communication among developers on your project team, you may want to create a trigger type that sends an e-mail message to team members whenever a developer completes a deliver operation. The following sections include scripts for detecting deliveries and notifying developers:

- Script 1 creates a trigger type that fires at the end of a successful deliver operation.
- Script 2 is the postoperation trigger action that sends e-mail messages to developers.

For information about sharing scripts, see "Sharing triggers among different types of platform" on page 131.

E-mail notification setup script

This script creates a postoperation trigger type that fires when a developer finishes a deliver operation, as represented by the **deliver_complete** opkind. The **mktrtype** command uses the **–stream** option to indicate that the trigger type applies only to deliver operations that target the specified integration stream.

```
# This is a Perl script to set up the triggertype
# for e-mail notification on deliver.
use Config;

# define platform-dependent arguments.
my $PVOBTAG;
if   ($Config{'osname'} eq 'MSWin32') {
   $PVOBTAG  = '\cyclone_pvob';
   $WCMD     = '-execwin "ccperl
\\\\pluto\disk1\ucmtrig_examples\ex2\ex2_postop.pl"';
}
else {
   $PVOBTAG  = '/pvobs/cyclone_pvob';
   $WCMD     = '-execwin "ccperl
\\\\\\pluto\disk1\ucmtrig_examples\ex2\ex2_postop.pl"';
}
my $STREAM = "stream:P1_int\@$PVOBTAG";
my $TRTYPE = "trtype:ex2_postop\@$PVOBTAG";
my $UCMD   = '-execunix "Perl
/net/pluto/disk1/ucmtrig_examples/ex2/ex2_postop.pl"';

print 'cleartool mktrtype -ucmobject -all -postop deliver_complete
$WCMD $UCMD -stream $STREAM -nc $TRTYPE`;
```

E-mail notification postoperation trigger script

This postoperation trigger action fires when a developer finishes delivering work to the integration stream. The script composes and sends an e-mail message to other developers on the project team telling them that a deliver operation has just finished. The script uses Rational ClearCase environment variables to provide the following details about the deliver operation in the body of the message:

- Project name
- Development stream that delivered work
- Integration stream that received delivered work
- Integration activity created by the deliver operation
- Activities delivered
- Integration view used by deliver operation

```
# Perl script to send mail on deliver complete.

#################################################################
# Simple package to override the "open" method of Mail::Send so we
# can control the mailing mechanism.
package SendMail;

use Config;
use Mail::Send;

@ISA = qw(Mail::Send);

sub open {
   my $me = shift;
   my $how;  # How to send mail
   my $notused;
   my $mailhost;

   # On Windows use SMTP

   if ($Config{'osname'} eq 'MSWin32') {
   $how = 'smtp';
   $mailhost = "localmail0.company.com";
   }

   # else use defaults supplied by Mail::Mailer

   Mail::Mailer->new($how, $notused, $mailhost)->open($me);
}

#
#################################################################
# Main program

   my @to = "developers\@company.com";
   my $subject = "Delivery complete";

   my $body = join '', ("\n",
      "UCM Project: ", $ENV{CLEARCASE_PROJECT}, "\n",
      "UCM source stream: ", $ENV{CLEARCASE_SRC_STREAM}, "\n",
      "UCM destination stream: ", $ENV{CLEARCASE_STREAM}, "\n",
      "UCM integration activity: ", $ENV{CLEARCASE_ACTIVITY}, "\n",
      "UCM activities delivered: ", $ENV{CLEARCASE_DLVR_ACTS}, "\n",
      "UCM view: ", $ENV{CLEARCASE_VIEW_TAG}, "\n"
   );

   my $msg = new SendMail(Subject=>$subject);

   $msg->to(@to);
   my $fh  = $msg->open($me);
   $fh->print($body);
   $fh->close();
   1; # return success
#
#################################################################
```

Do not allow activities to be created on the integration stream

Anyone who has an integration view attached to the integration stream can create activities on that stream, but the UCM process calls for developers to create activities in their development streams. You may want to implement a policy that prevents developers from creating activities on the integration stream inadvertently. This section shows a Perl script that enforces that policy.

For information about sharing scripts, see “Sharing triggers among different types of platform” on page 131.

The following **mktrtype** command creates a preoperation trigger type called **block_integration_mkact**.

```
cleartool mktrtype -ucmobject -all -preop mkactivity -execwin "ccperl ^
\\pluto\disk1\triggers\block_integ_mkact.pl" -execunix "Perl ^
/net/jupiter/triggers/block_integ_mkact.pl"
block_integration_mkact@\my_pvob
```

The trigger type fires when a developer attempts to make an activity.

The following preoperation trigger script runs when the **block_integration_mkact** trigger fires.

```
# Get the integration stream name for this project
my $istream = 'cleartool lsproject -fmt "%[istream]p"
$ENV{"CLEARCASE_PROJECT"}';

# Get the current stream and strip off VOB tag
$_ = $ENV{"CLEARCASE_STREAM"};
s/\@.*//;
my $curstream = $_;
# If it's the same as our stream, then it is the integration stream
if   ($istream eq $curstream) {
   # Only allow this mkact if it is a result of a deliver
   # Determine this by checking the parent op kind
   if ($ENV{"CLEARCASE_POP_KIND"} ne "deliver_start") {
      print "Activity creation is only permitted in integration
streams for
      delivery.\n";
      exit 1
   }
}

exit 0
```

The script uses the **cleartool lsproject** command and the CLEARCASE_PROJECT environment variable to determine the name of the project’s integration stream. An integration activity is created to keep track of changes that occur during a deliver operation. The script uses the CLEARCASE_POP_KIND environment variable to determine whether the activity being created is an integration activity. If the **mkactivity** operation is the result of a deliver operation, the value of CLEARCASE_POP_KIND, which identifies the parent operation, is **deliver_start**.

If the value of CLEARCASE_POP_KIND is not **deliver_start**, the activity is not an integration activity, and the script disallows the **mkactivity** operation.

Implementing a role-based access control system

In a Rational ClearCase environment, where users perform different roles, you may want to restrict access to certain Rational ClearCase operations based on role. You can use a trigger definition and script that implement a role-based access control system.

For information about sharing scripts, see “Sharing triggers among different types of platform” on page 131.

The following **mktrtype** command creates a preoperation trigger type called **role_restrictions**.

```
cleartool mktrtype -nc -ucmobject -all -preop mkstream,mkbl,mkactivity \
-execunix "perl  /net/jupiter/triggers/role_restrictions.pl" \
-execwin  "ccperl \\pluto\disk1\triggers\role_restrictions.pl" \
role_restrictions@\my_pvob
```

The trigger type fires when a user attempts to make a baseline, stream, or activity.

Role-based preoperation trigger script

The following preoperation trigger script maps users to the following roles:

- Project manager
- Integrator
- Developer

```
use strict;

sub has_permission
{
   my ($user,$op,$pop,$proj) = @_;

   #When performing a composite operation like 'deliver' or 'rebase',
   #we don't need to check permissions on the individual sub-operations
   #that make up the composite.

   return 1 if($pop eq 'deliver_start' || $pop eq 'rebase_start' ||
          ($pop eq 'deliver_complete' || $pop eq 'rebase_complete' ||
          ($pop eq 'deliver_cancel' || $pop eq 'rebase_cancel');

   # Which roles can perform what operations?
   # Note that these maps can be stored in a Rational ClearCase attribute
   # on each project instead of hard-coded here in the trigger script
   # to give true per-project control.

   my %map_op_to_roles = (
      mkactivity => [ "projectmgr", "integrator", "developer" ],
      mkbl       => [ "projectmgr", "integrator" ],
      mkstream   => [ "projectmgr", "integrator", "developer" ],
   );

   # Which users belong to what roles?

   my %map_role_to_users = (
      projectmgr => [ "kate" ],
      integrator => [ "kate", "mike" ],
      developer  => [ "kate", "mike", "jones" ],
   );

   # Does user belong to any of the roles that can perform this
operation?

   my ($role,$tmp_user);

   for $role (@{ $map_op_to_roles{$op} }) {
      for $tmp_user (@{ $map_role_to_users{$role} }) {
         if ($tmp_user eq $user) {
            return 1;
         }
      }
   }

   return 0;
}
sub Main
{
   my $user = $ENV{CLEARCASE_USER};
   my $proj = $ENV{CLEARCASE_PROJECT};
   my $op   = $ENV{CLEARCASE_OP_KIND};
```

```
    my $pop  = $ENV{CLEARCASE_POP_KIND};

    my $perm = has_permission($user, $op, $proj);

    printf("$user %s permission to perform '$op' in project $proj\n",
       $perm ? "has" : "does NOT have");

    exit($perm ? 0 : 1);
}

Main();
```

The script maps the **mkactivity**, **mkbl**, and **mkstream** operations to the roles that are permitted to perform them. For example, only users designated as project managers or integrators can make a baseline.

The script uses the CLEARCASE_USER environment variable to retrieve the user's name, the CLEARCASE_OP_KIND environment variable to identify the operation the user attempts to perform, and the CLEARCASE_POP_KIND environment variable to identify the parent operation. If the parent operation is **deliver** or **rebase**, the script does not check permissions.

Additional uses for UCM triggers

The examples shown in sections "Enforce serial deliver operations" on page 133 through "Implementing a role-based access control system" on page 138 represent just a few ways that you may use UCM triggers to enforce development policies. Other uses for UCM triggers include the following:

- Create an integration between UCM and a change request management (CRM) system. Although most customers can use the UCM integration with Rational ClearQuest, you may want to integrate with another CRM system. To accomplish this, you could perform the following steps:
  - Create a trigger type on **mkactivity** that creates a corresponding record in the CRM database when a developer makes a new activity.
  - Create a trigger type on **setactivity** that transitions the record in the CRM database to a scheduled state when a developer starts working on an activity.
  - Create a trigger type on **deliver** that transitions the record in the CRM database to a completed state when a developer finishes delivering the activity to the integration stream.
- Create a trigger type on **rebase** that prevents developers from rebasing certain development streams. You may want to enforce this policy on a development stream that is being used to fix one particular bug.
- Create a trigger type on **setactivity** that allows specific developers to work on specific activities.

Chapter 9. Managing multiple projects

Chapters Chapter 2 through Chapter 8 discuss the management of a single UCM project. This chapter discusses the uses and management of multiple UCM projects.

Project uses

Generally, multiple-project organization falls into one of two categorizations: release-oriented and component-oriented.

Release-oriented projects

A project team can organize its work for product releases. First, the team might work on Release 1 of the product. To work on Release 2, the team branches off Release 1 and begins new development work in the Release 2 stream, and, when Release 3 work begins, it might branch off the Release 2 stream.

After Release 1 ships to end users, patches for the release might be developed in its own project, which branches off Release 1. The patch project delivers its work to Release 2, so that the bug fixes can be incorporated into the new release.

This type of project organization results in a cascade of branches that can cause some difficulty. To avoid the difficulty, a slightly different project organization is generally recommended for release-oriented projects (see Figure 41).

In the release-oriented organization shown in Figure 41, all projects start from a foundation baseline in the **Mainline** project. The **Webotrans_1.0** project delivers its release to the **Mainline** project. A patch release project **Rel_1_Patch** can be started from a stable baseline in the project **Webotrans_1.0** and can deliver its work to the **Mainline** project integration stream. Instead of cascading from the previous release, a follow-on project **Webotrans_2.0** is then started from the baseline in the integration stream of the **Mainline** project.

A release-oriented project must have modifiable access to all of the components that are contained in the final product. Developers working on one component of the project need to consistently and frequently coordinate their work with developers working on other components. This type of project organization in UCM works well when there is tight coupling between components.

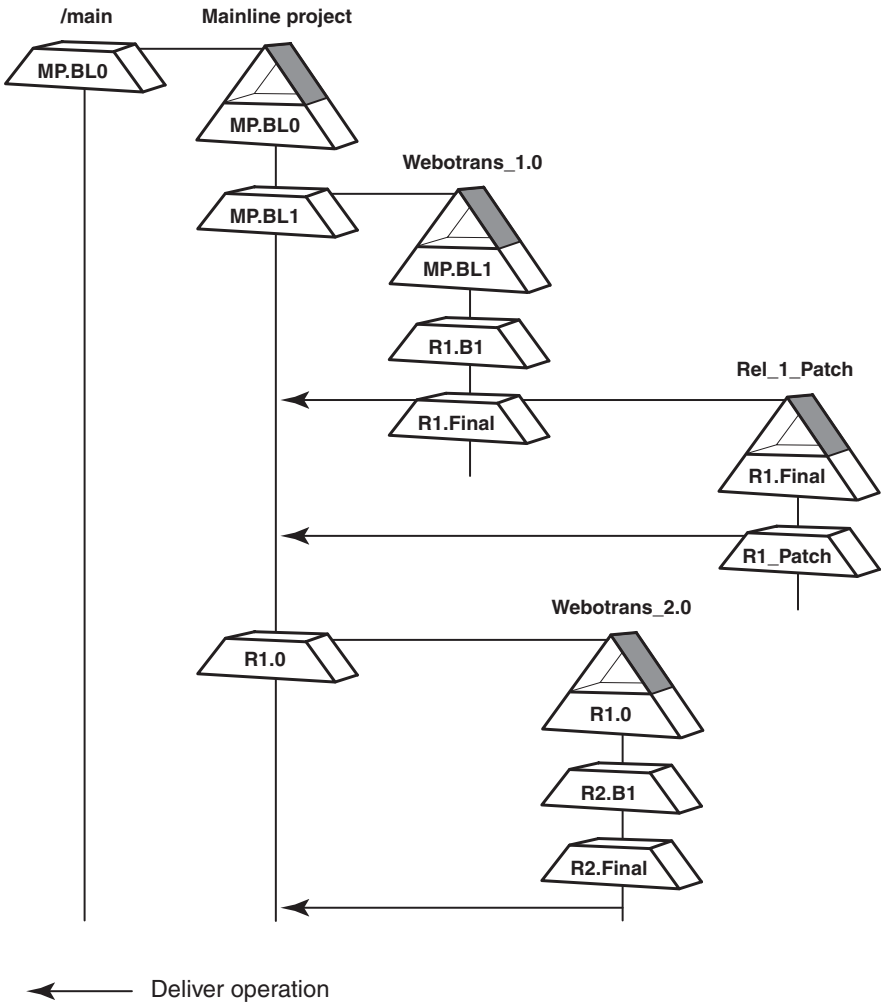


Figure 41. An organization for release-oriented projects

Using a mainline project

If you anticipate that your team will develop and release numerous versions of your system, create a mainline project (see Figure 41). A mainline project serves as a single point of integration for related projects over a period of time. It is not specific to any single release.

For example, assume the Webotrans team plans to develop and release new versions of their product every six months. For each new version, the project manager could create a project whose foundation baselines are the final recommended baselines in the prior project's integration stream. For example, the foundation baselines of Webotrans 2.0 are the final recommended baselines in the Webotrans 1.0 integration stream; the foundation baselines for Webotrans 3.0 are the final recommended baselines in the Webotrans 2.0 integration stream, and so on. This approach is referred to as a cascading projects design. The disadvantage to this approach is that you must look at all integration streams to see the entire history of the Webotrans projects.

In the mainline project approach, the Webotrans project manager creates a mainline project with an initial set of baselines, and then creates Webotrans 1.0 based on those initial baselines. When developers finish working on Webotrans 1.0, the

project manager delivers the final recommended baselines to the mainline project's integration stream. These baselines in the mainline project's integration stream serve as the foundation baselines for the Webotrans 2.0 project. When the Webotrans 2.0 team finishes its work, the project manager delivers the final recommended baselines to the mainline project integration stream, and so on. The advantage to this approach is that each project final recommended baselines are available in the mainline project integration stream.

Composite baselines in release-oriented projects

In release-oriented projects, use composite baselines to shorten recommended baseline lists. Using composite baselines makes it easier to tell which baselines were recommended at different times. The composite baselines can be used to represent major subsystems in the product. Or, in the simplest case, a project can consolidate all of its baselines into a single component, which it recommends and delivers to follow-on projects.

Additionally, using a single composite baseline to represent the final product (that is, collecting baselines from all of the components) reduces the occurrence of baseline conflicts. If a project uses a single composite baseline for the final product, the project integrator can implement a process rule that forces all builds to occur from the top-level component. This process rule reduces the probability of a developer inadvertently introducing conflicts.

Developers can still choose to use baseline overrides when they need to access materials that are not included in the recommended baseline set. For example, if a developer needs a bug fix that has not yet been included in a recommended baseline, the developer can rebase to the appropriate baseline for the specific component containing the bug fix, even if there is no baseline conflict.

Take care when you use baseline overrides to share code in this way, because you can introduce a conflict into a stream. Baseline conflicts can occur when more than one composite baseline is overridden — but only when the overridden baselines are composite baselines. Baseline conflicts cannot be created if a single composite baseline is overridden or if baselines of components that are not members of other components are overridden.

Component-oriented Projects

Development teams can organize their work in terms of reusable assets. These teams use projects to create individual components and composite baselines that select baselines of several components. Low-level or core components can be used to construct mid-level components or subsystems, until the highest level components are integrated into a product.

The goal of a component-oriented project is to produce a composite baseline or a set of baselines that represents the integration of the shared components into a subsystem (see Figure 42).

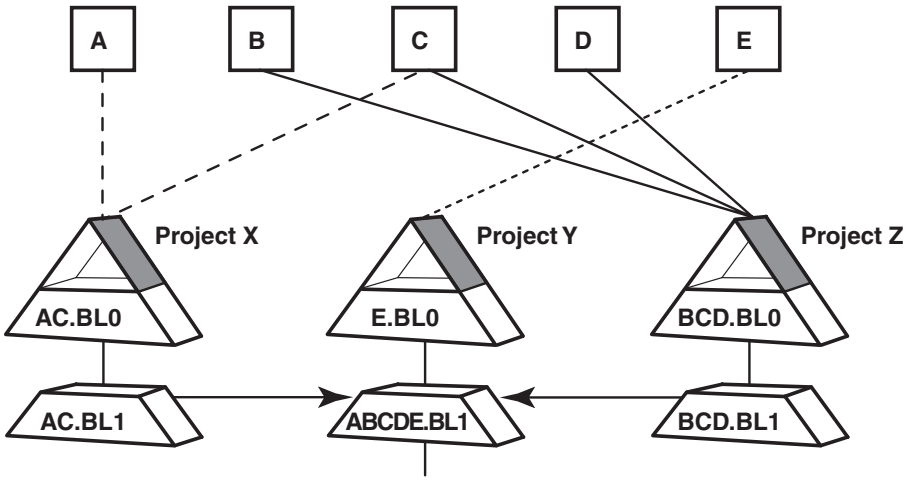


Figure 42. Structure for component-oriented projects

In Figure 42, project **X** uses components **A** and **C** to produce a set of baselines that define the X subsystem, and project **Z** uses components **B**, **C**, and **D** to implement the Z subsystem. Project **Y** uses the subsystems created by the **X** and **Z** projects, represented by the baselines that these projects produce. Additionally, project **Y** does custom work in component **E**.

The key attribute of managing the configuration is that all the code of a subsystem is released together. To the development team, the important factor is that the subsystem represents assets that can be easily reused. Designing a product, or a family of products based on subsystems allows the development effort to be divided into logical units. This organization simplifies the development efforts, allows better management for risk, and provides opportunities for code reuse.

The key aspect of this organization is that each project has access to two classes of components: modifiable components and read-only components. Each project does its development work on modifiable components, and only one project can modify these components. The read-only components are shared, but projects do not plan to modify the read-only components, because that violates the sharing model. Accordingly, these shared components are specified to be read-only in the project.

The teams in a development group with this type of project organization are restricted to a limited number of components that they can modify. Because the lower level components can be shared, the changes must be made in a central, compatible manner, in the project dedicated to that component. For example, if project **Y** needs changes in the **A** and **C** components, that work must be done in project **X**.

In a component-oriented organization, each project has more freedom in selecting baselines of shared components. Because these shared components are not modified by the project, theoretically the project should be able to change from one version of a shared component to any other version of that component. The component-oriented project organization works well when components are loosely coupled with well-defined interfaces. This type of project organization tends to promote component reuse more effectively than the release-oriented project organization does.

Composite baselines in component-oriented projects

In component-oriented projects, significant savings can be realized by modeling subsystems with composite baselines. For example, projects that integrate the subsystems can rebase to a single baseline to configure a subsystem.

When composite baselines are used to model subsystems, they are logical components built of smaller components. The components used in constructing a subsystem are either shared (read-only) or modifiable (they contain the work particular to that subsystem). The important property of the shared components is that they are non-modifiable. If a subsystem requires changes to a shared component, that requirement can interfere with the ability to share that component. The modifiable, custom components store the code that is needed to integrate the shared components and to provide the unique functions of the subsystem (see Figure 43).

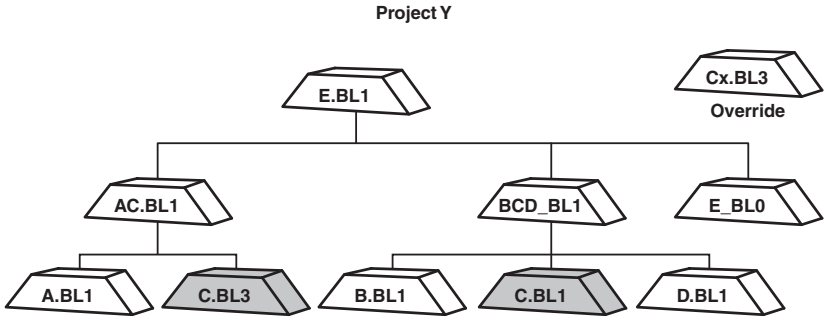


Figure 43. Composite baselines representing subsystems

Project **Y** is composed of the custom component **E** and the components **AC** and **BCD**, which both share component **C**. It does not matter whether baseline **E.BL1** is a product, a .DLL, a lower-level library, or another component.

In the composite baseline **E.BL1**, a component-oriented approach to development can be more prone to baseline conflicts than a release-oriented approach. The frequency of baseline conflicts depends on the following factors.

- The number of shared components
- The processes in the development organization
- The coordination among the different teams developing subsystems

The more closely the teams that produce each component coordinate their work, the less likely conflicts will occur.

If development teams need complete freedom in selecting baselines of the components that they use, conflicts are more likely. Conflicts occur because of the difficulty in ensuring that a random baseline of one component, for example, **AC**, will work with a random baseline of another component.

Baseline **E.BL1** can be considered as consuming baselines **AC.BL1** and **BCD.BL1**. However, it is not a true producer and consumer relationship. Composite baseline **E.BL1** is not using the **AC.BL1** baseline; it is using **AC.BL1** plus the override, baseline **Cx.BL3** (see Figure 43). Therefore, baseline **E.BL1** is not actually using the product of project **X**. The **X** project does produce **AC** baselines, but, in a situation with conflicts, the baseline is a guideline rather than a rule for projects that need the **AC** component.

If these subsystems were in a release-oriented organization, composite baselines would represent a tight coupling of baselines, for example, baselines **A.BL1** and **C.BL3** shall be used to make baseline **AC.BL1**. But in a component-oriented organization, composite baselines represent a looser coupling, that is, baselines **A.BL1** and **C.BL3** should be used to make baseline **AC.BL1**. But the project integrator can chose to override the coupling between baselines. Therefore, in a component-oriented organization of projects, a composite baseline is more of an indication of the components that should be used to create a subsystem rather than a requirement to use them.

Therefore, selecting a baseline override in a component-oriented organization needs to be careful and deliberate. Project integrators need to be aware that, in selecting a baseline override, they are changing the decisions made by the project that produced the component. There might be specific reasons why the **BCD** component is compatible with the **C.BL1** baseline, for example, a different baseline could cause the component to fail. Often, using a descendant of baseline **C.BL1** is successful, but the selection of override baselines is not restricted in the ClearCase environment. There is no guarantee of the relationship between baseline **C.BL1** and the override baseline **Cx.BL3**.

As the time approaches for a component to be completed, the use of baseline overrides can be destabilizing to a product, because they can represent significant code differences. These differences can be managed by coordinating the work of the projects that produce each subsystem. As the time for completing a component approaches, the teams can agree on the lower-level components so that conflicts are reduced.

Bootstrap projects

If baseline relationships are known at the start of development and they do not change much after that, you can use a bootstrap project to configure other projects with a single composite baseline as the foundation baseline.

When you create a composite baseline for a project, the foundation baselines of the integration stream are not affected. After the creation of the composite baseline, the foundation of the stream still contains the baselines of the individual member components. For development streams in the project to appropriately use the composite baseline, you simply recommend just the composite baseline in the integration stream. If all the member baselines remain listed in the foundation of the integration stream, individual developers who have to rebase to the recommended baseline can be confused by multiple baselines being listed.

To reduce the confusion, set up a special project to “bootstrap” the composite baseline for other projects. The bootstrap project includes all of the components in its foundation. You create the composite baseline in the bootstrap project and then create a development project with this initial composite baseline as its foundation.

As a result, in the development project, the integration stream and the development streams have similar foundation sets. The project integrators do not have to keep track of the individual baselines, since the project can start with a single, all-inclusive composite baseline. Developers in the projects that follow are less likely to be confused when they select a baseline during a rebase operation.

Mixing project organizations

The strategy for project organizations is usually dictated by the size, structure, and philosophy of the development team. UCM supports both release-oriented and

component-oriented projects. You can mix the two strategies, so that some projects produce certain components, and other projects integrate components into a release.

About managing multiple projects

Most project managers and project integrators manage a single project. However, you may need to manage multiple releases of a project simultaneously. To do so, you need to merge changes from one project to another. You can accomplish that merging in the following common scenarios:

- Managing a current project and a follow-on project simultaneously
- Migrating unfinished work to a follow-on project
- Incorporating a patch release into a new release of the project
- Delivering work to another project
- Sharing baselines between sibling streams

You can also use base ClearCase tools to merge work from a UCM project to a base ClearCase project (see "Merging from a project to a non-UCM branch" on page 152).

Managing a current project and a follow-on project simultaneously

Given the tight software development schedules that most organizations operate within, it is common practice to begin development of the next release of a project before work on the current release is completed. The next release may add new features, or it may involve porting the current release to a different platform.

Figure 44 illustrates the flow of a current project, Webotrans 4.0, and a follow-on project, Webotrans 4.1.

In this example in Figure 44, note the following points:

- The project manager for the follow-on project created the Webotrans 4.1 project based on the **Beta** baselines of the components used in the Webotrans 4.0 project. Developers on both project teams then continued to make changes, and the 4.0 and 4.1 integrators continued to create new baselines that incorporate those changes.
- When the 4.0 team completed its work, the integrator created the final baselines, named FCS. The 4.1 project manager then rebased the 4.1 integration stream to the FCS baselines.

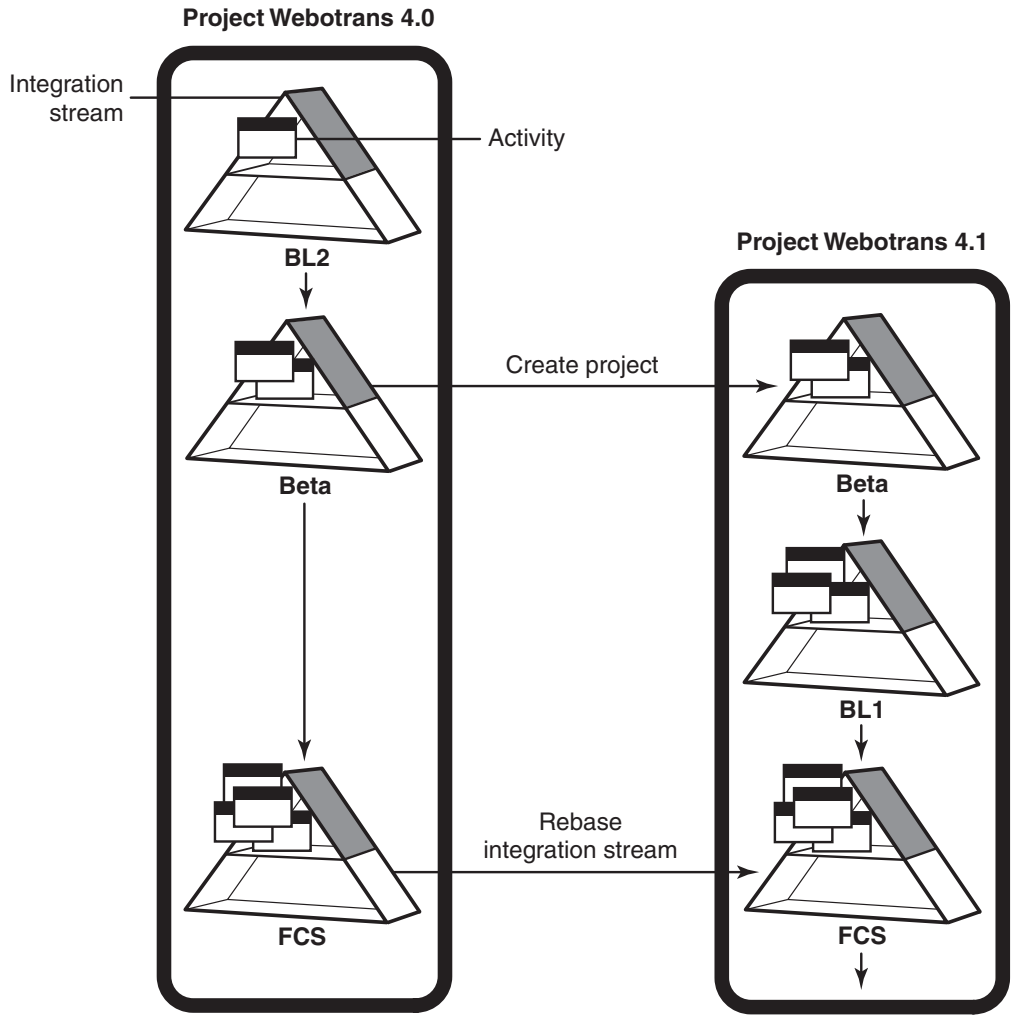


Figure 44. Managing a follow-on release

To rebase an integration stream to baselines of another project

1. In Project Explorer, select the integration stream that you want to rebase.
2. Click **Tools > Rebase Stream**.
3. In the Rebase Stream Preview window, click **Advanced**.
4. In the Change Rebase Configuration window, select a component that contains the baseline you want to use to rebase your stream. Click **Change**.
5. In the Change Baseline window:

   On the Windows system, click **Change.**

   On Linux and the UNIX system, click the arrow at the end of the **From Stream** field.
6. On the Windows system, in the Choose Stream window, navigate to the integration stream of the other project. Select the integration stream and click **OK**.

   On Linux and the UNIX system, select the integration stream of the other project.

This updates the Change Baseline window with the set of baselines available in the other project integration stream.

7. In the Change Baseline window, select the component. The **Baselines** list displays all baselines available for the selected component in the other project's integration stream. Select the baseline to which you want to rebase your integration stream. Click **OK**. The baseline that you selected now appears in the Change Rebase Configuration window.
8. Repeat Step 4 on page 148 through Step 7 on page 149 until you finish selecting the set of baselines to which you want to rebase your integration stream.
9. Click **OK** to close the Change Rebase Configuration window. Click **OK** in the Rebase Stream Preview window.
10. All nonconflicting changes are merged automatically. If there are conflicting changes, a prompt asks you whether to start Diff Merge, a tool with which you resolve conflicting changes. For details on using Diff Merge, see the Diff Merge Help and *Developing Software* online help.

Tip: You can rebase your project's integration stream only if the baseline to which you are rebasing is a successor of the current *foundation baseline* of your integration stream. In the previous example, the FCS baseline is a successor to the Beta baseline, which is the current foundation baseline for the Webotrans 4.1 integration stream.

Migrating unfinished work to a follow-on project

A development stream in one project can deliver its activities to a cousin stream in another project (see Figure 45).

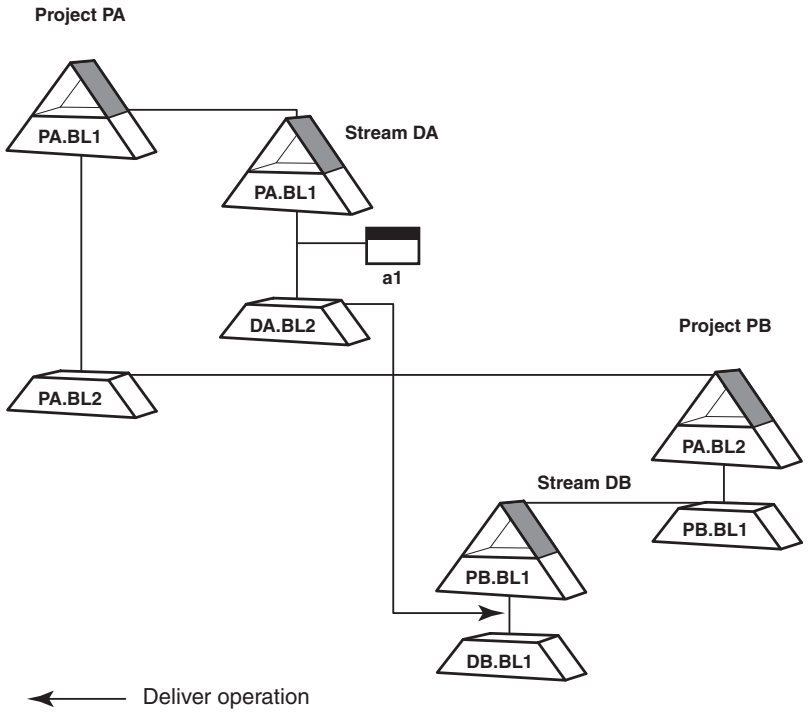


Figure 45. Alternate target inter-project deliver operation

Work is in progress in project **PA**, but must be delivered before work in stream **DA** can be completed. Follow-on work continues in project **PB** which starts from

recommended baselines **PA.BL2** in the integration stream of project **PA**. (The integration streams in projects **PA** and **PB** are siblings because they share a parent, project **PA**.)

You can migrate the changes in activity **a1** to a cousin stream **DB** in project **PB** by using an alternate target deliver operation. Only the changes in activity **a1** are delivered because the remaining contents of stream **DA** are in stream **DB**. Work on the feature that started in stream **DA** can be continued in stream **DB**.

Incorporating a patch release into a new version of the project

A common development scenario with multiple projects involves working on a patch release and a new release of a project at the same time. Figure 46 illustrates the flow of a patch release and a new release.

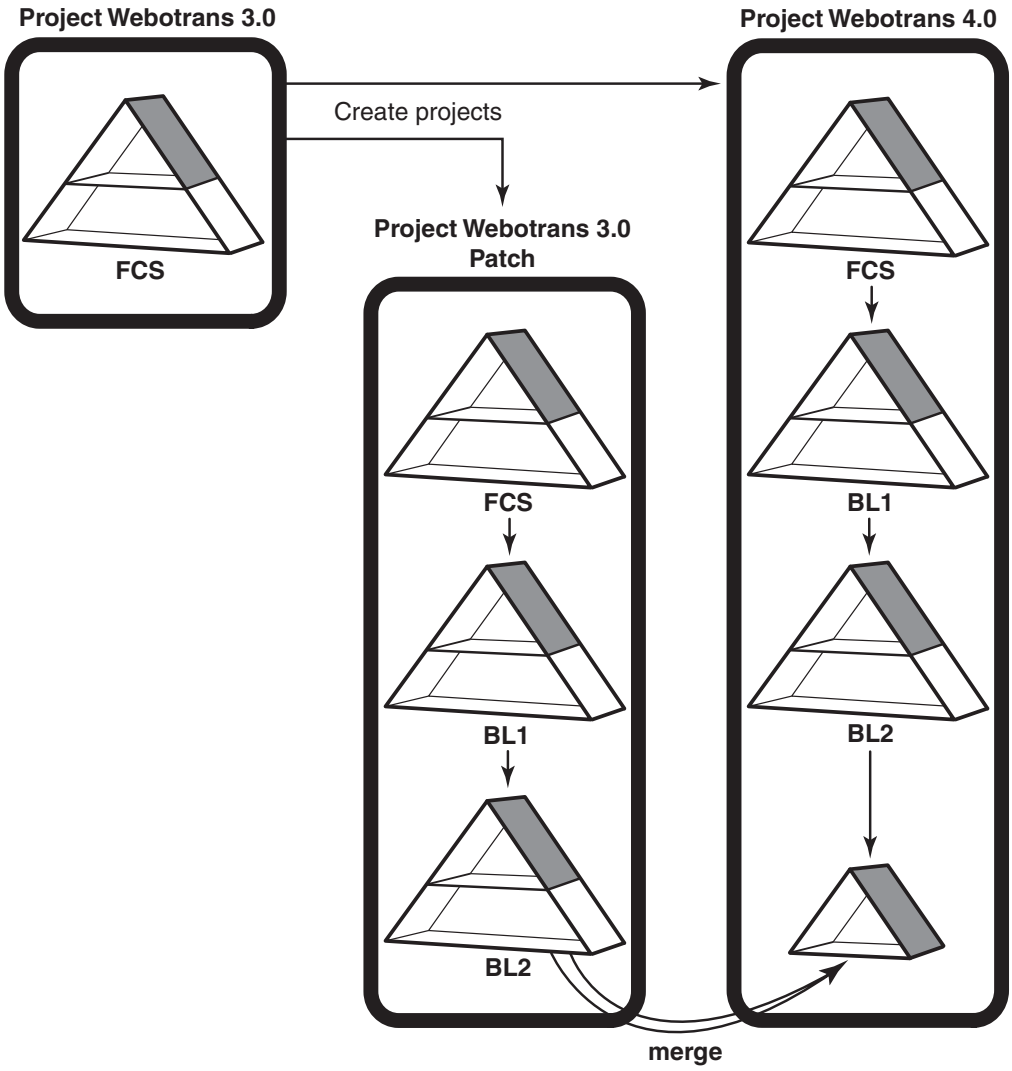


Figure 46. Incorporating a patch release

In this example shown in Figure 46:

- Both the Webotrans 3.0 Patch and Webotrans 4.0 projects use the **FCS** baselines of the components in the Webotrans 3.0 project as their foundation baselines.

The purpose of the patch release is to fix a problem detected after Webotrans 3.0 was released. Webotrans 4.0 represents the next major release of the Webotrans product.

- Development continues in both the 3.0 Patch and 4.0 projects, with the integrators creating baselines periodically.
- The developers working on the 3.0 Patch project finish their work, and the integrator incorporates the final changes in the **BL2** baseline. The integrator then needs to deliver those changes from the 3.0 Patch integration stream to the 4.0 integration stream so that the 4.0 project contains the fix.

Delivering work from an integration stream to another project

You can deliver work from an integration stream in one project to an integration stream in another project. When you deliver work from an integration stream, you must deliver baselines.

To deliver work between integration streams

1. In the source stream, make one or more baselines that incorporate the changes you want to deliver.
2. Check the deliver policy settings for the target integration stream to confirm that it allows deliveries from other projects. In the Project Explorer, select the target integration stream, and click **File > Policies**. If the **Allow interproject deliver to project or stream** policy is not enabled, ask the project manager to change the setting to **enabled**.
3. In the Project Explorer, select the source integration stream, and click **Tools > Deliver Baselines To Default** or **Deliver To Alternate Target**. To determine the default deliver target for the integration stream, select the stream and click **File > Properties**. The **Deliver to** box on the **General** tab identifies the default deliver target. You can change the default deliver target by clicking **Change**. The **Deliver To Alternate Target** option opens the Deliver from Stream (alternate target) window, which lets you select the target stream.
4. In the Deliver from Stream Preview window, use **Add**, **Change**, and **Remove** to select the baselines that you want to deliver. Make sure that the **View** box identifies a view that is attached to the target integration stream. If necessary, click **Change** to select a different view. Click **OK** to start the merge part of the deliver operation.
5. All nonconflicting changes are merged automatically. If there are conflicting changes, a prompt asks you whether to start Diff Merge, a tool with which you resolve conflicting changes. For details on using Diff Merge, see the Diff Merge Help and *Developing Software* online help.
6. When you finish merging files, test the result. When the testing is complete, click **Complete** to check in the changes.

Sharing baselines between sibling streams in different projects

Baselines can be used with rebase and deliver operations to configure streams with changes from related streams in different projects (Figure 47).

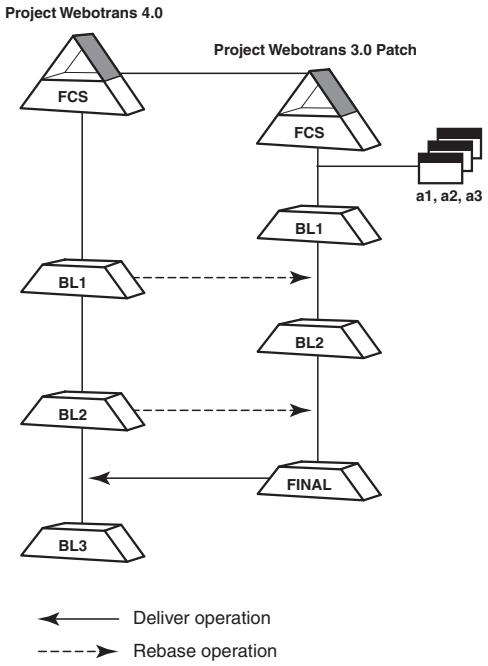


Figure 47. Baselines distributed to a different project

The integrators rebase the integration stream in the **Webotrans 3.0 Patch** project to baselines **BL1** and **BL2** from the **Webotrans 4.0** project. Rebasing in this situation allows the integrators to test and validate the patch with ongoing development in the follow-on project. When the integrators deliver the patch changes in the **FINAL** baseline, the process of testing and validating the patch in **Webotrans 4.0** integration stream is made much easier.

Note: From an integration stream, you can deliver only baselines and not individual activities.

Merging from a project to a non-UCM branch

You may be in a situation in which part of the development team works in a UCM project, and the rest of the team works in base ClearCase. If you are a longtime Rational ClearCase user, you may decide to use UCM initially on a small part of your system. This approach would allow you to migrate from base ClearCase to UCM gradually, rather than all at once.

In this case, you need to merge work periodically from the integration stream of the project to the branch that serves as the integration branch for the system. To do so, use a script similar to the one shown here, which uses base ClearCase functionality to merge changes.

```
# Sample Perl script for delivering contents of one UCM project to
# a nonUCM project. Run this script while set to a view that sees the
# destination branch.
#
# Usage: Perl <this-script> <project-name> <project-vob>

use strict;

my $mergeopts = '–print';
my $project = shift @ARGV;
my $pvob = shift @ARGV;
my $bl;
```

```
chdir ($pvob) or die("can't cd to project VOB '$pvob'");

print("######## Getting recommended baselines for project
'$project'\n");
my @recbls = split(' ', 'cleartool lsproject –fmt "%[rec_bls]p"
$project');

foreach $bl (@recbls) {

   my $comp = 'cleartool lsbl –fmt "%[component]p" $bl';
   my $vob = 'cleartool lscomp –fmt "%[root_dir]p" $comp';

   print("######## Merging changes from baseline '$bl' of $vob\n");

   my $st = system("cleartool findmerge $vob –fver $bl $mergeopts");
   $st == 0 or die("findmerge error");
}

exit 0;
```

The script finds the recommended baselines for the integration stream from which you are merging. It then uses the **cleartool findmerge** command to find differences between the versions represented by those recommended baselines and the latest versions in the target branch. For details, see the **findmerge** reference page.

You can add error handling and other logic appropriate for your site to this script before using it.

Part 3. Working in base ClearCase

Chapter 10. Managing projects in base ClearCase

This chapter describes base ClearCase project management.

About base ClearCase project management

A project manager is responsible for planning, staffing, and managing the technical aspects of a software development project. You decide what will be worked on, assign work to team members of the project, establish the work schedule, and perhaps the policies and procedures for doing the work.

When development is underway, you monitor progress and generate project status reports. You may also approve the specific work items included in a build and subsequently a baseline.

You may also be the project integrator, responsible for incorporating work that each developer completes into a deliverable and buildable system. You create the project's baselines and establish the quality level of those baselines.

In the base ClearCase configuration, many features are offered to make this work easier. Before development begins, you need to complete several planning and setup tasks:

- Setting up the project environment
- Implementing development policies
- Defining and implementing an integration policy

This chapter introduces these topics. The remaining chapters cover the implementation details. Chapter 16, "Using Rational ClearCase throughout the development cycle," on page 241, follows a project throughout the development cycle to show how you can use Rational ClearCase features.

Before reading project management, read *Developing Software* online help to become familiar with the concepts of VOBs, views, and config specs.

Setting up the project

You need to do planning and setup work before development begins.

Creating and populating VOBs

If your project is migrating to Rational ClearCase version control from another version control product or is adopting a configuration and change management plan for the first time, you must populate the VOBs for your project with an initial collection of data (file and directory elements). If your site has a dedicated Rational ClearCase administrator, he or she may be responsible for creating and maintaining VOBs, but not for importing data into them.

The *IBM Rational ClearCase Administrator's Guide* includes detailed information on these topics.

Planning a branching strategy

Branches are used to enable parallel development. A *branch* is an object that specifies a linear sequence of versions of an element. Every element has one *main branch*, which represents the principal line of development, and may have multiple subbranches, each of which represents a separate line of development. For example, a project team can use two branches concurrently: the **main** branch for new development work and a subbranch to fix a bug. The aggregated **main** branches of all elements constitutes the **main** branch of a code base.

Subbranches can have subbranches. For example, a project team designates a subbranch for porting a product to a different platform; the team then decides to create a bug-fixing subbranch off that porting subbranch. In a Rational ClearCase configuration, you can create complex branch hierarchies, for example, a multilevel branching hierarchy like that shown in Figure 1 on page 4. As a project manager in such an environment, you need to ensure that developers are working on the correct branches. To do that, you must tell them which rules to include in their *config specs* so that their views access the appropriate set of versions.

Chapter 11, "Defining project views," on page 163, describes config specs and branches in detail. Before you read it, a little background on branching strategies may be helpful.

Branching policy is influenced by the development objectives of the project and provides a mechanism to control the evolution of the code base. There are as many variations of branching policy as organizations that use Rational ClearCase version control. But there are also similarities that reflect common adherence to best practices.

Some of the more common branching types and uses are:

- Task branches

  Are short-lived, typically involve a small percentage of files, and are merged into their parent branch after the task is completed. Task branches promote accountability by leaving a permanent audit trail that associates a set of changes with a particular task; they also make it easy to identify the task artifacts, such as views and derived objects, that can be removed when they are no longer needed. If individual tasks do not require changes to the same files, it is easy to merge a task branch to its parent.
- Private development branches

  Are useful when a group of developers need to make a more comprehensive set of changes on a common code base. By branching as much of the **main** branch as needed, developers can work in isolation as long as necessary. Merging back to the **main** branch can be simplified if, before merging, each developer merges the **main** branch to the private branch to resolve any differences there before checking in the changed files.
- Integration branches

  Provide a buffer between private development branches and the **main** branch and can be useful if you delegate the integration task to one person, rather than making developers responsible for integrating their own work.

Branch names

It is a good idea to establish naming conventions that indicate the work the branch contains. For example, **rel2.1_main** is the branch on which all code for Release 2.1 ultimately resides, **rel2.1_feature_abc** contains changes specific to the ABC feature, and **rel2.1_bl2** is the second stable baseline of Release 2.1 code. (If necessary, branch names can be much longer and more descriptive, but long branch names can crowd a version tree display.)

Note: Make sure that you do not create a branch type with the same name as a label type. This can cause problems when config specs use labels in version selectors. For example, make all branch names lowercase, and make all label names uppercase.

Branches and Rational ClearCase MultiSite

Product Note: Rational ClearCase LT does not support Rational ClearCase MultiSite.

Branches are particularly important when your team works in VOBs that have been replicated to other sites with the Rational ClearCase MultiSite product. Developers at different sites work on different branches of an element. This scheme prevents collisions, for example, developers at two sites creating version `/main/17` of the same element. In some cases, versions of files cannot or should not be merged, and developers at different sites must share branches. For more information, see "Certain branches are shared among Rational ClearCase MultiSite sites" on page 187.

Creating shared views and standard config specs

As a project manager, you want to control the config specs that determine how branches are created when developers check out files. There are several ways to handle this task:

- Create a config spec template that each developer must use. Developers can either paste the template into their individual config specs or use the Rational ClearCase include file facility to get the config spec from a common source.
- Create a view that developers will share. This is usually a good way to provide an integration view for developers to use when they check in work that has evolved in isolation on a private branch.

  Note: Working in a single shared view can degrade system performance.
- To ensure that all team members configure their views the same way, you can create files that contain standard config specs. For example:
  - `/public/config_specs/ABC` contains the ABC team config spec
  - `/public/config_specs/XYZ` contains the XYZ team config spec

Store these config spec files in a standard directory outside a VOB, to ensure that all developers get the same version.

Recommendations for view names

You may want to establish naming conventions for views for the same reason that you do for branches: it is easier to associate a view with the task it is used for. The Rational ClearCase view-creation tools suggest appropriate view names, but you may want to use something different. For example, you can require all view names (called view tags) to include the owner's name and the task (**bill_V4.0_bugfix**) or the name of the computer hosting the view (**platinum_V4.0_int**).

Implementing development policies

To enforce development policies, you can create Rational ClearCase *metadata* to preserve information about the status of versions. To monitor the progress of the project, you can generate a variety of reports from this data and from the information captured in event records.

Using labels

A label is a user-defined name that can be attached to a version. Labels are a powerful tool for project managers and system integrators. By applying labels to groups of elements, you can define and preserve the relationship of a set of file and directory versions to each other at a given point in the development life cycle. For example, you can apply labels to these versions:

- All versions considered stable after integration and testing. Use this baseline label as the foundation for new work.
- All versions that are partially stable or contain some usable subset of functionality. Use this checkpoint label for intermediate testing or as a point to which development can be rolled back in the event that subsequent changes result in regressions or instability.
- All versions that contain changes to implement a particular feature or that are part of a patch release.

Using attributes, hyperlinks, triggers, and locks

Attributes are name/value pairs that allow you to capture information about the state of a version from various perspectives. For example, you can attach an attribute named **CommentDensity** to each version of a source file, to indicate how well the code is commented. Each such attribute can have the value **unacceptable**, **low**, **medium**, or **high**.

Hyperlinks allow you identify and preserve relationships between elements in one or more VOBs. This capability can be used to address process-control needs, such as requirements tracing, by allowing you to link a source file to a requirements document.

Triggers allow you to control the behavior of **cleartool** commands and Rational ClearCase operations by arranging for a specific program or executable script to run before or after the command executes. Virtually any operation that modifies an element can fire a trigger. Special environment variables make the relevant information available to the script or program that implements the procedure.

Preoperation triggers fire before the designated Rational ClearCase command is executed. A preoperation trigger on **checkin** can prompt the developer to add an appropriate comment. Postoperation triggers fire after a command has exited and can take advantage of the command's exit status. For example, a postoperation trigger on the **checkin** command can send an e-mail message to the QA department, indicating that a particular developer modified a particular element.

Triggers can also automate a variety of process management functions. For example:

- Applying attributes or attaching labels to objects when they are modified
- Logging information that is not included in the Rational ClearCase event records
- Initiating a build and/or source code analysis whenever particular objects are modified

For more information on these mechanisms, see Chapter 12, “Implementing project development policies,” on page 179.

A lock on an element or directory prevents all developers (except those included on an exception list) from modifying it. Locks are useful for implementing temporary restrictions. For example, during an integration period, a lock on a single object—the **main** branch type—prevents all users who are not on the integration team from making any changes.

The effect of a lock can be small or large. A lock can prevent any new development on a particular branch of a particular element; another lock can apply to the entire VOB, preventing developers from creating any new element of type **compressed_file** or using the version label **RLS_1.3**.

Locks can also be used to retire names, views, and VOBs that are no longer used. For this purpose, the locked objects can be tagged as *obsolete*, effectively making them invisible to most commands.

Global types

The Rational ClearCase global type facility makes it easy for you to ensure that the branch, label, attribute, hyperlink, and element types they need are present in all VOBs your project uses. The *IBM Rational ClearCase Administrator's Guide* has more information about creating and using global types.

Generating reports

An event record is created and stored each time an element is modified or merged. Many Rational ClearCase commands include selection and filtering options that you can use to create reports based on these records. The scope of such reports can cover a single element for a set of objects or for entire VOBs.

You can use event records and metadata to implement project policies. (For more detail, see Chapter 12, “Implementing project development policies,” on page 179.) Event records and other metadata can also be useful if you need to generate reports on activities managed by Rational ClearCase operations (for example, the complete history of changes to an element). a variety of report-generation tools are provided. For more information on this topic, see the **fmt_ccase** reference page in the *IBM Rational ClearCase Command Reference*.

Integrating changes

During the lifetime of a project, the contents of individual elements diverge as they are branched and usually converge in a merge operation. Typically, the project manager periodically merges most branches back to the **main** branch to ensure that the code base maintains a high degree of integrity and to have a single latest version of each element from which new versions can safely branch. Without regular merges, the code base quickly develops a number of dangling branches, each with slightly different contents. In such situations, a change made to one version must be propagated by hand to other versions, a tedious process that is prone to error.

As a project manager, you must establish merge policies for your project. Typical policies include the following:

- Developers merge their changes to the **main** branch. This can work well when the number of developers or the number of changed files is small and the developers are familiar with the mechanics of merging. Developers must also

understand the nature of other changes they may encounter when the merge target is not the immediate predecessor of the version being merged, which happens when several developers are working on the same file in parallel.

- Developers merge their changes to an integration branch. This provides a buffer between individual developers' merges and the **main** branch. The project manager or system integrator then merges the integration branch to the **main** branch.
- Developers must merge from the **main** branch to their development branch before merging to the **main** branch or integration branch. This type of merge promotes greater stability by forcing merge-related instability to the developers' private branches, where problems can be resolved before they affect the rest of the team.
- The project manager designates slots for developer merges to the **main** branch. This is a variation on several of the mechanisms already described. It provides an additional level of control in situations where parallel development is going on.

For more information about merging, see Chapter 14, "Integrating changes," on page 219.

Chapter 11. Defining project views

This chapter explains project views in a base ClearCase environment.

About defining project views

You need to know how config specs work and understand how config specs are useful for project development work, for nondevelopment tasks such as monitoring progress and doing research, and for running project builds. It also may be necessary to know how to share config specs among the Windows system, Linux, and the UNIX system.

How config specs work

When you create views for your project, you must prepare one or more *config specs* (configuration specifications). Config specs allow you to achieve the degree of control that you need to have over project work by controlling which versions developers see and what operations they can perform in specific views. You can narrow a view to a specific branch or open it to an entire VOB. You can also disallow checkouts of all selected versions or restrict checkouts to specific branches.

A config spec contains a series of rules that are used to select the versions that appear in the view. When team members use a view, they see the versions that match at least one of the rules in the config spec. The version tree of each element is searched for the first version that matches the first rule in the config spec. If no versions match the first rule, a version that matches the second rule is sought. If no versions of an element match any rule in the config spec, no versions of the element appear in the view.

The order in which rules appear in the config spec determine which version of a given element is selected. The various examples in this chapter examine this behavior in different contexts. For details about preparing config specs, see the **config_spec** reference page.

Default config spec

The following config spec defines a dynamic configuration:

```
(1)     element * CHECKEDOUT
(2)     element * /main/LATEST
```

The config spec selects changes made on the **main** branch of every element throughout the entire source tree, by any developer. This is the *default config spec*, to which each newly created view is initialized.

When you create a view with the **mkview** command or the View Creation Wizard (the Windows system only), the contents of file **default_config_spec** (located in *ccase–home–dir*) become the config spec of the new view. A view with this config spec provides a private work area that selects your checked-out versions (Rule 1). By default, when you check out a file, you check out from the latest version on the **main** branch (Rule 2). While an element is checked out to you, you can change it without affecting anyone else's work. When you check in the new version, the changes are available to developers whose views select **/main/LATEST** versions.

The view also selects all other elements (that is, all elements that you have not checked out) on a read-only basis. If another user checks in a new version on the **main** branch of such an element, the new **LATEST** version appears in this *dynamic view* immediately.

By default, *snapshot views* also include the two *version selection rules* shown above. In addition, snapshot view config specs include *load rules*, which specify which elements or subtrees to load into the snapshot view. For details on creating snapshot views, see *Developing Software* online help.

Product Note: Rational ClearCase LT supports only snapshot views.

The standard configuration rules

The two configuration rules in the default config spec appear in many of this chapter examples. The **CHECKEDOUT** rule allows you to modify existing elements. If you try to check out elements in a view that omits this rule, you can do so, but **cleartool** generates the following warning:

```
% cleartool checkout –nc cmd.c
cleartool: Warning: Unable to rename "cmd.c" to "cmd.c.keep":
Read-only filesystem.
cleartool: Error: Checked out version, but could not copy to "cmd.c":
File exists.
Correct the condition, then uncheckout and re-checkout the element.
cleartool: Warning: Copied checked out version to "cmd.c.checkedout".
cleartool: Warning: Checked-out version is not selected by view.
Checked out "cmd.c" from version "/main/7".
```

In this example, the config spec continues to select version 7 of element **cmd.c**, which is read-only. A read-write copy of this version, **cmd.c.checkedout**, is created in view-private storage. (This is not a recommended way of working.)

The `/main/LATEST` rule selects the most recent version on the **main** branch to appear in the view.

In addition, a `/main/LATEST` rule is required to create new elements in a view. If you create a new element when this rule is omitted, your view does not select that element. (Creating an element involves creating a **main** branch and an empty version, `/main/0`.)

Omitting the standard configuration rules

It makes sense to omit one or both of the standard configuration rules only if a view is not going to be used to modify data. For example, you can configure a **historical view**, to be used only for browsing old data. Similarly, you can configure a view in which to compile and test only or to verify that sources have been labeled properly.

Config spec include files

An include file facility makes it easy to ensure that all team members are using the same config spec. For example, the configuration rules in this config spec can be placed in file `/public/c_specs/major.csp`. Each developer then needs the following one-line config spec:

```
(1)             include /public/c_specs/major.csp
```

Note: If you are sharing config specs among Linux, the UNIX system, and Windows computers where the VOB tags are different, you must have two

sources, or you must store the config spec in a directory on Linux and the UNIX system that is also accessible from the Windows platform.

If you want to modify this config spec (for example, to adopt the no-directory-branching policy), only the contents of **major.csp** need to change.

To reconfigure your view with the modified config spec

Use this command:

```
cleartool setcs –current
```

This command causes the view server to flush its caches and reevaluate the current config spec.

Project environment for sample config specs

You can use different config specs for different kinds of development and management tasks. The three sections that follow present sample config specs useful for various aspects of project development, project management and research, and project builds. This section presents the development environment that these config specs are based on.

Developers use a VOB whose VOB tag is `/vobs_monet`, which has this structure:

```
/vobs/monet                                 (VOB tag, VOB mount point)
                    src/                    (C language source files)
                    include/                (C language header files)
                    lib/                    (project libraries)
```

For the purposes of this chapter, suppose that the `lib` directory has this substructure:

```
lib/
                    libcalc.a               (checked-in staged version of
                                            library)
                    libcmd.a                (checked-in staged version of
                                            library)
                    libparse.a              (checked-in staged version of
                                            library)
                    libpub.a                (checked-in staged version of
                                            library)
                    libaux1.a               (checked-in staged version of
                                            library)
                    libaux2.a               (checked-in staged version of
                                            library)
                    libcalc/                (sources for calc library)
                    libcmd/                 (sources for cmd library)
                    libparse/               (sources for parse library)
                    libpub/                 (sources for pub library)
                    libaux1/                (sources for aux1 library)
                    libaux2/                (sources for aux2 library)
```

Sources for libraries are located in subdirectories of **lib.** After a library is built in its source directory, it can be staged to `/vobs_monet/lib`.

On Linux and the UNIX system, the build scripts for the project executable programs can instruct the link editor, **ld(1)**, to use the libraries in this directory (the library staging area) instead of a more standard location (for example, `/usr/local/lib`).

On the Windows system, you can use the libraries in this directory (the library staging area) instead of a more standard location by setting the LIB environment variable or by changing the makefile.

The following labels are assigned to versions of **vobs_monet** elements.

Version Labels	Description
R1.0	First customer release
R2_BL1	Baseline 1 prior to second customer release
R2_BL2	Baseline 2 prior to second customer release
R2.0	Second customer release

These version labels have been assigned to versions on the **main** branch of each element. Most project development work takes place on the **main** branch. For some special tasks, development takes places on a subbranch.

Subbranches	Description
major	Used for work on the application's graphical user interface, certain computational algorithms, and other major enhancements
r1_fix	Used for fixing bugs in Release 1.0

Windows Note: Config specs allow absolute VOB paths—absolute paths that begin with a VOB tag but do not include a drive or view tag prefix. This form of path is required to specify VOB elements without regard for current drive assignments or active views. For example:

\vob_gopher\lib*
(absolute VOB path, where \vob_gopher is the VOB tag)

\monet\src*
(absolute VOB path, where \monet is the VOB tag)

Z:\monet\src*
(drive-specific path; not recommended)

M:\myview\vob_gopher\lib*
(view-extended path; not recommended)

Views for project development

The config specs in this section are useful for project development because they enforce various branching policies.

View for new development on a branch

You can use this config spec for work to be isolated on branches named **major**:

```
(1)     element * CHECKEDOUT
(2)     element * .../major/LATEST
(3)     element * BASELINE_X –mkbranch major
(4)     element * /main/LATEST –mkbranch major
```

In this scheme, all checkouts occur on branches named **major** (Rule 2).

The **major** branches are created at versions that constitute a consistent baseline: a major release, a minor release, or a set of versions that produces a working version of the application. In this config spec, the baseline is defined by the version label **BASELINE_X**.

Variation that uses a time rule

Other developers can check in versions that become visible in your view, but are incompatible with your own work. In such cases, you can continue to work on sources as they existed before those changes were made. For example, Rule 2 in this config spec selects the latest version on the main branch as of 4:00 P.M. on November 12:

```
(1)     element * CHECKEDOUT
(2)     element * /major/LATEST –time 12-Nov.16:00
(3)     element * BASELINE_X –mkbranch major
(4)     element * /main/LATEST –mkbranch major
```

Note that this rule has no effect on your own checkouts.

View to modify an old configuration

This config spec allows developers to modify a configuration defined with version labels:

```
(1)     element * CHECKEDOUT
(2)     element * .../r1_fix/LATEST
(3)     element * R1.0 –mkbranch r1_fix
```

Note the following points about the configuration:

- Elements can be checked out (Rule 1).
- The **checkout** command creates a branch named **r1_fix** at the initially selected version (the *auto-make-branch* clause in Rule 3).

A key aspect of this scheme is that the same branch name, **r1_fix**, is used in every modified element. The only administrative overhead is the creation of a single branch type, **r1_fix**, with the **mkbrtype** command.

This config spec is efficient. Two rules (Rules 2 and 3) configure the appropriate versions of all elements:

- For elements that have been modified, this version is the most recent on the **r1_fix** subbranch (Rule 2).
- For elements that have not been modified, this version is the one labeled **R1.0** (Rule 3).

Figure 48 illustrates these elements. The **r1_fix** branch is a subbranch of the **main** branch. But Rule 2 handles the more general case, too. The **...** wildcard allows the **r1_fix** branch to occur anywhere in the version tree of any element, and at different locations in the version trees of different elements.

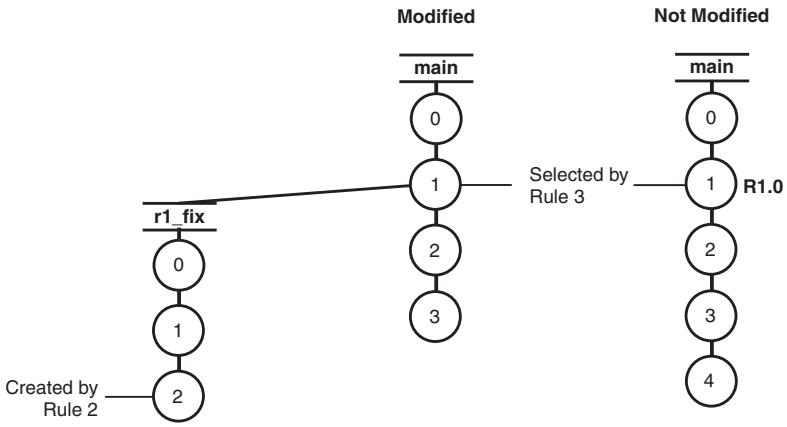


Figure 48. Making a change to an old version

Omitting the /main/LATEST rule

The config spec in "View to modify an old configuration" on page 167 omits the standard `/main/LATEST` rule. This rule is not useful for work with VOBs in which the version label **R1.0** does not exist. In addition, it is not useful in situations where new elements are created. If your development policy is to not create new elements during maintenance of an old configuration, the absence of a `/main/LATEST` rule is appropriate.

To allow creation of new elements during the modification process, add a fourth configuration rule:

```
(1)     element * CHECKEDOUT
(2)     element * /main/r1_fix/LATEST
(3)     element * R1.0 –mkbranch r1_fix
(4)     element * /main/LATEST –mkbranch r1_fix
```

When a new element is created with **mkelem**, the **–mkbranch** clause in Rule 4 causes the new element to be checked out on the **r1_fix** branch (which is created automatically). This rule conforms to the scheme of localizing all changes to **r1_fix** branches.

Variation that uses a time rule

This baseline configuration is defined with a time rule.

```
(1)     element * CHECKEDOUT
(2)     element * /main/r1_fix/LATEST
(3)     element * /main/LATEST –time 4-Sep:02:00 –mkbranch r1_fix
```

View to implement multiple-level branching

This config spec implements and enforces consistent multiple-level branching.

```
(1)     element * CHECKEDOUT
(2)     element * .../major/autumn/LATEST
(3)     element * .../major/LATEST –mkbranch autumn
(4)     element * BASELINE_X –mkbranch major
(5)     element * /main/LATEST –mkbranch major
```

A view configured with this config spec is appropriate in the following situation:

- All changes from the baseline designated by the **BASELINE_X** version label must be made on a branch named **major**.

- Moreover, you are working on a special project, whose changes are to be made on a subbranch of **major**, named **autumn**.

Figure 49 shows what happens in such a view when you check out an element that has not been modified since the baseline.

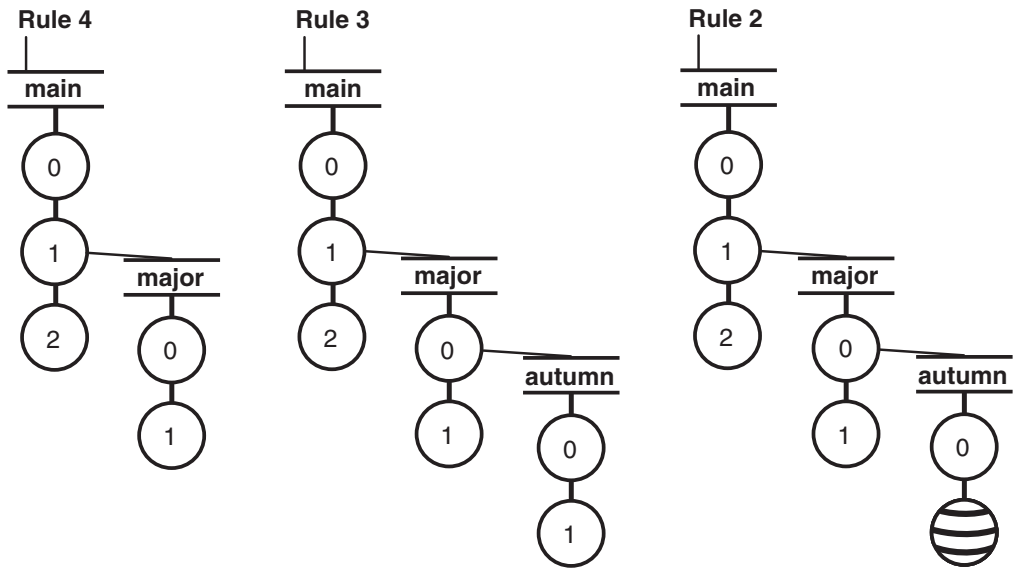


Figure 49. Multiple-level auto-make-branch

1. After an element is checked out, the **mkbranch** clause in Rule 4 creates the **major** branch at the **BASELINE_X** version.
2. The **mkbranch** clause in Rule 3 creates the **autumn** branch at `\main\major\0`.
3. When the **checkout** operation finishes, Rule 2 applies; the most recent version, `\main\major\autumn\0`, is checked out.

For more information about multiple-level branching, see the **config_spec** and **checkout** reference pages.

View to restrict changes to a single directory

This config spec is appropriate for a developer who can make changes in one directory only, `/vobs/monet/src`:

```
(1)     element * CHECKEDOUT
(2)     element src/* /main/LATEST
(3)     element * /main/LATEST –nocheckout
```

The most recent version of each element is selected (Rules 2 and 3), but Rule 3 prevents checkouts to all elements except those in the directory specified.

Note that Rule 2 matches elements in any directory named **src**, in any VOB. The pattern `/vobs/monet/src/*` restricts matching to only one VOB.

You can easily extend this config spec with additional rules that allow additional areas of the source tree to be modified.

Views to monitor project status

The config specs presented in "View that uses attributes to select versions" through "Historical view defined by a version label" on page 173 are useful for views used for research and monitoring project status.

View that uses attributes to select versions

Suppose that the QA team also works on the **major** branch. Individual developers are responsible for making sure that their modules pass a QA check. The QA team builds and tests the application, using the most recent versions that have passed the check.

The QA team can work in a view that uses this config spec:

```
(1)     element –file src/* /main/major/{QAOK=="Yes"}
(2)     element * /main/LATEST
```

To make this scheme work, you must create an attribute type, **QAOK**. Whenever a new version that passes the QA check is checked in on the **major** branch, an instance of **QAOK** with the value **Yes** is attached to that version. (This can be done manually or with a Rational ClearCase trigger.)

If an element in the /src directory has been edited on the **major** branch, this view selects the branch's most recent version that has been marked as passing the QA check (Rule 1). If no version has been so marked or if no **major** branch has been created, the most recent version on the **main** branch is used (Rule 2).

Tip: Rule 1 on this config spec does not provide a match if an element has a **major** branch, but no version on the branch has a **QAOK** attribute. This command can locate the branches that do not have this attribute:

On Linux and the UNIX system:

```
cleartool find . –branch '{brtype(major) && \! attype_sub(QAOK)}' –print
```

The backslash (\ ) is required in the C shell only, to keep the exclamation point (!) from indicating a history substitution.

On the Windows system:

```
cleartool find . –branch "{brtype(major) && ! attype_sub(QAOK)}" –print
```

The **attype_sub** primitive searches for attributes on versions and branches of an element and on the element itself.

This scheme allows the QA team to monitor the progress of the rest of the group (see Figure 50).

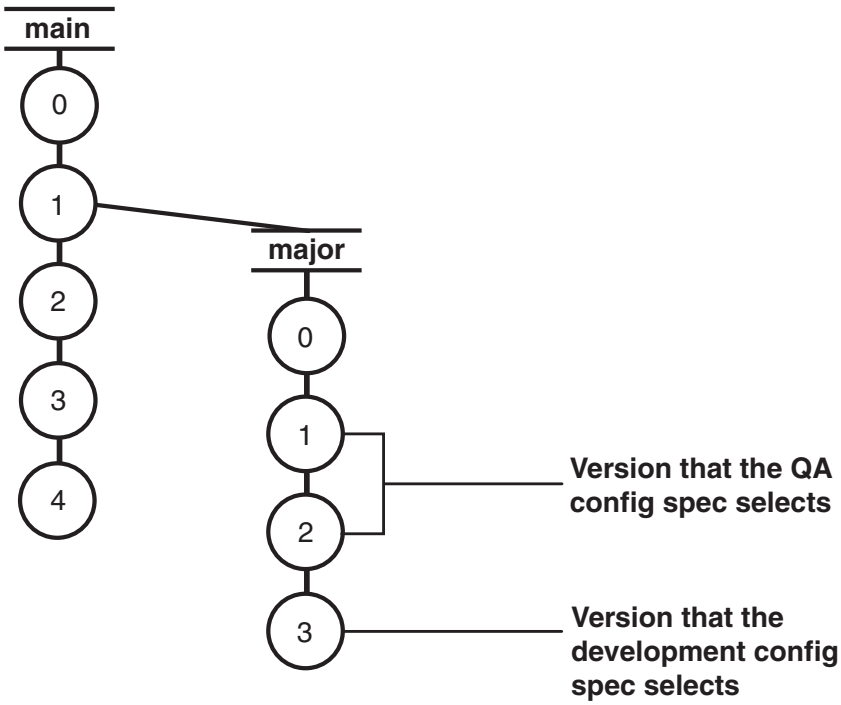


Figure 50. Development config spec versus QA config spec

The development config spec always selects the most recent version on the **major** branch, but the QA config spec may select an intermediate version.

Pitfalls for development of using attributes to select versions

You may be tempted to add a **CHECKEDOUT** rule to the config spec described in "View that uses attributes to select versions" on page 170. This addition turns the QA configuration into the following development configuration:

```
(0)     element * CHECKEDOUT
(1)     element -file src/* /main/major/{QAOK=="Yes"}
(2)     element * /main/LATEST
```

It may seem desirable to use attributes, or other kinds of metadata, in addition to (or instead of) branches to control version selection in a development view. But such schemes introduce complications. Suppose that the config spec above selects version /main/major/2 of element .../src/cmd.c (see Figure 51).

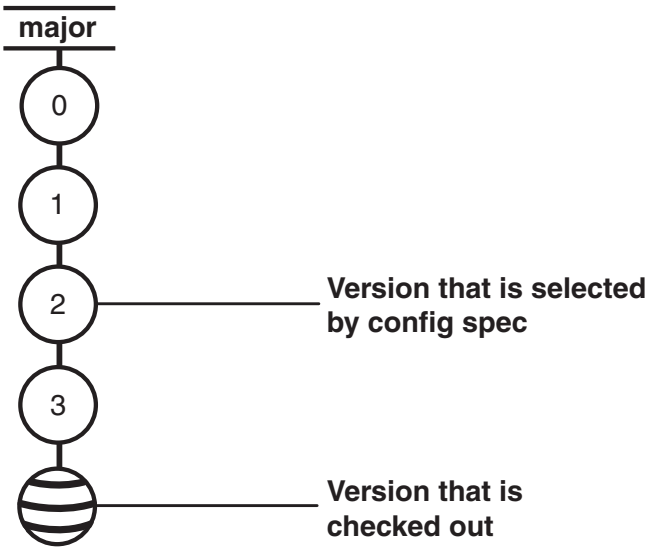


Figure 51. Checking out a branch of an element

Performing a checkout in this view checks out version /main/major/3, not version /main/major/2, and generates the following message:

```
cleartool: Warning: Version checked out is different from version
previously selected by view.
Checked out "cmd.c" from version "/main/major/3".
```

This behavior reflects the Rational ClearCase restriction that new versions can be created only at the end of a branch. Although such operations are possible, they are potentially confusing to other team members. And, in this situation, it is almost certainly not what the developer who checks out the element wants to happen.

You can avoid the problem by making the following indicated changes in the config spec:

```
(0)     element * CHECKEDOUT
(0a)    element * /main/major/temp/LATEST
(1)     element -file src/* /main/major/{QAOK=="Yes"} -mkbranch temp
(2)     element * /main/LATEST
```

The modified config spec creates another branching level at the version that the attribute selects.

View that shows changes of one developer

The following config spec makes it easy to examine all changes that a developer has made since a certain milestone:

```
(1)     element * '/main/{created_by(jackson) && created_since(25-Apr)}'
(2)     element * /main/LATEST -time 25-Apr
```

Tip: Rule 1 must be contained on a single physical text line.

A particular date, April 25, is used as the milestone. The configuration is a snapshot of the main line of development at that date (Rule 2), overlaid with all changes that user **jackson** has made on the **main** branch since then (Rule 1).

The output of the **cleartool ls** command distinguishes **jackson**'s files from the others: each entry includes an annotation as to which configuration rule applies to the selected version.

This is a research view, not a development view. The selected set of files may not be consistent: some of **jackson**'s changes may rely on changes made by others, and those other changes are excluded from this view. Thus, this config spec omits the standard **CHECKEDOUT** and `/main/LATEST` rules.

Historical view defined by a version label

The following config spec defines a historical configuration:

```
(1)             element * R1.0 –nocheckout
```

This view always selects the set of versions labeled **R1.0**. In this scenario, all these versions are on the **main** branch of their elements. If the **R1.0** label type is *one-per-element*, not *one-per-branch*, this config spec selects the **R1.0** version on a subbranch. (For more information, see the **mklbtype** reference page.)

The **–nocheckout** qualifier prevents any element from being checked out in this view. (It also prevents creation of new elements, because the parent directory element must be checked out.) Thus, there is no need for the **CHECKEDOUT** configuration rule.

Tip: The set of versions selected by this view can change, because version labels can be moved and deleted. For example, using the command **mklabel –replace** to move **R1.0** from version 5 of an element to version 7 changes which version appears in the view. Similarly, using **rmlabel** suppresses the specified elements from the view. (The **cleartool ls** command lists them with a `[no version selected]` annotation.) If the label type is locked with the **lock** command, the configuration cannot change.

You can use this configuration to rebuild Release 1.0, verifying that all source elements have been labeled properly. You can also use it to browse the old release.

Historical view defined by a time rule

The following config spec defines a frozen configuration in a slightly different way than the config spec that is described in "Historical view defined by a version label":

```
(1)             element * /main/LATEST –time 4-Sep.02:00 –nocheckout
```

This configuration selects the version that was the most recent on the **main** branch on September 4 at 2 A.M. Subsequent checkouts and checkins cannot change which versions satisfy this criterion; only deletion commands such as **rmver** or **rmelem** can change the configuration. The **–nocheckout** qualifier prevents anyone from checking out or creating elements.

This configuration can be used to view a set of versions that existed at a particular point in time. If modifications must be made to this source base, you must modify the config spec to "unfreeze" the configuration.

Views for project builds

Certain config specs (described in "View that uses results of a nightly build" through "View that selects versions that built a particular program" on page 175) are useful for running the various types of builds that are required for a project.

View that uses results of a nightly build

Many projects use scripts to run unattended software builds every night. The success or failure of these builds determine the impact of any checked-in changes on the application. In layered build environments, the builds can also provide up-to-date versions of lower-level software (for example, libraries and utility programs).

Suppose that every night, a script does the following operations:

- Builds libraries in various subdirectories of /vobs/monet/lib
- Checks them in as DO versions in the library staging area, /vobs/monet/lib
- Labels the versions **LAST_NIGHT**

You can use the following config spec if you want to use the libraries produced by the nightly builds:

```
(1)     element * CHECKEDOUT
(2)     element lib/*.a LAST_NIGHT
(3)     element lib/*.a R2_BL2
(4)     element * /main/LATEST
```

The **LAST_NIGHT** version of a library is selected whenever such a version exists (Rule 2). If a nightly build fails, the previous build still has the **LAST_NIGHT** label and is selected. If no **LAST_NIGHT** version exists (the library is not currently under development), the stable version labeled **R2_BL2** is used instead (Rule 3).

For each library, selecting versions with the **LAST_NIGHT** label rather than the most recent version in the staging area allows developers to stage new versions the next day, without affecting developers who use this config spec.

Variations that select versions of project libraries

The scheme that is described in "View that uses results of a nightly build" uses version labels to select particular versions of libraries. For more flexibility, other versions can be selected as shown by the following added rules:

```
(1)     element * CHECKEDOUT
(2a)    element lib/libcmd.a LAST_NIGHT
(2b)    element lib/libparse.a LAST_NIGHT
(3a)    element lib/libcalc.a R2_BL2
(3b)    element lib/*.a /main/LATEST
(4)     element * /main/LATEST
```

The **LAST_NIGHT** version of some libraries can be selected, the **R2_BL2** version of others, and the most recent version of still others. (Rule 3b is not required here, because Rule 4 handles all other libraries. It is included for clarity only.)

Other kinds of metadata can also be used to select library versions. A config spec can mix and match library versions as the following added rules indicate:

```
(1)     element * CHECKEDOUT
```

```
(2)     element lib/libcmd.a {lib_selector=="experimental"}
(3)     element lib/libcalc.a {lib_selector=="experimental"}
(4)     element lib/libparse.a {lib_selector=="stable"}
(5)     element lib/*.a {lib_selector=="released"}
(6)     element * /main/LATEST
```

For example, **lib_selector** attributes can take values such as **experimental**, **stable**, and **released**.

View that selects versions of application subsystems

The following config spec selects specific versions of the application subsystems:

```
(1)     element * CHECKEDOUT
(2)     element /vobs/monet/lib/... R2_BL1
(3)     element /vobs/monet/include/... R2_BL2
(4)     element /vobs/monet/src/... /main/LATEST
(5)     element * /main/LATEST
```

In this situation, a developer is making changes to the application source files on the **main** branch (Rule 4). Builds of the application use the libraries in directory /lib that were used to build Baseline 1, and the header files in directory /include that were used to build Baseline 2.

View that selects versions that built a particular program

The following config spec defines a view that selects only enough files that are required to rebuild a particular program or examine its sources:

```
(1)             element * –config /vobs/monet/src/monet
```

All elements that were not involved in the build of **monet** appear in the output of Rational ClearCase **ls** with a `[no version selected]` annotation.

This config spec selects the versions listed in the config record (CR) of a particular derived object (and in the config records of all its build dependencies). It can be a derived object that was built in the current view, or another view, or it can be a *DO version*.

In this config spec, **monet** is a derived object in the current view. You can reference a derived object in another view with an extended path that includes a *DO-ID* in the following format:

```
(1)             element * –config /vobs/monet/src/monet@@09-Feb.13:56.812
```

But typically, this kind of config spec is used to configure a view from a derived object that has been checked in as a DO version.

Configuring the makefile

By default, a derived object config record does not list the version of the makefile that was used to build it. Instead, the config record includes a copy of the build script itself. (Why? When a new version of the makefile is created with a revision to one target build script, the configuration records of all other derived objects built with that makefile are not rendered out of date.)

But if the **monet** program is to be rebuilt in this view using **clearmake** (or standard **make** on Linux and the UNIX system or **omake** on the Windows system), a version of the makefile must be selected somehow. You can have **clearmake**

record the makefile version in the config record by including the special **clearmake** macro invocation **$(MAKEFILE)** in the target dependency list in the following formats:

On the Windows system:

```
monet.exe: $(MAKEFILE) monet.obj ...
   link –out:monet.exe monet.obj ...
```

On Linux and the UNIX system:

```
monet: $(MAKEFILE) monet.o ...
   cc –o monet ...
```

The **clearmake** command always records the versions of explicit dependencies in the config record.

Alternatively, you can configure the makefile at the source level: attach a version label to the makefile at build time, and then use a config spec like the one in Historical view defined by a version label on page 173 or View to modify an old configuration on page 167 to configure a view for building. You can also use the special target **.DEPENDENCY_IGNORED_FOR_REUSE**. For more information, see *IBM Rational ClearCase Guide to Building Software*.

Fixing bugs in the program

If a bug is discovered in the **monet** program, as rebuilt in a view that selects only enough files required to rebuild a particular program, it is easy to convert the view from a build configuration to a development configuration. As usual, when making changes in old sources, follow this strategy:

- Create a branch at each version to be modified
- Use the same branch name (that is, create an instance of the same branch type) in every element

If the fix branch type is **r1_fix,** this modified config spec reconfigures the view for performing the fix in the following rules:

```
(1)     element * CHECKEDOUT
(2)     element * .../r1_fix/LATEST
(3)     element * –config /vobs/monet/src/monet –mkbranch r1_fix
(4)     element * /main/LATEST –mkbranch r1_fix
```

Selecting versions that built a set of programs

You can expand the config spec that selects only enough files required to rebuild a particular program (see "View that selects versions that built a particular program" on page 175). You can configure a view with the sources used to build a set of programs, rather than to build a single program. Use a config spec similar to the following one:

```
(1)     element * –config /proj/monet/src/monet
(2)     element * –config /proj/monet/src/xmonet
(3)     element * –config /proj/monet/src/monet_conf
```

However, there can be version conflicts in such configurations. For example, different versions of file params.h may have been used in the builds of **monet** and **xmonet**. In this situation, the version used in **monet** is configured, because its configuration rule came first. Similarly, there can be conflicts when using a single **–config** rule. If the specified derived object was created by actually building some targets and using DO versions of other targets, multiple versions of some source files may be involved.

As described in “Fixing bugs in the program” on page 176, you can modify this config spec to change the build configuration to a development configuration.

Sharing config specs among Linux, the UNIX system, and Windows system

In principle, you can share config specs among Linux, the UNIX system, and the Windows system. That is, users on all types of systems, using views whose storage directories reside on another type of platform, can set and edit the same set of config specs.

You should avoid sharing config specs across platforms. If possible, maintain separate config specs for each platform. However, if you must share config specs, adhere to the following requirements:

- Use slashes (/) in paths instead of backslashes (\).
- Use relative paths instead of full paths, whenever possible. And do not use VOB tags in paths. You can ignore this requirement if your VOB tags on Linux, the UNIX system, and Windows systems all use single, identical path components that differ only in their leading slash characters, for example \src and /src.
- Always edit and set config specs on Linux or the UNIX system.

For more information, see “Path separators” on page 177, “Paths in config spec element rules” on page 177, and “Config spec compilation” on page 178.

Path separators

When writing config specs to be shared by the Windows, Linux, and the UNIX system, you must use slash (/) as the path separator instead of backslash (\). In Rational ClearCase configurations on Linux or the UNIX system, only slashes are recognized.

Tip: The **cleartool** command recognizes both slashes and backslashes in paths; **clearmake** is less flexible. See *IBM Rational ClearCase Guide to Building Software* for more information.)

Paths in config spec element rules

Network regions on the Windows system, Linux, and the UNIX system often use different VOB tags to register the same VOBs. Only single-component VOB tags, such as **\proj1**, are permitted on the Windows computer; multiple-component VOB tags, such as **/vobs/src**, are common on Linux and the UNIX system.

When VOB tags differ between regions, any config spec element rules that use full paths (which include VOB tags) can be resolved when the config spec is compiled (**cleartool edcs** and **setcs** commands) but only by computers in the applicable network region. This implies a general restriction regarding shared config specs: a given config spec must be compiled only on the operating system for which full paths in element rules make sense. That is, a config spec with full paths is shareable across network regions, even when VOB tags disagree, but it must be compiled in the right place.

The restrictions do not apply if either of the following is true (see “Config spec compilation” on page 178):

- The config spec element rules use only relative paths, which do not include VOB tags.

- Shared VOBs are registered with identical, single-component VOB tags in the network regions of the Windows system and Linux and the UNIX system. (The VOB tags **\r3vob** and **/r3vob** are treated as if they were identical because they differ only in the leading slashes.)

Config spec compilation

A config spec that is in use exists in both text file and compiled formats. A config spec compiled form is portable. The restriction is that full VOB paths in element rules must be resolvable at compile time. A config spec is compiled when you edit or set it (with the **cleartool edcs** or **cleartool setcs** command or a Rational ClearCase graphic user interface (GUI)). If a user on the other operating system recompiles a config spec (by issuing the **edcs** or **setcs** command or causing the GUI to execute one of these commands), the config spec becomes unusable by *any* computer using that view. If this happens, recompile the config spec on the original operating system.

The following config spec element rule may cause problems:

```
element \vob_p2\abc_proj_src\*      \main\rel2\LATEST
```

If the VOB is registered with VOB tag **\vob_p2** on a Windows network region, but with VOB tag **/vobs/vob_p2** in the network region on Linux and the UNIX system, only Windows computers can compile the config spec.

To address the problem, do one of the following:

- Use relative paths that do not include VOB tags, for example:

  ```
  element ...\abc_proj_src\*      \main\rel2\LATEST
  ```

- On Linux and the UNIX system, change the VOB tag so that it has a single component, for example, **/vob_p2**.

Chapter 12. Implementing project development policies

This chapter presents scenarios that implement policies in a base ClearCase project.

About implementing project development policies

You need to know how to implement and enforce common development policies with Rational ClearCase configurations. You can use various combinations of these functions and metadata:

- Attributes
- Labels
- Branches
- Triggers
- Config specs
- Locks
- Hyperlinks

For information about the way to define triggers for use on Linux, the UNIX system, and the Windows computer, see "Sharing triggers among different types of platform" on page 188.

Good documentation of changes is required

Each Rational ClearCase command that modifies a VOB creates one or more *event records*. Many such commands (for example, **checkin**) prompt for a comment. The event record includes the user name, date, comment, host, and description of what was changed.

To prevent developers from subverting the system by providing empty comments, you can create a preoperation trigger to monitor the **checkin** command.

Product Note: When a trigger is fired on a Windows system, a Rational ClearCase function proceeds based on the success or failure of the trigger operation, as determined by the trigger script exit code. A .bat file returns the exit code of its last command. Preoperation triggers are the only kind of trigger that causes the Rational ClearCase operation to fail.

Trigger definition on Linux and the UNIX system:

```
cleartool mktrtype –element –all –preop checkin \
–c "must enter descriptive comment" \
–exec /public/scripts/comment_policy.sh CommentPolicy
```

Trigger definition on the Windows system:

```
cleartool mktrtype –element –all –preop checkin ^
–c "must enter descriptive comment" ^
–exec \\neon\scripts\comm_pol.bat CommentPolicy
```

Trigger action script on Linux and the UNIX system:

```
#!/bin/sh
#
#   comment_policy
#
```

```
ACCEPT=0
REJECT=1
WORDCOUNT='echo $CLEARCASE_COMMENT | wc -w'

if [ $WORDCOUNT -ge 10 ] ; then
   exit $ACCEPT
else
   exit $REJECT
fi
```

Trigger action script on the Windows system:

```
@echo off
rem comm_pol.bat
rem
rem Check for null comment
rem
if "%CLEARCASE_COMMENT%"=="" copy > NUL:
```

The trigger action script analyzes the user's comment (passed in an environment variable), and disallows unacceptable ones.

All source files require a progress indicator

You can monitor the progress of individual files or determine which or how many files are in a particular state. You can use attributes to preserve this information and triggers to collect it.

In this case, you can create a string-valued attribute type, **Status**, which accepts a specified set of values.

Attribute definition on Linux and the UNIX system:

```
cleartool mkattype -c "standard file levels" \
-enum ' "inactive","under_devt","QA_approved" ' Status
Created attribute type "Status".
```

Attribute Definition on the Windows system:

```
cleartool mkattype -c "standard file levels" ^
-enum "\"inactive\",\"under_devt\",\"QA_approved\"" Status
Created attribute type "Status".
```

Developers apply the **Status** attribute to many different versions of an element. Its value in early versions on a branch is likely to be **inactive** and **under_devt**; on later versions, its value is **QA_approved**. The same value can be used for several versions, or moved from an earlier version to a later version.

To enforce consistent application of this attribute to versions of all source files, you can create a **CheckStatus** trigger whose action script prevents developers from checking in versions that do not have a **Status** attribute.

Trigger definition on Linux and the UNIX system:

```
cleartool mktrtype -element -all -preop checkin \
-c "all versions must have Status attribute" \
-exec 'Perl /public/scripts/check_status.pl' CheckStatus
```

Trigger definition on the Windows system:

```
cleartool mktrtype -element -all -preop checkin ^
-c "all versions must have Status attribute" ^
-exec "ccperl \\neon\scripts\check_status.pl" CheckStatus
```

Trigger action script:

```
$pname = $ENV{'CLEARCASE_PN'};
$val = "";
$val = 'cleartool describe -short -aattr Status $pname';

if ($val eq "") {
exit (1);
} else {
   exit (0);
}
```

Label all versions used in key configurations

To identify which versions of which elements contributed to a particular baseline or release, you can attach labels to these versions. For example, after Release 2 is built and tested, you can create label type **REL2,** using the **mklbtype** command. You can then attach **REL2** as a version label to the appropriate source versions, using the **mklabel** command.

Which are the appropriate versions? If Release 2 was built from the bottom up in a particular view, you can use the following commands to label the versions selected by that view:

```
cleartool mklbtype –c "Release 2.0 sources" REL2

cleartool mklabel –recurse REL2 top-level-directory
```

Alternatively, you can use the configuration records of the release derived objects to control the labeling process:

```
clearmake vega

... sometime later, after QA approves the build:

cleartool mklabel –config vega@@17-Jun.18:05 REL2
```

Using configuration records to attach version labels ensures accurate and complete labeling, even if developers have created new versions since the release build. Development work can continue while quality assurance and release procedures are performed.

To prevent version label **REL2** from being used again, you must lock the label type:

```
cleartool lock –nusers vobadm lbtype:REL2
```

The object is locked to all users except those specified with the **–nusers** option, in this case, **vobadm**.

Isolate work on release bugs to a branch

You can fix bugs found in the released system on a named bug-fix branch, and begin this work with the exact configuration of versions from that release.

This policy reflects the Rational ClearCase baseline-plus-changes model. First, a label (for example, **REL2**) must be attached to the release configuration. Then, you or any team member can create a view with the following config spec to implement the policy:

```
element * CHECKEDOUT
element * .../rel2_bugfix/LATEST
element * REL2 -mbranch rel2_bugfix
```

If all fixes are made in one or more views with this configuration, the changes are isolated on branches of type **rel2_bugfix**. The **–mkbranch** option causes such branches to be created, as needed, when elements are checked out.

This config spec selects versions from **rel2_bugfix** branches, where branches of this type exist; it creates such a branch whenever a **REL2** version is checked out.

Avoid disrupting the work of other developers

To work productively, developers need to control when they see changes and which changes they see. The appropriate mechanism for this purpose is a view. Developers can modify an existing config spec or create a new one to specify exactly which changes to see and which to exclude.

To implement this policy, you can also require developers to write and distribute the config spec rule that filters out their checked-in changes. The following are sample config specs:

- To select your own work, plus all the versions that went into the building of Release 2:

```
element * CHECKEDOUT
element * REL2
```

- To select your own work, plus the latest versions as of Sunday evening:

```
element * CHECKEDOUT
element * /main/LATEST -time Sunday.18:00
```

- To select your own work, new versions created in the **graphics** directory, and the versions that went into a previous build:

```
element * CHECKEDOUT
element graphics/* /main/LATEST
element * -config myprog@@12-Jul.00:30
```

- To select your own work, the versions either you (**jones**) or Mary has checked in today, and the most recent quality-assurance versions:

```
element * CHECKEDOUT
element * '/main/{ created_since(06:00) && ( created_by(jones) ||
created_by(mary) ) }'
element * /main/{QAed=="TRUE"}
```

- To use the config spec include facility to set up standard sets of configuration rules for developers to add to their own config specs:

```
element * CHECKEDOUT
element msg.c /main/18
include /usr/cspecs/rules_for_rel2_maintenance
```

Deny access to project data when necessary

Occasionally, you may need to deny access to all or most project team members. For example, you may want to prevent changes to public header files until further notice. The **lock** command is designed to enforce such temporary policies:

- Lock all header files in a certain directory:

```
cleartool lock src/pub/*.h
```

- Lock the header files for all users except Mary and Fred:

```
cleartool lock –nusers mary,fred src/pub/*.h
```

- Lock all header files in the VOB:

```
cleartool lock eltype:c_header
```

- Lock an entire VOB:

```
cleartool lock vob:/vobs/myproj
```

Notify team members of relevant changes

To help team members keep track of changes that affect their own work, you can use postoperation triggers to send notifications of various events. For example, when developers change the graphic user interface (GUI), an e-mail message to the documentation group ensures that these changes are documented.

To enforce this policy, create a trigger type that sends mail, and then attach the trigger to the relevant elements (see "To attach triggers to existing elements" on page 184).

Trigger definition on Linux and the UNIX system:

```
cleartool mktrtype –nc -element –postop checkin \
   –exec /public/scripts/informwriters.sh InformWriters
Created trigger type "InformWriters".
```

Trigger action script on Linux and the UNIX system:

```
#!/bin/sh
#
#Init
tmp=/tmp/checkin_mail

# construct mail message describing checkin

cat > $tmp <<EOF
Subject: Checkin $CLEARCASE_PNAME by $CLEARCASE_USER
$CLEARCASE_XPNAME
Checked in by $CLEARCASE_USER.

Comments:
$CLEARCASE_COMMENT
EOF

# send the message

mail docgrp <$tmp

# clean up

#rm -f $tmp
```

Trigger definition on the Windows system:

```
cleartool mktrtype –nc -element –postop checkin ^
–exec "ccperl \\neon\scripts\informwriters.pl" InformWriters
Created trigger type "InformWriters".
```

Trigger action script on the Windows system:

```
use Net::SMTP;

my $smtp = new Net::SMTP 'neon.purpledoc.com';

$smtp->mail('Rational ClearCase Admin');
$smtp->to('Rational ClearCase Admin');
$smtp->to('docgrp');

$smtp->data();
$smtp->datasend("From: Rational ClearCase Admin\n");
$smtp->datasend("To: docgrp\n");
$smtp->datasend("Subject: checkin\n");
$smtp->datasend("\n");

# create variables for path/user/comment
```

```
$ver = $ENV{'CLEARCASE_XPN'};
$user = $ENV{'CLEARCASE_USER'};
$comment = $ENV{'CLEARCASE_COMMENT'};

$var = "Version: $ver\nUser: $user\nComment: $comment\n";

$smtp->datasend($var);
$smtp->dataend();
$smtp->quit;
```

To attach triggers to existing elements

1. Place the trigger on the *inheritance list* of all existing directory elements within the GUI source tree:

   ```
   cleartool find /vobs/gui_src –type d \
   –exec 'cleartool mktrigger –nattach InformWriters $CLEARCASE_PN'
   ```

2. Place the trigger on the *attached list* of all existing file elements within the GUI source tree:

   ```
   cleartool find /vobs/gui_src –type f  \
   –exec 'cleartool mktrigger InformWriters $CLEARCASE_PN'
   ```

All source files must meet project standards

To ensure that developers are following coding guidelines or other standards, you can evaluate their source files. You can create preoperation triggers to run user-defined programs, and cancel the commands that trigger them. For example, you can disallow checkin of C-language files that do not satisfy quality metrics. Suppose that you have defined an element type, **c_source**, for C language files (*.c).

Trigger definition on Linux and the UNIX system:

```
cleartool mktrtype –element –all –eltype c_source \
–preop checkin –exec '/public/scripts/apply_metrics.sh $CLEARCASE_PN'
ApplyMetrics
```

Trigger definition on the Windows system:

```
cleartool mktrtype –element –all –eltype c_source ^
–preop checkin –exec "\\neon\scripts\appl_met.bat %CLEARCASE_PN%"
ApplyMetrics
```

This trigger type **ApplyMetrics** applies to all elements; it fires when any element of type **c_source** is checked in. (When a new **c_source** element is created, the element is monitored.) If a developer attempts to check in a **c_source** file that fails the **apply_metrics.sh** or **appl_met.bat** test, the checkin fails.

Tip: The **apply_metrics.sh** script and the **appl_met.bat** file can read the value of CLEARCASE_PN from its environment. Having it accept a file name argument provides flexibility because the script or batch file can be invoked as a trigger action, and developers can also use it manually.

Associate changes with change orders

To keep track of work done in response to an engineering change order (ECO), you can use attributes and triggers. For example, to associate a version with an ECO, define **ECO** as an integer-valued attribute type:

```
cleartool mkattype –c "bug number associated with change" –vtype integer ECO
Created attribute type "ECO".
```

Then, define an all-element trigger type, **EcoTrigger**, which fires whenever a new version is created and runs a script to attach the **ECO** attribute.

Trigger definition:

```
cleartool mktrtype –element –all –postop checkin \
–c "associate change with bug number" \
–execunix 'Perl /public/scripts/eco.pl' \
–execwin 'ccperl \\neon\scripts\eco.pl' EcoTrigger
Created trigger type "EcoTrigger".
```

Trigger action script:

```
$pname = $ENV{'CLEARCASE_XPN'};

print "Enter the bug number associated with this checkin: ";
$bugnum = <STDIN>;
chomp ($bugnum);
$command = "cleartool mkattr ECO $bugnum $pname";

@returnvalue = '$command';
$rval = join "",@returnvalue;
print "$rval";

exit(0);
```

When a new version is created, the attribute is attached to the version. For example:

```
cleartool checkin –c "fixes for 4.0" src.c
Enter the bug number associated with this checkin: 2347
Created attribute "ECO" on "/vobs/dev/src.c@@/main/2".
Checked in "src.c" version "/main/2".
```

```
cleartool describe src.c@@/main/2
version "src.c@@/main/2"
...
  Attributes:
    ECO = 2347
```

Associate project requirements with source files

You can implement requirements tracing with *hyperlinks*, which associate pairs of VOB objects. The association should be at the version level rather than at the branch or element level. Each version of a source code module must be associated with a particular version of a related design document. For example, the project manager creates a hyperlink type named **DesignDoc**, which is used to associate source code with design documents.

```
cleartool mkhltype –c "associate code with design docs" \
DesignDoc@/vobs/dev DesignDoc@/vobs/design
Created hyperlink type "DesignDoc".
Created hyperlink type "DesignDoc".
```

The *hyperlink inheritance* feature enables the implementation of requirements tracing:

- When the source module, hello.c, and the design document, hello_dsn.doc, are updated, you create a new hyperlink connecting the two updated versions:

```
cleartool mkhlink -c "source doc" DesignDoc hello.c /vobs/design/hello_dsn.doc
Created hyperlink "DesignDoc@90@/vobs/dev".
```

- When either the source module or the design document incorporates a minor update, no hyperlink-level change is required. The new version *inherits* the hyperlink connection of its predecessor.

```
cleartool checkin -c "fix bug" hello.c
Checked in "hello.c" version "/main/2".
```

To list the inherited hyperlink, use the **–ihlink** option to the **describe** command.

On Linux and the UNIX system:

version that inherits hyperlink-> version to which -> hyperlink is explicitly attached

```
cleartool describe –ihlink DesignDoc hello.c@@/main/2
hello.c@@/main/2
  Inherited hyperlinks: DesignDoc@90@/vobs/dev
/vobs/dev/hello.c@@/main/1 ->
/vobs/doc/hello_dsn.doc@@/main/1
```

On the Windows system:

version that inherits hyperlink-> version to which -> hyperlink is explicitly attached

```
cleartool describe –ihlink DesignDoc hello.c@@\main\2
hello.c@@\main\2
  Inherited hyperlinks: DesignDoc@90@\dev
    \dev\hello.c@@\main\1 ->
    \doc\hello_dsn.doc@@\main\1
```

- When either the source module or the design document incorporates a significant update, which renders the connection invalid, you create a null-ended hyperlink to sever the connection.

```
cleartool mkhlink -c "sever connection to design doc" DesignDoc hello.c
Created hyperlink "DesignDoc@94@/vobs/dev".
```

Figure 52 illustrates the hyperlinks that connect the source file to the design document.

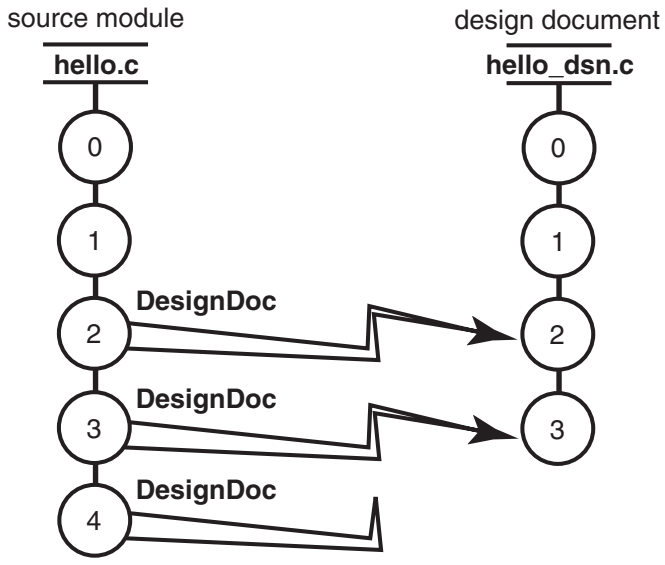


Figure 52. Requirements tracing

Prevent use of certain commands

To control which users can execute certain commands on Rational ClearCase objects, you can create a pair of trigger types.

- One type controls the use of the command on element-related objects
- Another type controls the use of the command on type objects

Both trigger types use the **–nuser** flag to specify the users who are allowed to use the command.

Tip: You cannot use triggers to prevent a command from being used on an object that is not element related or a type object. For example, you cannot create a trigger type to prevent operations on VOB objects or replica objects.

For a list of commands that can be triggered, see the **events_ccase** and **mktrtype** reference pages.

For example, the following commands create two trigger types that prevent all users except **stephen**, **hugh**, and **emma** from running the **chmaster** command on element-related objects and type objects in the current VOB:

```
cleartool mktrtype –element –all –preop chmaster –nusers stephen,hugh,emma \
–execunix 'Perl –e "exit –1;"' –execwin 'ccperl –e "exit (–1);"' \
–c "ACL for chmaster" elem_chmaster_ACL

cleartool mktrtype –type –preop chmaster –nusers stephen,hugh,emma \
–execunix 'Perl –e "exit –1;"' –execwin 'ccperl –e "exit (–1);"' \
–attype –all –brtype –all –eltype –all –lbtype –all –hltype –all \
–c "ACL for chmaster" type_chmaster_ACL
```

When user **tony** tries to run the **chmaster** command on a restricted object, the command fails. For example:

```
cleartool chmaster –c "give mastership to london" london@/vobs/dev \
/vobs/dev/acc.c@@/main/lex_dev
cleartool: Warning: Trigger "elem_chmaster_ACL" has refused to let
chmaster proceed.
cleartool: Error: Unable to perform operation "change master" in
replica "lex" of VOB "/vobs/dev".
```

Certain branches are shared among Rational ClearCase MultiSite sites

Product Note: Rational ClearCase LT does not support Rational ClearCase MultiSite.

If your organization uses Rational ClearCase MultiSite to support development at different sites, you must tailor your branching strategy to the needs of these sites. The standard MultiSite development model is to have a replica of the VOB at each site. Each replica controls (masters) a site-specific branch type, and developers at one site cannot work on branches mastered at another site. (For more information on Rational ClearCase MultiSite mastership, see the *IBM Rational ClearCase Administrator's Guide*.)

However, sometimes you cannot, or may not want to, branch and merge an element. For example, some file types cannot be merged, so development must occur on a single branch. In this scenario, all developers must work on a single branch (usually, the main branch). Rational ClearCase MultiSite allows only one

replica to master a branch at any given time. Therefore, if a developer at another site needs to work on the element, mastership of the branch must be transferred to that site.

Rational ClearCase MultiSite provides the following models for transferring mastership of a branch:

- The push model, in which the administrator at the replica that masters the branch uses the **chmaster** command to give mastership to another replica.

  This model is not efficient in a branch-sharing situation, because it requires communication with an administrator at a remote site. For more information about this model, see the *IBM Rational ClearCase Administrator's Guide*.
- The pull model, in which the developer who needs to work on the branch uses the **reqmaster** command to request mastership of it.

  Tip: The developer can also request mastership of branch types. For more information, see the *IBM Rational ClearCase Administrator's Guide*.

  The pull model requires the Rational ClearCase MultiSite administrators to enable requests for mastership in each replica and to authorize individual developers to request mastership. If you decide to implement this model, you must provide the following information to your Rational ClearCase MultiSite administrator:
  - Replicated VOBs that should be enabled to handle mastership requests
  - Identities (domain names and user names) of developers who should be authorized to request mastership
  - Branch types and branches for which mastership requests should be denied (for example, branch types that are site specific, or branches that must remain under the control of a single site)

  The *IBM Rational ClearCase MultiSite Administrator's Guide* describes the process of enabling the pull model and a scenario in which developers use the pull model. The *Developing Software* online help describes the procedure developers use to request mastership.

Sharing triggers among different types of platform

You can define a trigger that fires correctly depending on the type of platform on which it runs (Linux, the UNIX system, and Windows computers). The following techniques are available:

- "Using different paths or different scripts"
- "Using the same script" on page 189

With one technique, you use different paths or different scripts; with the other technique, you use the same script for all platforms. For more information about sharing triggers, see "Tips for sharing scripts" on page 132.

Using different paths or different scripts

To define a trigger that fires on Linux and the UNIX system; the Windows system; or both types of platform, and that uses different paths to point to the trigger scripts, use the **mktrtype** command with the **–execunix** and **–execwin** options. These options behave the same as **–exec** when the trigger fires on the appropriate platform (Linux and the UNIX system for **–execunix** or the Windows system for **–execwin**). On the inappropriate type of platform, the related script does not run.

This technique allows a single trigger type to use different paths for the scripts or to use completely different scripts on Linux or the UNIX system and the Windows computer. For example:

```
cleartool mktrtype –element –all –nc –preop checkin
–execunix /public/scripts/precheckin.sh
–execwin \\neon\scripts\precheckin.bat
pre_ci_trig
```

Tip: The command line example is broken across lines to make the example easier to read. You must enter the command on one line.

On Linux or the UNIX system, only the script precheckin.sh runs. On the Windows system, only precheckin.bat runs.

To prevent users on a new platform from bypassing the trigger process, triggers that specify only **–execunix** always fail on the Windows system. Likewise, triggers that specify only **–execwin** fail on Linux and the UNIX system.

Using the same script

To use the same trigger script on Linux, the UNIX system, and the Windows system, use a batch command interpreter that runs on all operating systems. For this purpose, the **ratlperl** program is included in the Rational ClearCase configuration. You can use this version of Perl on the Windows system, Linux, and the UNIX system. The commands **Perl** on Linux and the UNIX system and **ccperl** on the Windows system are wrapper programs that run **ratlperl**.

The following **mktrtype** command creates sample trigger type **pre_ci_trig** and names precheckin.pl as the executable trigger script.

```
cleartool mktrtype –element –all –nc –preop checkin  \
–execunix 'Perl /public/scripts/precheckin.pl' \
–execwin 'ccperl \\neon\scripts\precheckin.pl' \
pre_ci_trig
```

Note: In your scripts, you can run **ratlperl** directly. Ensure that you include the following default paths to execute the scripts successfully:

- On Linux and the UNIX system: /opt/rational/common/
- On the Windows system: <*install_location*>\Rational\Common\

The value *install_location* is the root folder in which you installed Rational ClearCase.

Chapter 13. Setting up the base ClearCase integration with Rational ClearQuest

This chapter provides an overview of the base ClearCase integration with Rational ClearQuest and describes how to set up the integration. For information about working in the integration, see *Developing Software* online help.

Overview of the base ClearCase integration with Rational ClearQuest

Rational ClearQuest is used to manage change requests, which report defects or request modifications for a project or product. Each change request is stored as a record in a Rational ClearQuest user database. Rational ClearCase is used to manage versions of the elements that represent a project or product. Each version embodies one or more changes to an element.

What the integration does

The base ClearCase integration with Rational ClearQuest associates one or more Rational ClearQuest change requests with one or more Rational ClearCase versions.

A single change request may be associated with multiple versions. The set of versions that implement the requested change is called the *change set* for that request.A single version may be associated with multiple change requests. These change requests are called the *request set* for that version.

The base ClearCase integration with Rational ClearQuest has the following roles:

- As a Rational ClearCase *project manager*, you specify the conditions under which developers are prompted to associate Rational ClearCase versions with Rational ClearQuest change requests. You can specify VOBs, branches, and element types for which developers can or must associate change requests.
- As a Rational ClearQuest *administrator*, you add Rational ClearCase definitions to a Rational ClearQuest schema. These definitions enable change requests in databases that use the schema to contain and display associated change sets.
- As a Rational ClearCase *developer*, you can:
  - Associate a version with one or more change requests at the time you check in or check out the version.
  - View the change set for a request.
  - Submit queries to identify the change requests that are associated with a project over a period of time.

You can use the Rational ClearQuest Integration Query interface on a Windows system to search for associations (see "About the Integration Query wizard" on page 217).

How the integration works

To work successfully, the base ClearCase integration with Rational ClearQuest must have:

- A VOB enabled for the integration
- A Rational ClearQuest schema upgraded with an integration package

- A configuration file with proper settings

About enabling a VOB and installing triggers

Using the Rational ClearQuest Integration Configuration tool that is supplied with Rational ClearCase integrations, a project manager enables a VOB for the base ClearCase integration. With the tool, you can do the following tasks:

- Select a VOB.
- Specify a policy for checkouts, checkins, and branch types.
- Select the version of triggers to use (either V2-Perl or V1-Visual Basic) and, for V2-Perl triggers, select whether to use a central server configuration and trigger files.
- Specify element type restrictions, branch type restrictions, or both.
- Specify the default Rational ClearQuest record type to be used in associations.

This enabling operation installs into the VOB triggers that fire before or after a Rational ClearCase operation. An integration trigger calls the cqcc_launch script which does the following tasks:

- Examines the environment to decide where the source for the trigger is stored.
- Decides the best version of Perl to use.
- Runs the config.pl file which loads the main trigger source code.

On certain Rational ClearCase operations (checkout, checkin, or cancel checkout), the triggers fire and do the following tasks:

- Connect to the Rational ClearQuest user database
- Run a query for specific change requests in the Rational ClearQuest user database
- Display the change request listing and prompt the developers for information

The developer associates change requests with the elements that are being accessed by a Rational ClearCase client. When the Rational ClearCase operation completes, the trigger stores associations (change set) in the Rational ClearQuest user database and uses hyperlinks to store matching information (request set) in Rational ClearCase storage. Thereafter, the result set of the Rational ClearQuest record that has associations displays in the **ClearCase** tab of the client window the names of the elements (the change set) that are associated with the defect (a change request). For more information, see “Setting policies and installing triggers in a ClearCase VOB” on page 197.

Query support

Queries in a Rational ClearQuest user database are run by the integration either from a trigger firing or from an explicit developer selection in one of the integration interfaces (either **Browse** in the GUI or **Queryname** in the command line interface). You can make these queries available in the following ways:

- Provide queries in either the **Personal Queries** or **Public Queries** folder in a Rational ClearQuest user database (Rational ClearQuest queries).
- Define queries in the integration configuration file (local queries).

By default, only local queries are available to developers. The Web interface supports only local queries. For more information, see “Controlling query usage” on page 210.

In making queries available to developers, keep in mind the following limitations:

- Currently there is no support for queries that prompt the developer for runtime parameters. That is, Rational ClearQuest queries that contain dynamic filters are **not** supported, and, if dynamic filters are used in queries in the integration, they act as if default values were selected without user interaction.
- Only single-line text fields in query results are supported in both Rational ClearQuest and local queries. The integration provides no special handling for multiple-line output within text fields.
- The first field in the query result set must be the Rational ClearQuest record ID. For more information, see "SetResultSet" on page 206.

About locally stored information

Between invocations, the integration stores information in local files on the end user host. The locally stored information is used to track operations that span multiple trigger calls. This information is kept in files with names in the format `.cqcc_`*`text`*. On Linux and the UNIX system, the files are stored in the user home directory. On the Windows system, the files are stored in the user profile folder under \Application Data\Rational\CQCC\.

Tip: The Rational ClearQuest login information is encrypted in a locally stored file (.cqcc_params). To change to an alternate Rational ClearQuest login account, remove this file. If the file is not found, a new login is forced when the trigger is next called and the new information is stored in the file.

Product note: Because the integration stores information in one central location dependent on the identification of the user, it cannot be safely run in multiple shells or windows on the same machine at the same time under the same user identification. And, on Linux and the UNIX system, the integration cannot be run under the same user identification on different machines because the information is stored in the user home directory.

About trigger versions

In the Rational ClearQuest Integration Configuration tool, you specify the version of the trigger that the integration should use on checkin and checkout operations. You can select **V2-Perl** for use on Linux and the UNIX system and either **V2-Perl** or **V1-Visual Basic** for use on the Windows system. Prior to Rational ClearCase version 2002.05.00, the integration used a Visual Basic trigger (V1-Visual Basic) through the CQIntSvr interface on the Windows system and a Perl trigger (V1-Perl) through the Rational ClearQuest Web server interface on Linux and the UNIX system. In Rational ClearCase version 2002.05.00, a Perl trigger (V2-Perl) was added that runs on the Windows system and on Linux and the UNIX system. (This trigger is also available in a patch to Rational ClearCase Release 4.2).

You can choose the following triggers:

- V2-Perl refers to the cross-platform Perl trigger. The same trigger code works on Linux, the UNIX system, and the Windows system.
- V1-Visual Basic refers to the separate Visual Basic implementation on the Windows system.
- V1-Perl refers to the earlier Perl trigger on Linux and the UNIX system that is superseded by the V2-Perl trigger. The V1-Perl trigger is no longer supported.

The V2-Perl trigger provides both a text-based user interface for developers who use the **cleartool** command-line interface and a graphic user interface (GUI) for developers who use one of the Rational ClearCase GUIs, for example, ClearCase Explorer (on the Windows system) or **xclearcase** (on Linux and the UNIX system).

If you are configuring the integration for the first time, use the V2-Perl trigger. If you currently use the V1-Visual Basic trigger, evaluate the V2-Perl trigger and consider migrating to it.

The V2-Perl trigger uses a configuration file called config.pl, which specifies your local configuration parameters or centrally-defined configuration parameters.

About the integration package

To install the base ClearCase integration with Rational ClearQuest requires that the Rational ClearQuest schema designer add the ClearCase 1.0 package to an existing schema. The package accompanies the Rational ClearQuest product and supplies new stateless records to target record types. The Rational ClearQuest administrator then upgrades the Rational ClearQuest user database with the revised schema. Because you cannot remove a package after you add a package to a schema, the Rational ClearQuest documentation suggests that you use a test environment before you modify your production environment. For more information about installation, see "Setting up the Rational ClearQuest user database for base ClearCase" on page 196.

About the configuration file

In the integration configuration file config.pl, the project manager sets the options and policies that determine the operational details. The config.pl file provides the following information and control:

Rational ClearQuest connection information
: Determines whether you connect to the user database with the Rational ClearQuest client or the Rational ClearQuest Web interface, specifies to what targets you connect (database set, user database, or record types), and controls what query information is presented to the developer (see "Connecting Rational ClearCase clients and a Rational ClearQuest user database" on page 203)

Rational ClearCase policies
: Determines whether single or multiple associations are allowed, which queries are allowed, and what restrictions are placed on queries (see "Making policy choices" on page 209)

Performance
: Controls use of options to improve processing efficiency, including auto-batch, auto-association, and batch-series; obtaining associations from Rational ClearCase comments; and commit after the checkin (see "Enhancing performance" on page 211)

Debugging
: Provides logging and timing information (see "Debugging and analyzing operations" on page 215)

Policy regarding customization and support

Project managers can make changes in the integration by editing the config.pl file. Integration source code changes that are made outside the config.pl file are possible, but are not supported.

The following is supported: the integration as provided and local configuration changes that you make by using the configuration parameters in the configuration file (config.pl).

Source code changes to the integration are **not** supported although source code is provided as Perl scripts. If you request additional information from IBM Customer

Support, you can receive internal documentation that describes making changes through creating and using subclasses in the Perl trigger. To enable local sites to make judicious changes, the requested internal documentation describes the use of subclasses to separate changes that you make and the main body of code that is supplied with the product. For information on using the subclasses, see "Customizing the integration" on page 217. But local source code changes cannot be and are not supported by IBM Customer Support.

Checklist of configuration steps

Refer to Table 5 for a checklist of steps to configure the base ClearCase integration with Rational ClearQuest.

Table 5. Configuration checklist

Location	Step
On the ClearCase system	Create a ClearCase VOB and mount it.
	Create views.
On the ClearQuest system	Set up your schema repository, user database, and the ClearQuest client.
	In the user database, open Designer, open the Package Wizard, select **ClearCase 1.0** package, and apply it to the schema.
	Enable the ClearCase 1.0 package for record types in the schema.
	Check in the schema and upgrade the user database.
On the ClearCase system	Start configuring the integration (see "To start the Rational ClearQuest Integration Configuration tool" on page 199).
	Select the appropriate ClearCase VOB. (If the VOB is on Linux or the UNIX system, set up the environment for inter-operation.)
	Select desired policy settings (Checkin, checkout, branch-type restrictions, and appropriate element-type restrictions).
	Enter record type. (This allows for only a single default preference, but multiple types are supported. See "Defining the Rational ClearQuest user database and database set" on page 204.)
	Select trigger type (**V2-Perl** in this case) for Windows and Linux and the UNIX system.
	Specify the config.pl path and set trigger scripts option to use a shared location (see "Using a shared configuration file and triggers" on page 198).
	Modify config.pl with the appropriate configuration parameters (see "Summary of configuration parameters" on page 201).
	Use **cqcc_launch -test** to see whether the basic communications is working (see "Testing the integration" on page 216).

Planning for the base ClearCase integration with Rational ClearQuest

After you successfully install both Rational ClearCase and Rational ClearQuest products, you should establish a test environment in which to evaluate what works well in your organization. Establishing a test environment requires someone who has Rational ClearCase administrator rights (for example, to create and own a VOB) and someone who has Rational ClearQuest administrator rights (for example, to create a user database and install packages and upgrade schemas). The evaluation can involve the following activities:

- Establish in a test environment a VOB that is like your production VOB and install the triggers to become familiar with the way the integration works. Ensure that the VOB is not UCM (that is, it will not contain UCM components).
- Create a production user database for testing purposes.
- Use the procedures defined in this chapter and the tasks defined in the online help to set up the Rational ClearQuest user database that you created for testing purposes, to configure the base ClearCase integration with Rational ClearQuest, and to connect the environments.
- Try different policies, performance options, and configuration options to see what works best.
- Use the test VOB to confirm that the configuration and options work.

An important deployment decision is whether to use a local or a central configuration. By default, the configuration file and trigger source files reside on each client machine. Because this arrangement is difficult to maintain, it is better to define a central location on which you maintain one copy of the configuration file and trigger source files. In the central arrangement, any changes are made in only one location and the one location is more easily made secure.

When you settle on an environment that works, communicate to developers how to work with the integration and publicize the policies that are to be in effect in the production VOBs. Then, install the triggers in a production VOB, apply the same configuration information that you established in the tested configuration, and perform the same basic tests again.

Setting up the Rational ClearQuest user database for base ClearCase

Before developers can associate Rational ClearCase versions with Rational ClearQuest change requests, the Rational ClearQuest administrator needs to configure Rational ClearQuest in the following manner:

- Add Rational ClearCase definitions to a Rational ClearQuest schema.

  Use the Rational ClearQuest Designer's Package Wizard to add the definitions (see "Adding Rational ClearCase definitions to a Rational ClearQuest schema" on page 197). You associate the Rational ClearCase definitions with one or more record types and their related forms. Each form then contains a **ClearCase** tab that displays the change set for a change request.
- Use the Rational ClearQuest Designer to upgrade the database with the new version of the schema. See the *Upgrading an existing database* topic in the Rational ClearQuest Designer Help. If you move the integration to a different database, repeat this step for that database.

Adding Rational ClearCase definitions to a Rational ClearQuest schema

A Rational ClearQuest schema contains the attributes associated with a set of Rational ClearQuest user databases, including definitions of record types, fields, and forms. The Rational ClearQuest administrator must add some Rational ClearCase definitions to the schema that the database uses. To do so, the Package Wizard within Rational ClearQuest Designer is used.

Note: If you are using a version of Rational ClearQuest earlier than version 2.0, use the Rational ClearQuest Integration Configuration tool to add Rational ClearCase definitions to a Rational ClearQuest schema.

To add Rational ClearCase definitions to and upgrade a Rational ClearQuest schema

1. Click **Start > Programs > IBM Rational > IBM Rational ClearQuest > ClearQuest Designer**.
2. In Rational ClearQuest Designer, click **Package > Package Wizard**.
3. In the Package Wizard, look for the Rational ClearCase 1.0 package. If this package is not listed, click **More Packages**, and add it to the list from the Install Packages window.
4. Select **ClearCase 1.0**, and click **Next**.
5. Select the schema for the Rational ClearQuest user database that you want to use in the integration. Click **Next**.
6. Do one of the following actions:
   - If you use the V2-Perl trigger, you can specify multiple record types. Click **Finish**.
   - If you use the V1-Visual Basic trigger, select the record type of Rational ClearQuest records to be associated with Rational ClearCase versions. Use this record type when you specify the **ClearQuest Record Type** field in the Rational ClearQuest Integration Configuration tool (see "To start the Rational ClearQuest Integration Configuration tool" on page 199).
7. Click **File > Check In** to save the new version of the schema.
8. Click **Database > Upgrade Database** to upgrade the Rational ClearQuest user database with the new version of the schema.

Setting policies and installing triggers in a ClearCase VOB

Before developers can associate Rational ClearCase versions with Rational ClearQuest change requests, you need to configure Rational ClearCase as follows:

- Using the Rational ClearQuest Integration Configuration tool, for each VOB, set policies that determine the conditions under which developers are prompted to associate versions with change requests. You can specify that developers are prompted on checking out a version, checking in a version, or both. You can also specify that prompting occurs only for some branch types or element types. Associations of checked-in versions with change requests can be either optional or required.
- Using the Rational ClearQuest Integration Configuration tool, select the trigger type that is to be used. If you use the V2-Perl trigger, you need to modify a configuration file to set database connectivity information and additional policy parameters (see "Editing the configuration file" on page 200).

The base ClearCase integration with Rational ClearQuest uses Rational ClearCase element triggers on **cleartool checkin** (preoperation and postoperation), **checkout** (postoperation), and **uncheckout** (preoperation) commands to allow developers to associate versions with Rational ClearQuest change requests.

To start the Rational ClearQuest Integration Configuration tool, see "To start the Rational ClearQuest Integration Configuration tool" on page 199.

Using a shared configuration file and triggers

In the Rational ClearQuest Integration Configuration tool, if you set **V2-Perl**, you have the following options:

- Use a local configuration file and local trigger scripts (default).
- Use a shared, centrally located configuration file and local copies of trigger scripts.
- Use a shared, centrally located configuration file and shared trigger scripts.

By default, the **Path** field is filled in with `CQCC/config.pl`, the path to the configuration file. In this path, the value CQCC resolves to the following value on each local client:

ccase–home–dir`/lib/CQCC`

By default, the cqcc_launch script is installed in the following location on each local client:

ccase–home–dir`/bin`

On the Linux and UNIX system, the script is cqcc_launch; on the Windows system, the script is cqcc_launch.bat.

In this configuration, each time the integration starts or a trigger fires on a client machine, the local instances of the configuration file, the cqcc_launch script, and trigger scripts are run. This is the default configuration.

To define a central location for accessing a shared, centrally located configuration file, in the Rational ClearQuest Integration Configuration tool, provide a sitewide path to the configuration file. To provide the path, change the **Path** field to a UNC path to a Windows system or to a full path to the Linux or the UNIX system that contains the configuration file and the cqcc_launch script. The integration uses that one central configuration file and the cqcc_launch script for all users of VOBs that are enabled for Rational ClearQuest. All users will run the shared, centrally located copy of the config.pl file. This configuration uses local copies of trigger scripts.

If you set **Use trigger scripts in Path directory**, you can use shared, centrally located trigger scripts with the centrally located configuration file. This setting allows you to centralize the trigger source code in one location. Triggers installed with this option look for the configuration file, cqcc_launch script, and trigger source code in the same directory in which the configuration file is located. This is the best configuration because it is easiest to secure and maintain.

If you use a centrally located configuration file or trigger scripts, perform some housekeeping procedures to set up the central location. This set-up involves copying the requisite files from the installed location on a Rational ClearCase system to the central location. For more information, see the online Help in the Rational ClearQuest Integration Configuration tool.

In support of the configuration, the cqcc_launch script provides as a convenience the following command-line option:

```
cqcc_launch -vob
```

This command searches the current VOB for its checkout trigger to determine the correct paths and configuration file to use in launching the script.

Installing triggers in a VOB on Linux and the UNIX system

If the VOB resides on Linux or the UNIX system, install the triggers from a Windows system that uses the same registry server as the system on which the VOB resides. The VOB tag must be imported from the region in which the Linux and the UNIX systems run to the region in which the Windows system runs. Then, you can use the Rational ClearQuest Integration Configuration tool.

To start the Rational ClearQuest Integration Configuration tool

1. Log into the system as the VOB owner.
2. Use one of the following methods:
   - On a system that runs Rational ClearCase, click **Start > Programs > IBM Rational > IBM Rational ClearCase > Administration > Integrations > ClearQuest Integration Configuration**.
   - On a system that runs Rational ClearCase LT server, click **Start > Programs > IBM Rational > IBM Rational ClearCase LT > ClearQuest Integration Configuration**.
   - Enter **cqconfig** at the command prompt.
3. In **Select a VOB**, select a VOB tag in the list.
4. In the **Windows Trigger Selection** or **UNIX Trigger Selection** fields of the Rational ClearQuest Integration Configuration window, specify which trigger you want to use by clicking one of the following options:
   - **V2-Perl** which refers to the cross-platform Perl trigger.
   - **V1-Visual Basic** which refers to the Visual Basic triggers (the Windows system only).

   For more information about triggers, see "About trigger versions" on page 193.

For information about completing the other fields in the tool, click **Help** within the tool.

To specify multiple record types

1. In the Rational ClearQuest Integration Configuration tool, specify the DEFAULT record type. See "To start the Rational ClearQuest Integration Configuration tool."
2. When you edit the config.pl file, specify multiple record types that the user can select when the integration runs. See "Defining the Rational ClearQuest user database and database set" on page 204.

To list triggers installed in a VOB

Enter the **lstype** command to see the types of triggers in a specific VOB. For example:

On Linux and the UNIX system:

```
cleartool lstype -kind trtype -invob /vobs/my_vob
```

On the Windows system:

```
cleartool lstype -kind trtype -invob \my_vob
```

Information about the triggers defined in the specified VOB are displayed. For example:

```
cleartool lstype -kind trtype -invob \my_vob
08-Jun.08:07  username trigger type "cq_ci_trigger"
  "ClearQuest Integration"
08-Jun.08:07   username    trigger type "cq_co_trigger"
  "ClearQuest Integration"
08-Jun.08:07   username    trigger type "cq_postci_trigger"
  "ClearQuest Integration"
08-Jun.08:07   username    trigger type "cq_unco_trigger"
  "ClearQuest Integration"
```

Use the describe command to see information about a specific base ClearCase integration with Rational ClearQuest trigger. For example:

On Linux and the UNIX system:

```
cleartool describe trtype:cq_co_trigger@vob:/vobs/my_vob
```

On the Windows system:

```
cleartool describe trtype:cq_co_trigger@vob:\my_vob
```

Information about the postoperation checkout trigger is displayed.

Quick start for evaluations

The default configuration file is set to use the SAMPL user database that is provided for evaluations in the Rational ClearQuest configuration. You can test the integration with the SAMPL Rational ClearQuest user database. If Rational ClearQuest is installed on the client machine, the base ClearCase integration with Rational ClearQuest uses the Rational ClearQuest Perl API to communicate with the Rational ClearQuest user database.

If Rational ClearQuest is not installed on the client machine, the integration uses the Rational ClearQuest Web Interface to communicate with the Rational ClearQuest user database. To use the Web interface, set server name in the configuration file or use the optional environment variable (see "Establishing the Rational ClearQuest Web interface" on page 203).

Editing the configuration file

The configuration file contains parameters that define local policy choices and how to access Rational ClearQuest user databases. Before you can edit the configuration file, change its permissions to make it modifiable.

Overview of the configuration file

The configuration file is set to access the Rational ClearQuest **SAMPL** user database and use the **defect** record type. To use the integration with a different Rational ClearQuest user database or record type, you need to change configuration parameters (see "Summary of configuration parameters" on page 201). The configuration file contains comments that describes the values that are allowed of configuration parameters. Information about the integration and configuration parameters that could not be put in the product documentation is made available in the `README` file in the same folder as the configuration file.

Locating the configuration file

When Rational ClearCase is installed, the following file is placed on the system:

ccase–home–dir\lib\CQCC\configTemplate.pl

If there is no existing file with the same name, the following file is also placed on the system:

ccase–home–dir\lib\CQCC\config.pl

If the local configuration file config.pl exists, the installation tries to avoid replacing that file because it contains your changes. The installation action depends on the platform type.

- On Linux and the UNIX system, no new version of the config.pl file is installed. The current version is preserved.

  To take advantage of the latest changes, copy the new information from the configTemplate.pl file to the config.pl file.
- On the Windows system, if the installed version is not modified (based on comparing the date stamp on the file and the original installation date), the latest version of the config.pl file is installed.

  If the installed version is modified, no new version of the file is installed. The current version is preserved.

To see the latest changes in configuration information, compare the config.pl and the configTemplate.pl files. If you make changes to your configuration, edit the configTemplate.pl file and save the changes to both configTemplate.pl and config.pl. By locating the configuration file in a central place, you simplify the task of implementing the latest changes. For more information, see "Using a shared configuration file and triggers" on page 198.

If you uninstall Rational ClearCase, both configTemplate.pl and config.pl are removed from the system. However, on Linux and the UNIX system, the config.pl file is first copied to the standard file preservation area and then removed from its original location.

Configuration file use and format

The config.pl file is a Perl script that contains comments and commands for local site configuration information. You change one or more configuration parameters in one Perl function called **ConfigureTrigger**. Remaining functions in the script call that function to implement the change. The script runs to establish the configuration and to call the main trigger routine. Each configuration parameter has the following general format:

```
# Name: [ENV] (Default: value)
# values
# Sample to enable
```

The [ENV] notation indicates whether the related configuration parameter can be set as an environment variable in the developer's context. In the Default: entry, *value* indicates what the integration uses if you do not edit the sample line. The line with *values* shows all possible selections that you can use in the *Sample to enable* line. Edit the sample line with your change and remove the pound (#) character from the beginning of the line to enable your change.

Summary of configuration parameters

Configuration parameters provide the following functional capabilities:

- "Connecting Rational ClearCase clients and a Rational ClearQuest user database" on page 203
- "Making policy choices" on page 209
- "Enhancing performance" on page 211
- "Debugging and analyzing operations" on page 215

Table 6 summarizes the configuration parameters that you can set in the config.pl file. Some of the configuration parameters have related environment variables of the same name that you or your developers can set in their local context. These parameters are noted by Yes in the Locally settable column of Table 6. All parameters are described more fully in the subsequent sections about functional capabilities.

Table 6. Configuration parameters summary

Configuration parameter	Description	Locally settable
CQCC_ASSOC_BATCH_CONFIRM	Displays a window that confirms that the batch completed successfully.	Yes
CQCC_ASSOC_BATCH_ENABLE	Allows delay in processing multiple Rational ClearQuest association transactions until end of single Rational ClearCase operation or user-defined batch	Yes
CQCC_ASSOC_BATCH_SERIES	Specifies that a user-defined series is active and normal Rational ClearCase series-end processing should be suppressed	Yes
CQCC_AUTO_ASSOCIATE	Sets one or more change requests for automatic association on checkout and checkin without user interaction	Yes
CQCC_AUTO_ASSOCIATE_ENABLE	Specifies whether developers can use CQCC_AUTO_ASSOCIATE	No
CQCC_COMMENT_PATTERN	Sets pattern by which developers can make associations in a checkout or checkin comment	No
CQCC_CQWEB_ONLY	Forces use of Rational ClearQuest Web interface	No
CQCC_CQWEB_VERSION	Specify either 2.0 (Java™) or "1.0" (ASP with IIS) as Rational ClearQuest Web server protocol	Yes
CQCC_DATABASE_ENTITY_LIST	Defines logical name of database and related record types (entities) that support associations.	No
CQCC_DATABASE_SET	Database set name (connection) for one of multiple schema repositories; used with CQCC_DATABASE_ENTITY_LIST	Yes
CQCC_DEBUG	Controls level of output generated for problem diagnosis; 0 (none), 1 (basic), 2 (verbose)	Yes
CQCC_GUI_ENABLE	Allows use of Perl/TK graphic user interface	Yes
CQCC_LOG_OUTPUT	Records to a log file messages for problem diagnosis	Yes

Table 6. Configuration parameters summary (continued)

Configuration parameter	Description	Locally settable
CQCC_MULTIPLE_ASSOCS	Allows or prevents multiple defects to be associated with change	No
CQCC_MULTISITE	Enables Rational ClearCase MultiSite support	No
CQCC_POSTCHECKIN_COMMIT	Allows commitment of associations in Rational ClearQuest user database to be delayed until checkin completes in Rational ClearCase VOB	No
CQCC_QUERY_ENABLE	Allows developer-selected queries for making associations	No
CQCC_QUERY_FILTER	Controls which queries are presented to developers for associations	No
CQCC_REPLICA_NAME	For Rational ClearQuest Web client to specify user database replica name	Yes
CQCC_RESTRICTIONS_TIMEOUT	Specifies number of seconds that a restrictions check can be reused during batch processing	Yes
CQCC_SERVER	Name of Rational ClearQuest Web server	Yes
CQCC_SERVERROOT	Name of folder in which Rational ClearQuest Web server is located	Yes
CQCC_SERVER_SSL	Enables secure communications for the Rational ClearQuest Web connection.	Yes
CQCC_TIMER	Allows recording of internal timing data	Yes
CQCC_WEB_DATABASE_SET	CQCC_DATABASE_SET for Web server	Yes

Connecting Rational ClearCase clients and a Rational ClearQuest user database

You have multiple options for connectivity.

Establishing the Rational ClearQuest Web interface

If the Rational ClearQuest client is installed on a developer system, it is used by default. If the Rational ClearQuest client is not installed on a developer system, the Rational ClearQuest Web Interface is the default interface. To enable a client to use the Rational ClearQuest Web Interface, set the following configuration parameters in the configuration file or the environment variables from the command-line prompt:

Configuration parameter or environment variable

CQCC_CQWEB_VERSION
Because two versions are supported, configure the correct protocol. 2.0 (Rational ClearQuest Web Java protocol) is the default. 1.0 (Rational ClearQuest Web ASP) is for earlier Rational ClearQuest Web servers.

CQCC_SERVER
Specifies the name of the host at which the Rational ClearQuest Web server resides and uses port 80. To specify a different port, add to the host name

a colon (:) and the number of the port; for example, `myhost:81` specifies that port number 81 be used instead of port 80.

CQCC_SERVERROOT
Specifies the root directory in which the Rational ClearQuest Web Interface files are installed; usually cqweb.

CQCC_SERVER_SSL
Specify TRUE to enable secure communications for the Rational ClearQuest Web connection. The URL is set to use `https:`. Specify FALSE (the default) to use `http:` as the protocol.

If you enable secure communications, more time is used to establish a new connection when a trigger fires and a connection to a Rational ClearQuest user database is required relative to establishing an ordinary connection.

Configuration parameter only

CQCC_CQWEB_ONLY
Set this to force use of the Rational ClearQuest Web Interface even if the Rational ClearQuest client is installed.

Defining the Rational ClearQuest user database and database set

By default, the base ClearCase integration with Rational ClearQuest uses the SAMPL user database with a defect record type. Use the configuration parameter CQCC_DATABASE_ENTITY_LIST in the configuration file to specify the logical name of the Rational ClearQuest user database and record types in your production environment that support associations. You can specify multiple record types per database. For each Rational ClearQuest user database, you must provide, in a list format, record types (entities) that accept Rational ClearCase associations. This list is used to provide choices to the developers when they make associations.

Use the following format:

```
dbname1: entity1,entity2; dbname2: entity3,entity4
```

For an entity, specify the record type. For example:

```
&SetConfigParm("CQCC_DATABASE_ENTITY_LIST","SAMPL: defect");
```

The database name is case sensitive. For each record type that you specify, describe the field names in the schema definition (see “Establishing the schemas” on page 205).

Database sets (connections) allow developers to select from multiple schema repositories when they start a ClearQuest client or the base ClearCase integration with Rational ClearQuest. Only one database set is supported. In the integration, if your site uses multiple database sets, define CQCC_DATABASE_SET to specify the database set that is used for the integration and for the Rational ClearQuest native client interface. In CQCC_DATABASE_SET, supply the database connection name that you created in the Rational ClearQuest Maintenance tool. For example:

```
&SetConfigParm("CQCC_DATABASE_SET", "cqcc_db");
```

If you also use the Rational ClearQuest Web interface, supply this name in CQCC_WEB_DATABASE_SET. For example:

```
&SetConfigParm("CQCC_WEB_DATABASE_SET", "cqcc_db");
```

If you set both configuration parameters, the names can differ.

Establishing the schemas

Schemas provide more information about Rational ClearQuest record types (entities) to identify field names and query definitions. If the schemas share basic information in common, the schemas can describe one or more record types but do not distinguish between different user databases.

Overview of DefineCQSchema

In the configuration file, edit the DefineCQSchema() definition to provide the required and optional field names. Map your database field names to conventions that are used in SAMPL database. This mapping allows query definitions to be more generic, but also tells the integration what field names to use for its own operations, for example, internal queries that it has to do. Certain field names are required, for example, ID, OWNER, STATE, and HEADLINE.

Tip: The ID field must be first in the results set. The field is used to select records for later operations.

Add other field names for your convenience, but these other mappable names are not required by the integration. Field names can be used directly in queries that you define if at least the required fields are made known to the integration.

The integration uses a **CQSchema** object to relate the field names that are defined to the field names that you use in your local record types. A CQSchema is loosely related to a Rational ClearQuest schema, but it really just describes record type fields and queries that are needed by the integration. It accomplishes the following tasks:

- Defines a new CQSchema and provides a set of similar record types (entities) and properties (**DefineCQSchema**)
- Relates the field names of the defect record type to those used by the record types that you specify (**ChangeFieldMap**)
- Defines one or more queries (**SetQuery**) that are used by the trigger to provide the QUERY option that the developers see and specifies the final RESTRICTIONS check that is made before associations are made
- Defines the query output format (**SetResultSet**)

DefineCQSchema

```
$s = DefineCQSchema(NAME=>name, ENTITY_LIST=>entityList,
              RESTRICTIONS=>queryName);
```

NAME

Used only to provide a unique reference. Change the NAME for the new schema.

ENTITY_LIST

Provides one or more record types. For example, `"defect,feature,patch"`.

RESTRICTIONS

Optional; refers to a query within the schema that defines conditions that new associations must meet. For example, the change request must be in a specific state. If you do not provide any restrictions, any existing change requests can be used.

For example:

```
$s = &DefineCQSchema(NAME=>"MainSchema",
                        ENTITY_LIST => "defect",
                        RESTRICTIONS => "STANDARD");
```

ChangeFieldMap

```
$s->ChangeFieldMap( name=>value, ...)
```

ChangeFieldMap() is a hash list that maps standard field names (for example, Owner) to a local name that you specify in *value*. The *name* is usually the name of a field in the SAMPL defect record type, but you can use any name that needs mapping in a query. The field map names are used in query or result set lists as variables. For example:

```
$s->ChangeFieldMap(OWNER => "Owner",
                       STATE => "State",
                       ID => "id", #Note: ID shouldn't need to change
                       HEADLINE => "Headline",
                       PRIORITY => "Priority",
                       SEVERITY => "Severity",
                       RATL_MASTERSHIP => "ratl_mastership",

                       # Other mappable names
                       QUERY_STATES => "Submitted,Assigned,Opened",
                       MODIFY => "modify"
                       );
```

The value for `<OWNER>` is translated to the local value that you provide in the `OWNER=>"Owner"` pair.

Tip: If you use multiple record types, the field names in the mapping must be present in all record types.

SetQuery

```
$s->SetQuery(queryName, clause1, clause2, ...);
```

SetQuery() provides a named set of filter clauses for the query. It can directly use your local field names or rely on the field map to translate them when left angle brackets (<) and right angle brackets (>) are used. The value *queryName* is local to the current schema. STANDARD is used for the QUERY option, but you can define a second query to present an alternate restrictions query. The value *clauseN* is a string defining a condition in the format `field operator value`. For example:

```
$s->SetQuery("STANDARD",
                "<OWNER> eq <*USER*>",
                "<STATE> in <QUERY_STATES>");
```

Use only commas to separate multiple values.

Tip: SetQuery() is effectively an AND group of clauses. OR or NOT connectives are not supported.

SetResultSet

```
$s->SetResultSet(queryName,ID,fieldList,formatString);
```

SetResultSet() identifies the fields that the query should return.

The value *fieldList* is a comma-separated list of field names. The first field must be ID (generated by Rational ClearQuest) because the integration must be able to identify and work with the selected records. A customer-generated ID field does not meet this requirement.

The value *formatString* is a standard **printf** format string. For example:

```
$s->SetResultSet("STANDARD",
                    "<ID>,<STATE>,<PRIORITY>,<SEVERITY>,<HEADLINE>",
                    "%s %-9.9s %1.1s %1.1s %-45.45s");
```

Tip: If you use both local and Rational ClearQuest queries (see "Controlling query usage" on page 210), the SetResultSet definitions must contain the exact fields in the exact order as your queries.

Sharing a CQSchema

A CQSchema can be shared by different record types if the record types share the small subset of field names that the integration uses. If you have record types that are too different to share one schema, you can define additional CQSchema objects. To define an additional CQSchema, copy the following commands: DefineCQSchema, ChangeFieldMap, SetQuery, and SetResultSet and modify them for additional definitions. Supply the mapping for each schema.

Establishing Rational ClearCase MultiSite support

To use Rational ClearCase MultiSite support, set the CQCC_MULTISITE configuration parameter in the configuration file to "TRUE". To use the Rational ClearQuest Web client with Rational ClearCase MultiSite support, set CQCC_REPLICA_NAME to the name of the VOB replica. This setting is needed because the Rational ClearQuest Web client cannot determine the replica name.

About code page conversion

To communicate with Rational ClearCase, Rational ClearQuest, and the user interface, the integration uses character strings to represent query results and association information. When clients and servers use only ASCII data, the communication is simple. But if characters require more than single-byte ASCII representations (for example, to provide accented or multiple-byte characters for other languages), the integration must process strings that are encoded in different code pages. For example, strings that are sent between the client and the server must be converted from the ″local″ code page used on the client to a more universal multiple-byte representation (for example, UTF-8 encoding form) on the Rational ClearQuest Web server.

The integration code page conversion process

The integration uses the following conversion process:

- Characters are converted between local code page and UTF-8 encoding form, where appropriate. Because performance overhead is negligible, disabling conversion for performance reasons is not encouraged. Code page conversion is only supported in Rational ClearCase 7.0 and later releases. Thus, conversion is ignored by clients running earlier versions of Rational ClearCase. If code page conversion errors occur, they are reported to the user as type error or type fatal, depending on the context as shown in the following example:

```
Conversion error for CQWebJava field 'Resultset.Rows'
Codepage conversion error: No valid character in output character set.
The string is 'French: Les na?fs ?githales h?tifs pondant ? No?l o? il
g?le sont
 s?rs d'?tre d??us et de voir leurs dr?les d'?ufs ab?m?s'
The following characters are invalid:
U+c3af at 14
U+c3a6 at 18
U+c3a2 at 29
U+c3a0 at 43
U+c3ab at 47
U+c3b9 at 51
U+c3a8 at 57
```

```
U+c3bb at 67
... (error message truncated)
Caller: \\serverA\central\CQCC\../CQCC/CQWebJava.pm Line: 896
```

The error message indicates the context in which the string is being used and the string or strings involved; and uses question mark (?) characters to indicate inconvertible characters. Additional information is provided on each individual character that can not be converted.

- If code page conversion is not enabled or is not yet supported on the client, the integration performs ASCII checking to filter non-ASCII data in strings. Strings that contain non-ASCII data are treated as code page conversion errors. Similar information is reported to identify the context and string contents. Performance overhead for ASCII checking is negligible. With ASCII checking enabled, the integration prevents transmission of incorrectly encoded data between client and server and minimizes potential data corruption and misrepresentation.
- The integration can produce large error messages if there are either code page conversion or ASCII check errors. There is a maximum number of lines that are displayed in a conversion error message, after which additional data is truncated. The number of lines is set to 50 by default, but, if your environment requires, the limit can be adjusted to provide less or more information.

The contents of the configuration file

The config.pl file can contain non-ASCII values (for example, non-ASCII database names) either as local code page when all client machines are using the same code page or UTF-8 values when more than one local code page is in use. If you use UTF-8 values in the config.pl file, perform the following actions:

1. Use a UTF-8 capable editor (for example, Windows Notepad) to open the config.pl file.
2. Include the following line at the top of the config.pl file, before the `use CQCC::TriggerCQCC;` statement:

   ```
   use utf8;
   ```

   This line tells Perl that the file contains UTF-8 strings.
3. Use the **Save As** command and, in **Encoding**, select **UTF-8** to save the config.pl file in UTF-8 format.

   This preserves the strings that you entered in the config.pl file as UTF-8 data.

Configuration parameters for code page conversion

The integration code page conversion can be disabled or modified through configuration parameters. Because most sites should not need to modify the parameters, they are not documented. For more detailed information, contact your customer support representative.

Testing the configured connections

If you have the connectivity information set up in the configuration file, you can test the connectivity. From a command line in a view context, change directory to a folder in your VOB and type a test command. For example:

```
cqcc_launch -vob -test
```

This command connects to your Rational ClearQuest user database, makes some test associations from Rational ClearCase elements to a Rational ClearQuest record, and then removes the test associations. For information about errors, see “Troubleshooting the configured connections” on page 209.

Troubleshooting the configured connections

If you have trouble connecting to a Rational ClearQuest user database through the integration in the connectivity test (see "Testing the configured connections" on page 208), the debug output produced in the connectivity test provides some help. The following are common reasons for connection problems:

- Wrong Rational ClearQuest database set

  The database set is not required in the CQCC_DATABASE_SET configuration parameter if only one database set is being used or if the database set that you want is the first one (with the version number). Otherwise, a database set must be provided (see "Defining the Rational ClearQuest user database and database set" on page 204).
- Cannot find the Rational ClearQuest user database name

  Check the CQCC_DATABASE_ENTITY_LIST configuration parameter (see "Defining the Rational ClearQuest user database and database set" on page 204).
- Record types do not have the Rational ClearCase package installed in the Rational ClearQuest schema

  An error message typically reports that there is no cc_change_set object, a stateless record type that is defined by the ClearCase 1.0 package. Consult your Rational ClearQuest administrator (see "Setting up the Rational ClearQuest user database for base ClearCase" on page 196).
- Rational ClearQuest Web client problems

  Try to replace the ASP interface with Rational ClearQuest Web and vice versa (see "Establishing the Rational ClearQuest Web interface" on page 203).
- Rational ClearQuest Web client cannot connect

  First, see whether you can use a browser to connect to the Rational ClearQuest Web interface. The Rational ClearQuest Web client uses a separate connection mechanism, so there can be separate security issues (see "Establishing the Rational ClearQuest Web interface" on page 203).

Making policy choices

Examine the following policy choices that control how the developers work in the Rational ClearCase environment:

Allowing multiple associations

In some environments, you want to capture the fact that one fix in the software resolved multiple problems that are recorded against the component. In the configuration file, set the CQCC_MULTIPLE_ASSOCS configuration parameter to TRUE to allow more than one change request record to be associated with an element that is being changed. For example:

```
&SetConfigParm("CQCC_MULTIPLE_ASSOCS", "TRUE");
```

If CQCC_MULTIPLE_ASSOCS is FALSE, then only one association is allowed for each version.

Controlling query usage

Set the CQCC_QUERY_ENABLE and CQCC_QUERY_FILTER configuration parameters to control query usage. For more information about queries in the integration, see "Query support" on page 192.

CQCC_QUERY_ENABLE

If developers use the Rational ClearQuest client, use the CQCC_QUERY_ENABLE configuration parameter to determine which Rational ClearQuest queries are available to developers to search for associations. The following values control query usage:

BOTH Both CQ and LOCAL. Queries defined both in the Rational ClearQuest user database and locally in the configuration file are available.

CQ Only named queries defined in the Rational ClearQuest user database schema and visible in **Personal Queries** and **Public Queries** folders in the workspace are available.

Restriction: To be used with the integration, a query must have the first column defined as the ID field that is generated by Rational ClearQuest. If the ID field is not first, the integration cannot identify and work with the user's selections that are returned from the query. See "SetResultSet" on page 206.

LOCAL
Only queries defined locally in the configuration file are available. By default, only local queries are shown.

OFF The **Browse** buttons and **Queryname** menu option are not displayed in the user interface.

For example:

```
&SetConfigParm("CQCC_QUERY_ENABLE", "BOTH");
```

Product tip: The full range of query capabilities is available only with the Rational ClearQuest Client through the Rational ClearQuest Perl API. The Rational ClearQuest Web interface displays only local queries due to limitations in the Rational ClearQuest Web API.

To open up query capabilities for users of the Rational ClearQuest Client, modify the CQCC_QUERY_ENABLE configuration parameter setting.

CQCC_QUERY_FILTER

The CQCC_QUERY_FILTER configuration parameter refines availability of Rational ClearQuest queries. With the CQCC_QUERY_FILTER setting, you can control which queries from the Rational ClearQuest user database workspace are permissible. Provide a Perl regular expression that the query names must match. For example:

```
&SetConfigParm("CQCC_QUERY_FILTER", "Public Queries");
```

If the filter is set to ″Public Queries,″ only queries with that string in their path are shown in the list from which the developer selects.

Allowing use of the graphic user interface (GUI)

With the CQCC_GUI_ENABLE configuration parameter, the project manager or developers can choose to use the integration with a Perl/TK GUI. CQCC_GUI_ENABLE is used to enable or disable the Perl/TK GUI. The following values are used:

'ALWAYS'
The GUI is presented for command-line access as well, when possible.

'OFF' The GUI is never presented and Perl/TK is not loaded.

'ON' If Perl/TK support is available and ClearCase Explorer or the File Browser is in use, then the GUI is presented.

For example:

```
&SetConfigParm("CQCC_GUI_ENABLE", "ON");
```

Forcing checkin success before committing associations

The CQCC_POSTCHECKIN_COMMIT configuration parameter enables the integration on checkin to delay committing associations in the Rational ClearQuest user database until after the Rational ClearCase checkin operation completes. This delay avoids problems caused if the checkin fails. Because the integration relies on a preoperation checkin trigger, it makes database changes to both the Rational ClearCase VOB and Rational ClearQuest user database before the checkin succeeds. If the checkin fails and the developer cancels the checkout, the Rational ClearQuest user database retains references to the checkin that never completed. Also, checking in identical files without the **-identical** option does not succeed. If the failed files are later checked out, the associations committed with the previous unsuccessful operation are incorrect. Using this configuration parameter avoids this problem. For example:

```
&SetConfigParm("CQCC_POSTCHECKIN_COMMIT", "TRUE");
```

By default, only the preoperation trigger fires on checkin. With this option enabled (set to TRUE), the preoperation trigger is used to decide associations and a second postoperation trigger fires to make the actual database changes. Using this option can also require an extra Rational ClearQuest login on the postoperation trigger. The extra operations take more time because of the additional Rational ClearQuest login but keeps a more accurate database.

Important: You can use the CQCC_ASSOC_BATCH_ENABLE environment variable to help minimize the cost of the extra operations. However, batch operations can be delayed under certain conditions. For information about those conditions, see "Using the association batch feature."

Enhancing performance

The performance of the integration triggers can vary depending on how they access Rational ClearCase and Rational ClearQuest configurations. The CQCC_TIMER configuration parameter can record diagnostic timing information about the trigger session (see "Producing timing information" on page 216).

Using the association batch feature

The configuration parameter CQCC_ASSOC_BATCH_ENABLE in the configuration file reduces the number of Rational ClearQuest logins and queries performed for batches of files. The following values control the batch feature:

AUTOFLUSH
Causes batches to be processed for each file in a defined series instead of waiting for the end of the series.

FALSE
Disables the association batch feature. Performs each transaction by establishing a new connection to the Rational ClearQuest user database, completing processing the single transaction, and closing the connection.

TRUE Enables the association batch feature and reduces the amount of overhead processing. Performs one Rational ClearQuest login, establishes a new connection to the Rational ClearQuest user database, processes multiple transactions in a series, and closes the connection at the end of the series.

Use the value TRUE to enable the reduction. For example:

```
&SetConfigParm("CQCC_ASSOC_BATCH_ENABLE", "TRUE");
```

This reduction is done by writing Rational ClearQuest transactions to a log file (.cqcc_assoc_batch) that is stored in the developer home directory (on Linux and the UNIX system) or in the application data directory (on the Windows system). When the batch completes, the transaction log file is read back and all the necessary Rational ClearQuest changes are made at the same time. The association batch feature improves performance in many cases.

The log is automatically processed at the end of each single operation or at the end of a batch operation. A batch is defined either as a set of files used in one Rational ClearCase operation, for example, a multiple-checkout, or as defined by the user using the CQCC_ASSOC_BATCH_SERIES environment variable (see "Defining a batch" on page 213).

If posting the log file fails, resolve the problem (for example, a login failure) and try again by either doing another Rational ClearCase operation or forcing the log to be processed (see "Handling an incomplete posting"). The log file can be rerun without causing duplications and is automatically renamed and moved aside when the posting process succeeds.

If you use the CQCC_POSTCHECKIN_COMMIT configuration parameter (see "Forcing checkin success before committing associations" on page 211), then any failed checkins are not written to the log for processing.

If you currently have the association batch feature enabled, you can gradually stop using batch processing by specifying AUTOFLUSH. For example:

```
&SetConfigParm("CQCC_ASSOC_BATCH_ENABLE", "AUTOFLUSH");
```

Batches are processed for each file in a defined series instead of waiting for the end of the series. Although each file is processed individually to completion, the pending transaction is saved locally to a log file until the transaction successfully completes. If the transaction fails, it is retried with the next Rational ClearQuest operation based on the locally stored information. The performance benefit of batch processing is lost, but reliability is maintained. If you instead disable batch processing suddenly (specify FALSE for CQCC_ASSOC_BATCH_ENABLE) after using it as a standard practice, you could leave unposted transactions in a locally stored batch log.

Handling an incomplete posting

The association batch feature depends on the integration being called at the end of a batch to process the log, usually after the last checkout or checkin completes. If

this last event is missed, for example, the last checkin failed because the file was identical, the log is not processed and the Rational ClearQuest operations are not posted. The data is not lost and is posted on the next successful Rational ClearCase operation that can begin processing the log. But the log may stay in the stored location for some time before the processing starts. You or the developer can force the log to be processed by using the following command from a command shell in the original VOB context:

```
cqcc_launch -vob -op batch
```

The batch operation processes any transactions that are in the log.

Defining a batch

If you use the CQCC_ASSOC_BATCH_ENABLE configuration parameter in the configuration file (see "Using the association batch feature" on page 211), the developer can also use the environment variable CQCC_ASSOC_BATCH_SERIES to define a batch. A batch is ordinarily the set of files to which a single Rational ClearCase operation is applied. The CQCC_ASSOC_BATCH_SERIES environment variable provides a way for a developer to broaden the meaning of a batch to be a series of Rational ClearCase operations. This is useful for checking in multiple files more efficiently than checking them in one at a time.

If the CQCC_ASSOC_BATCH_SERIES environment variable is set to TRUE, the integration assumes that a batch is in effect and operations are logged. When the environment variable is set to FALSE, the next Rational ClearCase operation automatically causes the logged operations to be run. Use this option carefully because Rational ClearQuest changes are deferred if a series is in process. The developer can force the batch log to be flushed. For example, a script that checks in a series of files might look like this example (in the Windows system).

```
set CQCC_ASSOC_BATCH_SERIES=TRUE
cleartool ci -nc file1
cleartool ci -nc file2
set CQCC_ASSOC_BATCH_SERIES=FALSE
cqcc_launch -vob -op batch
```

The last command forces the batch log to be flushed.

Requesting confirmation of batch completion

You can provide a visual cue that the batch operation has completed successfully by setting the CQCC_ASSOC_BATCH_CONFIRM configuration parameter. The following values are supported:

ALWAYS
Displays an information window when the batch processing succeeds for a single file or multiple files in a batch.

MULTIPLE
Displays an information window when the batch processing succeeds if more than a single file is being processed in a batch.

OFF Nothing is reported.

Set the value to MULTIPLE to avoid seeing the window for every operation. For example:

```
&SetConfigParm("CQCC_ASSOC_BATCH_CONFIRM", "MULTIPLE");
```

Tuning automatic association features

If you use CQCC_AUTO_ASSOCIATE or CQCC_ASSOC_BATCH_SERIES to automatically determine associations (see "Controlling and using automatic

associations"), use the CQCC_RESTRICTIONS_TIMEOUT configuration parameter to limit extra processing during the batch. The timeout value controls the number of seconds that a restrictions check can be reused during CQCC_ASSOC_BATCH_SERIES log processing.

The integration normally rechecks the new associations each time interval by logging into the Rational ClearQuest user database and performing a restrictions query. The integration caches the most recent restrictions query results and reuses them for a maximum time before it rechecks them again. After the time is exceeded, a new Rational ClearQuest login and restrictions query are performed to ensure that the selected associations are still valid. To delay this login operation and query during batch processing, set the CQCC_RESTRICTIONS_TIMEOUT configuration parameter in the configuration file or the environment variable to an increased number of seconds. To use a timeout of 10 minutes, set the value to 600. For example:

```
&SetConfigParm("CQCC_RESTRICTIONS_TIMEOUT, 600);
```

CQCC_RESTRICTIONS_TIMEOUT is set to 300 seconds (5 minutes) by default. The minimum value is 0 (disabled) and the maximum is 1200 (20 minutes).

Controlling and using automatic associations

Use the CQCC_AUTO_ASSOCIATE_ENABLE configuration parameter to control whether developers can use the CQCC_AUTO_ASSOCIATE and CQCC_ASSOC_BATCH_SERIES environment variables.

Enabling and disabling automatic associations

In the configuration file, use the CQCC_AUTO_ASSOCIATE_ENABLE configuration parameter to specify whether developers can use the CQCC_AUTO_ASSOCIATE environment variable (see "Using automatic associations") or CQCC_ASSOC_BATCH_SERIES environment variable (see "Defining a batch" on page 213).

- To prohibit use of CQCC_AUTO_ASSOCIATE or CQCC_ASSOC_BATCH_SERIES, set CQCC_AUTO_ASSOCIATE_ENABLE to "FALSE".
- To allow use of CQCC_AUTO_ASSOCIATE or CQCC_ASSOC_BATCH_SERIES, set CQCC_AUTO_ASSOCIATE_ENABLE to "TRUE". For example:

  ```
  &SetConfigParm("CQCC_AUTO_ASSOCIATE_ENABLE", "TRUE");
  ```

Using automatic associations

If the CQCC_AUTO_ASSOCIATE_ENABLE configuration parameter is enabled in the configuration file (see "Enabling and disabling automatic associations"), the developers can specify one or more change requests to associate with a batch of files on checkout and checkin operations without the user interface being displayed. The integration uses the change requests that the developers specify rather than prompting them through the user interface. (The ApplyToAll mechanism in the integration user interface works only for files being versioned in a single checkin or checkout command.) With this option, the developers can handle large batches of files in multiple checkin and checkout commands.

The developers can set the CQCC_AUTO_ASSOCIATE environment variable to the associations that they want to make and then start either **cleartool**, File Browser, or ClearCase Explorer. The following values are recognized:

- To specify change requests, the developer uses the same conventions that are used in the **Type In** option in the user interface. For example:

```
setenv CQCC_AUTO_ASSOCIATE "SAMPL1,2,3"
```

The value ″SAMPL1,2,3″ identifies change request IDs 1, 2, and 3 in the SAMPL user database.

- To disable the option, the developer uses ″″.
- To select no change requests, the developer specifies ″-″.

The values that they specify remain in effect until they disable the option. For example:

```
setenv CQCC_AUTO_ASSOCIATE ""
```

On checkin operations, the integration uses the values that they specify with this option to override associations that they made on checkouts. If an error occurs, the integration displays the related messages and starts the user interface for associating change requests.

Specifying associations in comment patterns

If the CQCC_COMMENT_PATTERN configuration parameter is enabled in the configuration file, the developers can provide that pattern to look for in checkout or checkin comments. The pattern provides a list of associations to use and replaces the prompting for associations by the integration interface. The pattern is disabled by default and enabled by setting a nonempty pattern.

To enable this option, in the CQCC_COMMENT_PATTERN configuration parameter, specify a pattern in the form of a Perl expression. For example:

```
&SetConfigParm("CQCC_COMMENT_PATTERN", "BUGS:\\[(\\S+)\\]");
```

This example uses double backslash characters to escape each backslash character.

The developers make associations by entering a checkin or checkout comment that matches the pattern, as in the following example of a checkout comment:

```
"This fixes BUGS:[SAMPL1,2,3]"
```

The pattern that is supplied in the checkout comment matches the pattern that is defined in the CQCC_COMMENT_PATTERN configuration parameter. This method avoids starting the integration user interface for associating files with change requests.

The change requests that developers enter on checkin commands override the change requests that they specify on checkout commands.

If the CQCC_AUTO_ASSOCIATE configuration parameter and the CQCC_COMMENT_PATTERN configuration parameter are enabled at the same time, the integration uses CQCC_AUTO_ASSOCIATE when it makes associations.

Debugging and analyzing operations

Environment variables are useful in troubleshooting and working with IBM Customer Support.

Generating operational information

To produce debugging information, set the CQCC_DEBUG environment variable with one of the following formats:

On Linux and the UNIX system: `setenv CQCC_DEBUG` *`value`*

On the Windows system: `set CQCC_DEBUG=value`

For *value*, use one of the following numbers:

0 No debugging information

1 Basic debugging information

2 Verbose debugging information that includes Rational ClearQuest traffic tracing

Producing timing information

To produce timing information, define the CQCC_TIMER configuration parameter or environment variable. For example:

```
&SetConfigParm("CQCC_TIMER", "1");
```

This provides large-grain timing information for basic internal operations such as Rational ClearQuest login, queries, and associations; and Rational ClearCase information gathering and hyperlink maintenance. Set the value to zero (0) to disable timing information.

The integration writes the information to standard output and can store it in a file (see "Controlling logged output").

Controlling logged output

Set the CQCC_LOG_OUTPUT configuration parameter to control the recording of all warning, error, and fatal messages that are written to a log file for convenience during problem diagnosis. Use the following values:

APPEND
Adds to the log file (use only during debugging).

OFF Disables log file output.

OVERWRITE
Overwrites the log file for each trigger session.

For example:

```
&SetConfigParm("CQCC_LOG_OUTPUT", "OVERWRITE");
```

The log file name is cqcc_output.log. On Linux and the UNIX system, the file is written to the user's home directory. On the Windows system, the file is written to the profile directory.

Testing the integration

After you install the triggers on one or more VOBs and edit the configuration file, you can test the connection between Rational ClearCase and Rational ClearQuest configurations by entering the following command:

On Linux and the UNIX system:

```
cqcc_launch CQCC/config.pl -test
```

On the Windows system:

```
cqcc_launch CQCC\config.pl -test
```

Tip: The preceding commands use the default path for the configuration file. If you specified a path to a central location when you configured the

integration, use that path when invoking the **cqcc_launch** command (see "Using a shared configuration file and triggers" on page 198).

The command displays output indicating whether it is able to connect to the target Rational ClearQuest user database. For more detailed output messages, set the CQCC_DEBUG environment variable to 2 (see "Generating operational information" on page 215).

Customizing the integration

To understand how the overall trigger works or how to customize its behavior, review the TriggerCQCC.pm class and its methods. Most changes in the visible behavior of the trigger require changes in this class.

Tip: For internal documentation, contact IBM Customer Support.

If you make local changes, use the following procedure:

1. Make a copy of the MyTrigger.pm template class.
2. Rename the copy.
3. Make your changes in the renamed file by overriding or extending the TriggerCQCC methods (see MyTrigger.pm for more details).

Tip: Do not make changes directly to the TriggerCQCC source code.

Following this procedure facilitates upgrading to later releases of TriggerCQCC.pm from Rational ClearCase and provides a fallback to the released trigger for working with IBM Customer Support. For information about support, see "Policy regarding customization and support" on page 194.

About the Integration Query wizard

After developers establish associations between Rational ClearCase versions and Rational ClearQuest change requests, you can use the Rational ClearQuest Integration Query wizard on Windows systems to identify the change requests that are associated with a project over a period of time. For example, you might use the wizard to answer the question, "Which change requests were associated with Release 3.1 of Project X?"

To start the Integration Query wizard

Do one of the following:

- Click **Start > Programs > IBM Rational > IBM Rational ClearCase > Administration > ClearQuest Integration Query**.
- Enter **cqquery** at the command prompt.

Click **Help** for instructions on completing each page of the wizard.

Chapter 14. Integrating changes

This chapter describes merging versions of text-file elements or directories. A merge calls an element-type-specific program (the merge method) to merge the contents of two or more files, or two or more directories.

About integrating changes

In a parallel development environment, the opposite of branching is merging. In the simplest scenario, merging incorporates changes on a subbranch into the **main** branch. However, you can merge work from any branch to any other branch. You need to be familiar with techniques and scenarios for merging versions of elements and branches. Automated merge facilities are included in Rational ClearCase to handle almost any scenario.

How merging works

A merge combines the contents of two or more files or directories into a single new file or directory. The Rational ClearCase merge algorithm uses the following files during a merge (see Figure 53):

- Contributors, which are typically one version from each branch you are merging. (You can merge up to 15 contributors.) You specify which versions are contributors.
- The base contributor, which is typically the closest common ancestor of the contributors. (For selective merges, subtractive merges, and merges in an environment with complex branch structures, the base contributor may not be the closest common ancestor.) The Rational ClearCase merge algorithm determines which contributor is the base contributor.
- The target contributor, which is typically the latest version on the branch that will contain the results of the merge. You determine which contributor is the target contributor.
- The merge output file, which contains the results of the merge and is usually checked in as a successor to the target contributor. By default, the merge output file is the checked-out version of the target contributor, but you can choose a different file to contain the merge output.

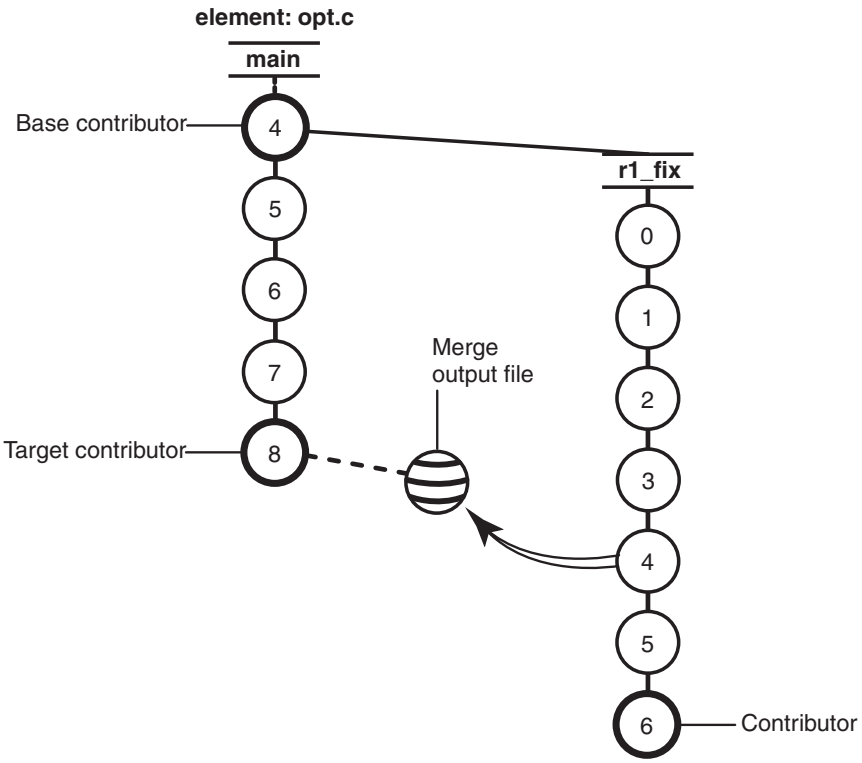


Figure 53. Versions involved in a typical merge

Merging files and directories involves the following actions:

1. The base contributor is identified.
2. Each contributor is compared against the base contributor. (See Figure 54.)
3. Any line that is unchanged between the base contributor and any other contributor is copied to the merge output file.
4. Any line that has changed between the base contributor and one other contributor is accepted in the contributor.

   Depending on how you started the merge operation, the change may be copied to the merge output file. However, you can disable the automated merge capability for any given merge operation. If you disable this capability, you must approve each change to the merge output file.
5. For any line that has changed between the base contributor and more than one other contributor, you are required to resolve the conflicting difference.

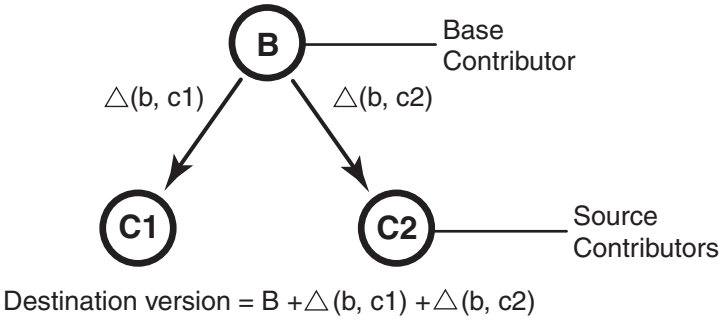


Figure 54. Rational ClearCase merge algorithm

To merge versions, you can use the graphic user interface (GUI) tools (see "Using the GUI to merge elements" on page 221) or the command-line interface (see "Using the command line to merge elements" on page 222).

Using the GUI to merge elements

Three graphical tools are provided to help you merge elements:

- Merge Manager
- Diff Merge
- Version Tree Browser

About the Merge Manager

The Merge Manager manages the process of merging one or more Rational ClearCase elements. It automates the processes of gathering information for a merge, starting a merge, and tracking a merge. It can also save and retrieve the state of a merge for a set of elements.

You can use the Merge Manager to merge from many directions:

- From a branch to the **main** branch
- From the **main** branch to another branch
- From one branch to another branch

To start the Merge Manager

On the UNIX system, type **clearmrgman** at a command prompt.

On the Windows system, do one of the following:

- Click **Start > Programs > IBM Rational > IBM Rational ClearCase > Merge Manager**.
- In ClearCase Explorer, click **Base ClearCase**, and then click **Merge Manager**.

About Diff Merge

The Diff Merge utility shows the differences between two or more versions of file or directory elements. Use this tool to compare up to 16 versions at a time, navigate through versions, merge versions, and resolve differences between versions.

To start Diff Merge

On the UNIX system, do one of the following at a command prompt:

- Type **xcleardiff** *path_1 path_2.*
- Use the **cleartool merge –graphical** command.

On the Windows system, do one of the following:

- In Rational ClearCase Explorer, right-click an element and click **Compare with Previous Version**.
- In Windows Explorer, right-click an element and click **ClearCase > Compare with Previous Version**.
- In the Merge Manager, click **Compare**.

About the Version Tree Browser

The Version Tree Browser displays the version tree for an element. The version tree is useful when merging to do the following tasks:

- Locate versions or branches that have contributed to or resulted from a merge
- Start a merge by clicking on the appropriate symbol

The merge can be recorded with a merge arrow, which is implemented as a hyperlink of type **Merge**.

To start the Version Tree Browser

On the UNIX system, do one of the following:

- At a command prompt, use the **cleartool lsvtree –graphical** command.
- In File Browser, click an element and click **Versions > Show version tree**

On the Windows system, do one of the following:

- In Rational ClearCase Explorer, click **Tools > Version Tree**.
- Click **Start > Programs > IBM Rational > IBM Rational ClearCase > Version Tree Browser**
- In Windows Explorer, right-click a versioned element and click **ClearCase > Version Tree**.

Using the command line to merge elements

Use the following commands to perform merges from the command line:

- **cleartool merge**
- **cleartool findmerge**
- **cleardiff**

For more information on these commands, see the *IBM Rational ClearCase Command Reference*.

Common merge scenarios

The following sections present a series of merge scenarios that require work on one branch of an element to be incorporated into another branch.

Each scenario shows the version tree of an element that requires a merge and indicates the appropriate command to perform the merge.

Selective merge from a subbranch

In a selective merge from a subbranch, you want to incorporate the changes in version `/main/r1_fix/4` into new development. To perform the merge, you specify which versions on the `r1_fix` branch to include. See Figure 55.

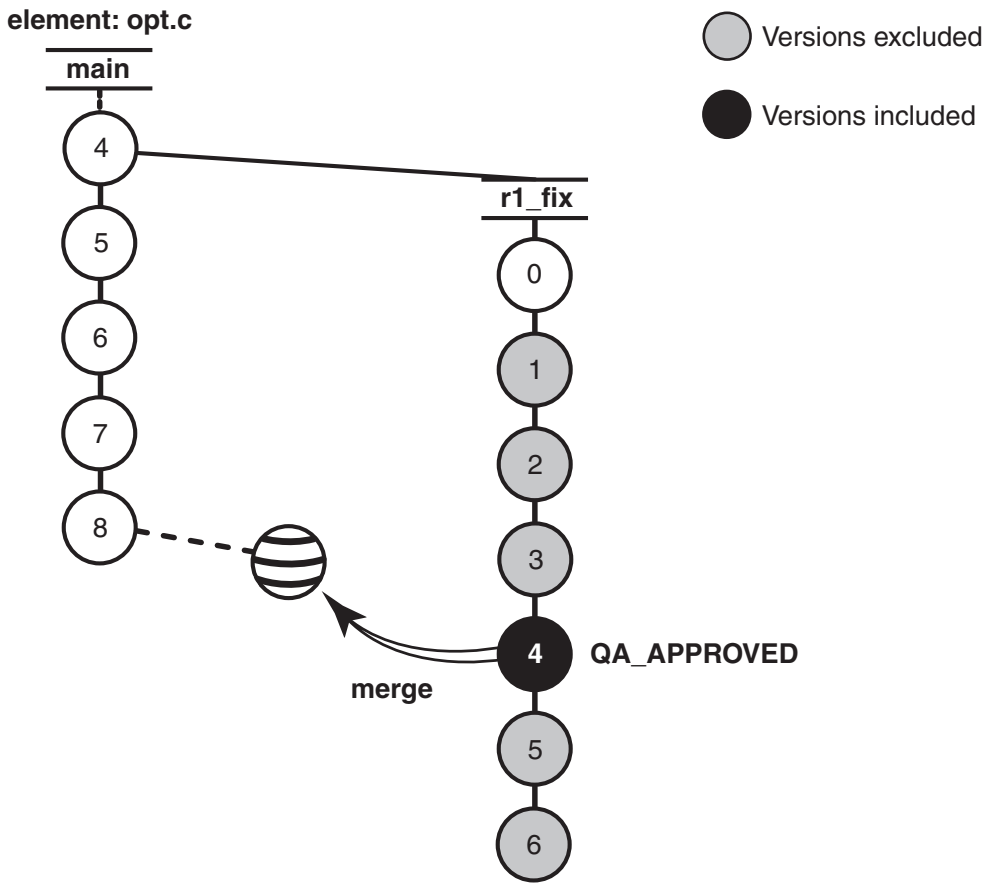


Figure 55. Selective merge from a subbranch

In a view configured with the default config spec, enter the following commands to perform the selective merge:

```
cleartool checkout opt.c
cleartool merge –to opt.c –insert –version /main/r1_fix/4
```

You can also specify a range of consecutive versions to be merged. For example, this command merges only the changes in versions `/main/r1_fix/2` through `/main/r1_fix/4`:

```
cleartool merge –to opt.c –insert –version /main/r1_fix/2 /main/r1_fix/4
```

No merge arrow is created for a selective merge.

Removing the contributions of some versions

In a subtractive merge, you remove contributions of some versions in the merge. For example, a new feature, implemented in versions 14 through 16 on the **main** branch, are not be included in the product. You must remove the changes made in those versions. See Figure 56.

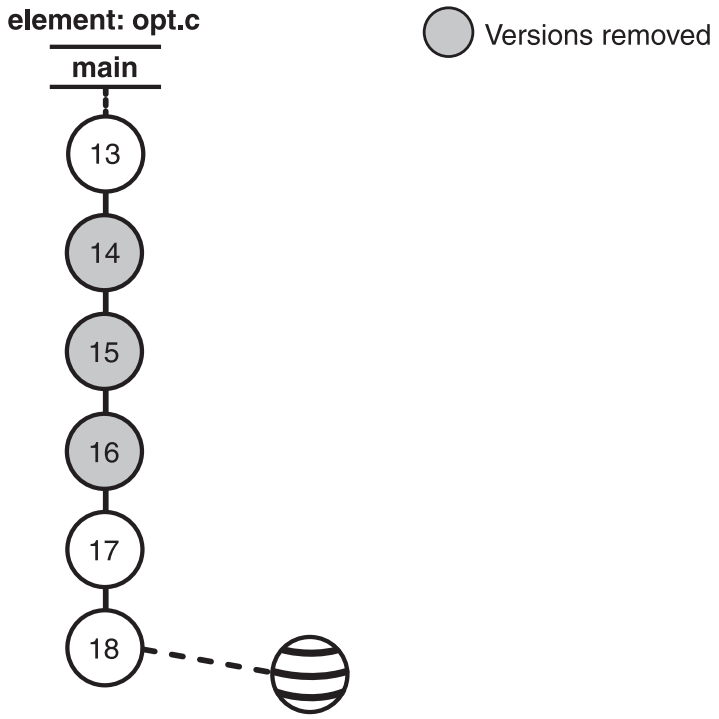


Figure 56. Removing the contributions of some versions

Enter the following commands to perform this subtractive merge:

```
cleartool checkout opt.c
cleartool merge –to opt.c –delete –version /main/14 /main/16
```

No merge arrow is created for a subtractive merge.

Merging all project work

Your team has been working on a branch. Now, your job is to merge all the changes into the **main** branch.

The **findmerge** command can handle most common cases easily. For isolating the project work, the command can accommodate the schemes described in "All project work isolated on a branch" and "All project work isolated in a view" on page 225.

All project work isolated on a branch

The standard approach to parallel development isolates all project work on the same branch. More precisely, all new versions of source files are created on like-named branches of their respective elements (that is, on branches that are instances of the same *branch type*). This makes it possible for a single **findmerge** command to locate and incorporate all the changes. Suppose the common branch is named **gopher**. You can enter these commands in a view configured with the default config spec:

```
cd root-of-source-tree
cleartool findmerge . –fversion .../gopher/LATEST –merge –graphical
```

The **–merge –graphical** syntax causes the merge to take place automatically whenever possible, and to start the graphical merge utility if an element merge

requires user interaction. If the project has made changes in several VOBs, you can perform all the merges at once by specifying several paths, or by using the **–avobs** option to **findmerge**.

All project work isolated in a view

Some projects are organized so that all changes are made in a single view (typically, a shared view). For such projects, use the **–ftag** option to **findmerge**. Suppose the project work has been done in a view whose view tag is **goph_vu**. These commands perform the merge:

```
cd root-of-source-tree
cleartool findmerge . –ftag goph_vu –merge –graphical
```

Tip: Working in a single shared view is not recommended because doing so can degrade system performance.

Merging a new release of an entire source tree

Your team has been using an externally supplied source-code product, maintaining the sources in a VOB. The successive versions supplied by the vendor are checked in to the **main** branch and labeled **VEND_R1**, **VEND_R2**, and **VEND_R3**. Your team's fixes and enhancements are created on subbranch **enhance**. The views in which your team works have the following configuration to branch from the **VEND_R3** baseline:

```
element * CHECKEDOUT
element * .../enhance/LATEST
element * VEND_R3 -mkbranch enhance
element * /main/LATEST -mkbranch enhance
```

The version trees in Figure 57 show the following various likely cases:

- An element that your team started changing at Release 1 (**enhance** branch created at the version labeled **VEND_R1**)
- An element that your team started changing at Release 3
- An element that your team has never changed

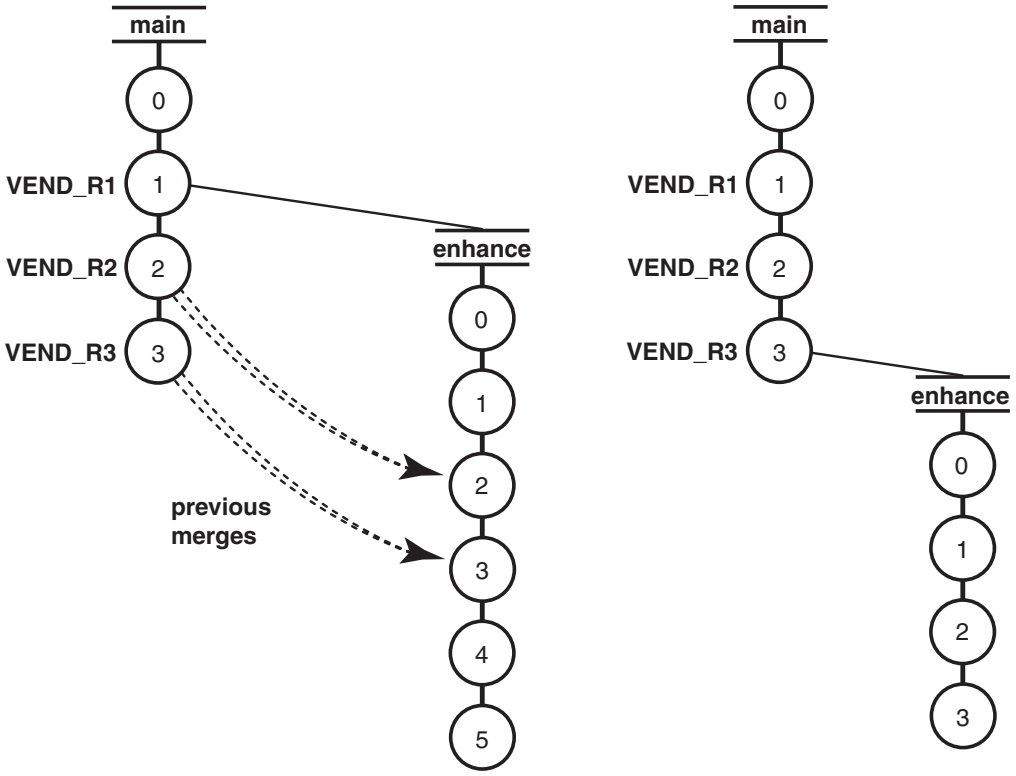


Figure 57. Merging a new release of an entire source tree

When Release 4 arrives, and you need to integrate this release with your team's changes.

To prepare for the merge, add the new release to the **main** branch and label the versions **VEND_R4**. Merging the source trees involves merging from the version labeled **VEND_R4** to the most recent version on the **enhance** branch; if an element has no **enhance** branch, nothing is merged.

This procedure accomplishes the following integration:

1. Load the vendor's Release 4 media into a standard directory tree:

   ```
   cd /usr/tmp
   tar –xv
   ```

   The directory tree created is **mathlib_4.0**.

2. As the VOB owner, run **clearfsimport** in a view configured with the default config spec to create Release 4 versions on the **main** branches of elements (and create new elements as needed).

   ```
   clearfsimport –recurse /usr/tmp/mathlib_4.0 /vobs/proj/mathlib
   ```

3. Label the new versions:

   ```
   % cleartool mklbtype –c "Release 4 of MathLib sources" VEND_R4
   Created label type "VEND_R4".
   % cleartool mklabel –recurse VEND_R4 /vobs/proj/mathlib
    . (lots of output)
    .
   ```

4. Set to a view that is configured with your team's config spec and selects the versions on the **enhance** branch:

   ```
   cleartool setview enh_vu
   ```

5. Merge from the **VEND_R4** configuration to your view:

```
cleartool findmerge -nback /vobs/proj/mathlib –fver VEND_R4 –merge \
–graphical
```

The **–merge –graphical** syntax instructs **findmerge** to merge automatically if possible, but if not, start the graphical merge tool.

6. Verify the merges, and check in the modified elements.

You have now established Release 4 as the new baseline. Developers on your team can update their view configurations.

```
element * CHECKEDOUT

element * .../enhance/LATEST
element * VEND_R4 –mkbranch enhance          (change from VEND_R3 to VEND_R4)

element * /main/LATEST –mkbranch enhance
```

Elements that have been active continue to evolve on their **enhance** branches. When elements are revised for the first time, their **enhance** branches are created at the **VEND_R4** version.

Merging directory versions

A feature of Rational ClearCase is versioning of directories. Each version of a directory element catalogs a set of file elements and directory elements (and VOB symbolic links on the UNIX system). In a development project, directories change as often as files do. Merging the changes to another branch is as easy merging files.

Take a closer look at the source tree scenario that is described in "Merging a new release of an entire source tree" on page 225. Suppose you find that the vendor has made the following changes in directory **/vobs/proj/mathlib/src**:

- File elements **Makefile**, **getcwd.c**, and **fork3.c** are revised.
- File elements **readln.c** and **get.c** are deleted.
- A new file element, **newpaths.c**, is created.

When you use **findmerge** to merge the changes made in the **VEND_R4** sources to the **enhance** branch, the changes to both the files and the directory are handled automatically. The following **findmerge** excerpt shows the directory merge activity:

```
*******************************
<<< directory 1: /vobs/proj/mathlib/src@@/main/3
>>> directory 2: .@@/main/enhance/1
>>> directory 3: .
*******************************
-------[ removed directory 1 ]-------|----------[ directory 2]------------
get.c  19-Dec-1991 drp                |-
*** Automatic: Applying REMOVE from directory 2
-----------[ directory 1 ]-----------|--------[ added directory 2]---------
                                    -| newpaths.c  08-Mar.21:49 drp
*** Automatic: Applying ADDITION from directory 2
-------[ removed directory 1 ]-------|-----------[ directory 2]------------
readln.c  19-Dec-1991 drp              |-
*** Automatic: Applying REMOVE from directory 2
Recorded merge of ".".
```

If you have changes to merge from both files and directories, it may be a good idea to run **findmerge** twice: first to merge directories, and again to merge files. Using the **–print** option to a **findmerge** command does not report everything that is merged, because **findmerge** does not see new files or subdirectories in the merge-from version of a directory until after the directories are merged. To report

every merge that takes place, use **findmerge** to merge the directories only, and then use **findmerge –print** to get information about the file merges that are needed. Afterward, you can cancel the directory merges by using the **uncheckout** command on the directories.

Using other merge tools

You can create a merged version of an element manually or with any available analysis and editing tools. Check out the target version, revise it, and check it in. Immediately before (or after) the checkin, record your activity by using the **merge** command with the **–ndata** (no data) option:

```
% cleartool checkout nextwhat.c
Checkout comments for "nextwhat.c":
merge enhance branch
.
Checked out "nextwhat.c" from version "/main/1".

% <invoke your own tools to merge data into checked-out version>

% cleartool merge –to nextwhat.c –ndata –version .../enhance/LATEST
Recorded merge of "nextwhat.c".
```

This form of the **merge** command does not change any file system data; it merely attaches a merge arrow (a hyperlink of type **Merge**) to the specified versions. After you make this annotation, your merge is indistinguishable from one performed with Rational ClearCase tools.

Chapter 15. Using element types to customize file element processing

This chapter describes element types, their creation, and their usage.

About element types and file processing

Most projects involve many different file types. For example, in a typical software release, developers may work on language-specific source files, language-specific header files, document files (in text and binary format), and library files.

Every file that is stored in a VOB is associated with an element type. Predefined element types for various kinds of file types are provided, and every element type has an associated type manager, which handles the operations performed on versions of the element.

For some file types in your project, you may want to create your own element types so that you can customize the handling of the files. You can also create your own type managers.

You need to understand how element types are used, how type managers classify and manage files, and how you can customize file classification and management.

File types in a typical project

Table 7 lists the file types used in a typical development project.

Table 7. Files used in a typical project

Type of file	**Identifying characteristic**
Source files	
C-language source file	.c file name extension
C-language header file	.h file name extension
FrameMaker binary file	.doc or .mif file name extension, first line of file begins with `<Maker`
The UNIX system: manual page source file	.1 to .9 file name extension
Derived files	
The UNIX system: **ar(1)** archive (library)	.a file name extension
The Windows system: library, shared library	.lib, .dll file name extension
Compiled executable	The UNIX system: *<varies with system architecture>* The Windows system: .exe file name extension

How element types are assigned

In various contexts, one or more *file types* may be determined for an existing file system object, or for a name to be used for a new object. When you create a new element and do not specify an element type, the file type for the element is determined by default.

The file-typing routines use predefined and user-defined *magic files*, as described in the **cc.magic** reference page. A magic file can use many different techniques to determine a file type, including file name pattern-matching, **stat(2)** data, and standard magic numbers on the UNIX system.

For example, the magic files in "Sample magic file on the UNIX system" and "Sample Magic File on the Windows system" specify several file types for each kind of file listed in Table 7.

Sample magic file on the UNIX system

```
(1)  c_src src_file text_file file:      -name "*.c" ;
(2)  hdr_file text_file file:            -name "*.h" ;
(3)  frm_doc binary_delta_file doc file: -magic 0, "<MakerFile" ;
(4)  manpage src_file text_file file:    -name "*.[1-9]" ;
(5)  archive derived_file file:          -magic 32, "archive" ;
(6)  sunexec derived_file file:          -magic 40, "SunBin" ;
```

Sample Magic File on the Windows system

```
(1)  c_src src_file text_file file:       -name "*.c";
(2)  hdr_file text_file file:             -name "*.h" ;
(3)  frm_doc binary_delta_file doc file:  -magic 0, "<MakerFile" ;
(4)  library derived_file file:           -name "*.lib";
(5)  program compressed_file:             -name "*.exe" ;
```

Element types and type managers

Different classes of files are handled differently because *element types* are used to categorize elements. Each file element in a VOB must have an element type. An element gets its type when it is created; you can change the type of an element subsequently, with the **chtype** command. (An element is an *instance* of its element type, in the same way that an attribute is an instance of an attribute type and a version label is an instance of a label type.)

Each element type has an associated *type manager*, a suite of programs that handle the storage and retrieval of versions from storage pools. (See the **type_manager** reference page for information on how type managers work.) Thus, the way in which the data of a file element is handled depends on its element type.

Tip: Each directory element also has an element type. But directory elements do not use type managers; the contents of a directory version are stored in the VOB database itself, not in storage pools.

When you create an element without specifying the element type, an element type is assigned as follows:

- One or more magic files are read to find the file types for the name of the element.
- The list of file types associated with the first rule in the magic file that matches the name is retrieved.

- This list is compared with the set of element types defined for the VOB that stores the element. The element is created by using the first element type in the list that matches an element type in the VOB.

For example, a new element named monet_adm.1 is assigned an element type as follows:

1. A developer creates an element:
   ```
   cleartool mkelem monet_adm.1
   ```
2. Because the developer did not specify an element type (**–eltype** option), **mkelem** uses one or more magic files to determine the file types of the specified name.

   Tip: A search path facility uses the environment variable MAGIC_PATH. See the **cc.magic** reference page for details.

   If the magic file shown in "Sample magic file on the UNIX system" on page 230 is the first (or only) one to be used, rule "(4)" on page 230 is the first to match the name monet_adm.1, yielding this list of file types:
   ```
   manpage src_file text_file file
   ```
3. This list is compared with the set of element types defined in the VOB for the new element. If **text_file** is the first file type that names an existing element type, monet_adm.1 is created as an element of type **text_file**.
4. Data storage and retrieval for versions of element monet_adm.1 are handled by the type manager associated with the **text_file** element type; its name is **text_file_delta**:
   ```
   % cleartool describe eltype:text_file
   element type "text_file"
   ...
     type manager: text_file_delta
     supertype: file
     meta-type of element: file element
   ```

File-typing mechanisms are defined on a per-user or per-site basis; element types are defined on a per-VOB basis. (To ensure that element types are consistent across VOBs, the Rational ClearCase administrator can use global types.) In this case, a new element, monet_adm.1, is created as a **text_file** element; in a VOB with a different set of element types, the same magic file may have created it as a **src_file** element.

Other applications of element types

Element types allow differential and customized handling of files beyond the selection of type managers. Some examples are presented in "Using element types to configure a view" and "Processing files by element type" on page 232.

Using element types to configure a view

Creating all C-language header files as elements of type **hdr_file** allows flexibility in configuring views. Suppose that one developer reorganizes the project header files, working on a branch named **header_reorg** to avoid disrupting the team's work. To compile with the new header files, another developer can use a view re-configured with one additional rule:

```
element * CHECKEDOUT
element -eltype hdr_file * /main/header_reorg/LATEST
element * /main/LATEST
```

Processing files by element type

Suppose that a coding-standards program named **check_var_names** executes on each C-language source file. If all such files have element type **c_src**, a single **cleartool** command runs the program:

On the UNIX system:

```
cleartool find –avobs –visible –element 'eltype(c_src)' \
   –exec 'check_var_names $CLEARCASE_PN'
```

On the Windows system:

```
cleartool find –avobs –visible –element 'eltype(c_src)' ^
   –exec 'check_var_names %CLEARCASE_PN%'
```

Predefined and user-defined element types

Some of the element types described in this chapter (for example, **text_file**) are predefined. Others (for example, **c_src** and **hdr_file**) are not predefined; the previous examples work only if user-defined element types with these names are created with the **mkeltype** command.

When a new VOB is created, it contains a full set of the predefined element types. Each element type is associated with one of the type managers provided with Rational ClearCase. The **mkeltype** reference page describes the predefined element types and their type managers.

When you create a new element type with **mkeltype**, you must specify an existing element type as its *supertype*. By default, the new element type uses the same type manager as its supertype; in this case, the only distinction between the new and old types is for the purposes described in "Other applications of element types" on page 231. For differential data handling, use the **–manager** option to create an element type that uses a different type manager from its supertype.

Predefined and user-defined type managers

Predefined type managers are provided in Rational ClearCase. The type managers are described in the **type_manager** reference page. Each type manager is implemented as a suite of programs in a subdirectory of *ccase–home–dir*/lib/mgrs; the name of the subdirectory is the name of the type manager.

The **mkeltype –manager** command creates an element type that uses an existing type manager. You can further customize Rational ClearCase by creating new type managers and creating new element types that use them. Architecturally, type managers are mutually independent, but new type managers can use symbolic links to inherit some of the functions of existing ones.

Creating a new type manager (the UNIX system)

On the UNIX system, you can create any number of new type managers for use throughout the local network. Use these guidelines:

- Choose a name for the new type manager. Ideally the name shows its relationship to the data format (for example, **bitmap_mgr**). Create a subdirectory of *ccase–home–dir*/lib/mgrs with this name.

  Tip: Names of user-defined type managers must not begin with underscore.
- Create symbolic links to make the new type manager inherit some of its methods (file-manipulation operations) from an existing type manager.

- Create your own program for the methods that you want to customize. See "Writing a type manager program (the UNIX system)."
- On each Rational ClearCase or Rational ClearCase LT client host in the network, either make a copy of the new type manager directory or create symbolic link to it. The standard storage, performance, and reliability trade-offs apply.
- If your organization uses Rational ClearCase MultiSite, at every site, either make a copy of the new type manager directory, or create a symbolic link to it.

  Tip: An element type belongs to a VOB, and thus is available on every host that mounts its VOB. But a type manager is host-specific; it is the *ccase–home–dir*/lib/mgrs/*manager-name* directory on some host.

See the **type_manager** reference page and the file *ccase–home–dir*/lib/mgrs/mgr_info.h for additional information on type managers.

Writing a type manager program (the UNIX system)

When **cleartool** invokes a type manager method, it passes to the manager, in ASCII format, all the arguments needed to perform the operation. For example, many methods accept a *new_container_name* argument, specifying the path of a data container to which data is to be written.

One or more of the parameters can be ignored. For example, the **create_version** method is passed **pred_container_name**, the path of the predecessor version data container. If the type manager implements incremental differences, this is required information. Otherwise, the predecessor data container is of no interest.

Arguments are often object identifiers (OIDs). You need not know anything about how OIDs are generated; consider each OID to be a unique name for an element, branch, or version. In general, only type managers that store multiple versions in the same data container need be concerned with OIDs.

For more information on argument processing, see files *ccase–home–dir*/lib/mgrs/mgr_info.h (for C-language programs) and *ccase–home–dir*/lib/mgrs/mgr_info.sh (for Bourne shell scripts).

Exit status of a method

A user-defined type manager method must return to **cleartool** an exit status that indicates how the command is to be completed. The symbolic constants in *ccase–home–dir*/lib/mgrs/mgr_info.sh specify all valid exit statuses. For example, an invocation of **create_version** may create a new data container, and return the exit status MGR_STORE_KEEP_JUST_NEW. If creation of the new data container fails, **create_version** returns the exit status MGR_STORE_KEEP_JUST_OLD.

Type manager for manual page source files

One kind of file supported is a manual page source file, which is coded in **nroff(1)** format (see Table 7). A type manager for this kind of file may have these characteristics:

- It stores all versions in compressed form in separate data containers, like the **z_whole_copy** type manager.
- It implements version-comparison (compare method) by running **diff** on formatted manual pages instead of the source versions.

The basic strategy is to use most of the **z_whole_copy** type manager methods. The compare method uses **nroff(1)** to format the versions before displaying their differences.

Creating the type manager directory

The name **mp_mgr** (manual page manager) is appropriate for this type manager. The first step is to create a subdirectory with this name in the *ccase–home–dir*/lib/mgrs directory. For example:

```
mkdir /usr/rational/lib/mgrs/mp_mgr
```

Inheriting methods from another type manager

Most of the **mp_mgr** methods are inherited through symbolic links from the **z_whole_copy** type manager. You can enter the following commands as the *root* user in a Bourne shell:

```
# MP=$CLEARCASEHOME/lib/mgrs/mp_mgr
# for FILE in create_element create_version construct_version \
   create_branch delete_branches_versions \
   merge xmerge xcompare get_cont_info
> do
> ln –s ../z_whole_copy/$FILE $MP/$FILE
> done
#
```

Any methods that the new type manager does not support can be omitted from this list. The lack of a symbolic link causes an `Unknown Manager Request` error.

The sections “The create_version method” and “The construct_version method” on page 235 describe two of these inherited methods, which can serve as models for user-defined methods. Both methods are implemented as scripts in the same file, *ccase–home–dir*/lib/mgrs/z_whole_copy/Zmgr.

The create_version method

The **create_version** method is invoked when a **checkin** command is entered. The **create_version** method of the **z_whole_copy** type manager does the following operations:

1. Compresses the data in the checked-out version
2. Stores the compressed data in a data container located in a source storage pool
3. Returns to the calling process an exit status that indicates what to do with the new data container

The file *ccase–home–dir*/lib/mgrs/mgr_info.h lists the arguments passed to the method from the calling program (usually **cleartool** or File Browser):

```
/*****************************************************************
******
 * create_version
 * Store the data for a new version.
 * Store the version's data in the supplied new container, combining it
 * with the predecessor's data if desired (e.g for incremental deltas).
 *
 * Command line:
 *  create_version create_time new_branch_oid new_ver_oid new_ver_num
 *                 new_container_pname pred_branch_oid pred_ver_oid
 *                 pred_ver_num pred_container_pname data_pname
```

The only arguments that require special attention are `new_container_pname` (fifth argument), which specifies the path of the new data container, and `data_pname` (tenth argument), which specifies the path of the checked-out file.

The file *ccase–home–dir*/lib/mgrs/mgr_info.sh lists the appropriate exit statuses and provides a symbolic name for the **create_version** method:

```
# Any unexpected value is treated as failure
MGR_FAILED=1

# Return Values for store operations
MGR_STORE_KEEP_NEITHER=101
MGR_STORE_KEEP_JUST_OLD=102
MGR_STORE_KEEP_JUST_NEW=103
MGR_STORE_KEEP_BOTH=104
 .
 .
MGR_OP_CREATE_VERSION="create_version"
```

The following example is the code that implements the **create_version** method:

```
(1)     shift 1
(2)     if [ -s $4 ] ; then
(3)         echo '$0: error: new file is not of length 0!'
(4)         exit $MGR_FAILED
(5)     fi
(6)     if $gzip < $9 > $4 ; ret=$? ; then : ; fi
(7)     if [ "$ret" = "2" -o "$ret" = "0" ] ; then
(8)         exit $MGR_STORE_KEEP_BOTH
(9)     else
(10)        exit $MGR_FAILED
(11)    fi
```

The Bourne shell allows only nine command-line arguments. The `shift 1` in Line 1 discards the first argument (*create_time*), which is unneeded. Thus, the path of the checked-out version (*data_pname*), originally the tenth argument, becomes $9.

In Line 6, the contents of *data_pname* are compressed, then appended to the new, empty data container: *new_container_pname*, originally the fifth argument, but shifted to become $4. (Lines 2 through 5 verify that the new data container is, indeed, empty.)

Finally, the exit status of the **gzip** command is checked, and the appropriate value is returned (Lines 7 through 11). The exit status of the **create_version** method indicates that both the old data container (which contains the predecessor version) and the new data container (which contains the new version) are to be kept.

The construct_version method

An element **construct_version** method is invoked when standard software on the UNIX system reads a particular version of the element (unless the contents are already cached in a *cleartext* storage pool). For example, the **construct_version** method of element monet_admin.1 is invoked by the **view_server** when a user enters these commands:

% cp monet_admin.1 /usr/tmp
(read version selected by view)

% cat monet_admin.1@@/main/4
(read a specified version)

The **construct_version** method is also invoked during a **checkout** command, which makes a view-private copy of the most recent version on a branch.

The **construct_version** method of the **z_whole_copy** type manager does the following operations:

1. Uncompresses the contents of the data container
2. Returns to the calling process an exit status that indicates what to do with the new data container

The file *ccase–home–dir*/lib/mgrs/mgr_info.h lists the arguments passed to the method.

```
/*********************************************************************
******
 * construct_version
 * Fetch the data for a version.
 * Extract the data for the requested version into the supplied path, or
 * return a value indicating that the source container can be used as the
 * cleartext data for the version.
 *
 * Command line:
 *  construct_version source_container_pname data_pname version_oid
```

The file *ccase–home–dir*/lib/mgrs/mgr_info.sh lists the appropriate exit statuses and provides a symbolic name for the **construct_version** method:

```
# Any unexpected value is treated as failure
MGR_FAILED=1

# Return Values for construct operations
MGR_CONSTRUCT_USE_SRC_CONTAINER=101
MGR_CONSTRUCT_USE_NEW_FILE=102
 .
 .
MGR_OP_CONSTRUCT_VERSION="construct_version"
```

This example code in “The construct_version method source code” implements the **construct_version** method.

The construct_version method source code:

```
(1)     if $gzip -d < $1 > $2 ; then
(2)         exit $MGR_CONSTRUCT_USE_NEW_FILE
(3)     else
(4)         exit $MGR_FAILED
(5)     fi
```

In Line 1, the contents of *source_container_pname* are uncompressed and stored in the cleartext container, *data_pname*. The remaining lines return the appropriate value to the calling process, depending on the success or failure of the **gzip** command.

Implementing a new compare method

The **compare** method is invoked by a **cleartool diff** command. This method does the following operations:

1. Formats each version using **nroff(1)**, producing an ASCII text file
2. Compares the formatted versions, using **cleardiff** or **xcleardiff**

The file *ccase–home–dir*/lib/mgrs/mgr_info.h lists the arguments passed to the method from **cleartool** or File Browser.

```
/*********************************************************************
 * compare
 * Compare the data for two or more versions.
 * For more information, see man page for cleartool diff.
 *
```

```
 * Command line:
 *  compare [-tiny | -window] [-serial | -diff | -parallel] [-columns n]
 *          [pass-through-options] pname pname ...
******/
```

This listing shows that a user-supplied implementation of the **compare** method must accept all the command-line options that the Rational ClearCase **diff** command supports. The strategy here is to pass the options to **cleardiff** and not attempt to interpret them. After all options are processed, the remaining arguments specify the files to be compared.

The file *ccase–home–dir*/lib/mgrs/mgr_info.sh lists the appropriate exit statuses and provides a symbolic name for the **compare** method.

```
# Return Values for COMPARE/MERGE Operations
MGR_COMPARE_NODIFFS=0
MGR_COMPARE_DIFF_OR_ERROR=1
 .
 .
MGR_OP_COMPARE="compare"
```

The Bourne shell script listed in "Script for compare method" implements the compare method. (You can modify this script to implement the **xcompare** method as a slight variant of compare.)

Script for compare method

```
#!/bin/sh -e
MGRDIR=${CLEARCASEHOME:-/usr/rational}/lib/mgrs

#### read file that defines methods and exit statuses
. $MGR_DIR/mgr_info.sh

#### process all options: pass them through to cleardiff
OPTS=""
while (expr $1 : '\-' > /dev/null) ; do
    OPTS="$OPTS $1"
    if [ "$1" = "$MGR_FLAG_COLUMNS" ] ; then
        shift 1
        OPTS="$OPTS $1"
    fi
    shift 1
done

#### all remaining arguments ($*) are files to be compared
#### first, format each file with NROFF
COUNT=1
TMP=/usr/tmp/compare.$$
for X in $* ; do
    nroff -man $X | col | ul -Tcrt > $TMP.$COUNT
    COUNT='expr $COUNT + 1'
done

#### then, compare the files with cleardiff
cleardiff -quiet $OPTS $TMP.*

#### cleanup and return appropriate exit status
if [ $? -eq MGR_COMPARE_NODIFFS ] ; then
    rm -f $TMP.*
    exit MGR_COMPARE_NODIFFS
else
    rm -f $TMP.*
    exit MGR_COMPARE_DIFF_OR_ERROR
fi
```

Testing the type manager

Test a new type manager by using it on some Rational ClearCase host. This testing procedure need not be obtrusive. Because the type manager has a new name, no existing element type and, therefore, no existing element, uses it automatically. To place the type manager in service, create a new element type, create some test elements of that type, and run some tests.

The testing sequences that are described in "Creating a Test Element Type" and "Creating and Using a Test Element" continue the **mp_mgr** example.

Creating a Test Element Type: To make sure that an untested type manager is not used accidentally, associate it with a new element type, **manpage_test**, of which you are the only user.

```
% cleartool mkeltype –nc –supertype compressed_file \
   –manager mp_mgr manpage_test
% cleartool lock –nusers $USER eltype:manpage_test
```

Creating and Using a Test Element: These commands create a test element that uses the new type manager, and tests the various data-manipulation methods:

cd *directory-in-test-VOB*

cleartool checkout –nc .
(tests create_element method)

cleartool mkelem –eltype manpage_test –nc –nco test.1
(tests construct_version method)

cleartool checkout –nc test.1

vi test.1
(edit checked-out version)

cleartool checkin –c "first" test.1
(tests create_ version method)

cleartool checkout –nc test.1
(tests construct_ version method)

vi test.1
(edit checked-out version)

cleartool checkin –c "second" test.1
(tests create_ version method)

cleartool diff test.1@@/main/1 test.1@@/main/2
(tests compare method)

Installing and using the type manager

After a type manager is fully tested, make it available to all users with the following procedure.

1. Install the type manager.

   A VOB is a networkwide resource; it can be mounted on any Rational ClearCase host. But a type manager is a host resource: a separate copy must be installed on each host where Rational ClearCase client programs run. If the copy is not installed, elements of the new type cannot be used. (It need not be installed on hosts that serve only as repositories for VOBs, views, or VOBs and views.)

If the VOB is replicated, you must install the type manager at all sites. Custom type managers are not replicated.

To install the type manager on a particular host, create a subdirectory in *ccase–home–dir*/lib/mgrs, and populate it with the programs that implement the methods. You can create symbolic links across the network to a master copy on a server host.

2. Create element types.

   Create one or more element types that use the type manager, just as you did in "Testing the type manager" on page 238 (do not include "test" in the name of the element type). For example, you can name the element type **manpage** or **nroff_src**.

3. Convert existing elements.

   Have at least a few existing elements use the new type manager. The **chtype** command changes an element type:

   ```
   % cleartool chtype –force manpage path ...
   ```

   Permission to change an element type is restricted to the element owner, the VOB owner, and the *root* user.

4. Revise magic files.

   If you want the new element types to be used automatically for certain newly created elements, create (or update) a **local.magic** file in each host *ccase–home–dir*/config/magic directory:

   ```
   manpage src_file text_file file: -name "*.[1-9]" ;
   ```

5. Inform the project team (and other teams, if appropriate).

   Advertise the new element types to all team members, describing the features and benefits of the new type manager. Be sure to provide directions on how to gain access to the new functionality automatically (through file names that match magic-file rules) and explicitly (with **mkelem –eltype**).

Icon use by GUI browsers

The File Browser can display file system objects either by list or graphically. In the latter case, the File Browser selects an icon for each file system object as follows:

1. The object name or its contents determines a list of file types, as described in "How element types are assigned" on page 230.
2. One by one, the file types are compared to the rules in predefined and user-defined icon files, as described in the **cc.icon** reference page. For example, the file type **c_source** matches this icon file rule:

   ```
   c_source : -icon c ;
   ```

   When a match is found, the search ends. The token that follows **–icon** names the file that contains the icon to be displayed.
3. The File Browser searches for the file, which must be in **bitmap(1)** format, in directory $HOME/.bitmaps, or *ccase–home–dir*/config/ui/bitmaps, or the directories specified by the environment variable BITMAP_PATH.
4. If a valid bitmap file is found, the File Browser displays it; otherwise, the search for an icon continues with the next file type.

The name of an icon file must include a numeric extension, which need not be specified in the icon file rule. The extension specifies how much screen space the File Browser must allocate for the icon. Each bitmap supplied with Rational ClearCase version control is stored in a file with a **.40** suffix (for example, lib.40), which indicates a 40x40 icon.

This procedure causes the File Browser to display manual page source files with a customized icon. All manual pages have file type **manpage**.

1. Add a rule to your personal magic file (in directory $HOME/.magic) that includes **manpage** among the file types assigned to all manual page source files:

   ```
   manpage src_file text_file file:  -name "*.[1-9]" ;
   ```

2. Add a rule to your personal icon file (in directory $HOME/.icon) that maps **manpage** to a user-defined bitmap file:

   ```
   manpage : -icon manual_page_icon ;
   ```

3. Create a **manpage** icon in your personal bitmaps directory ($HOME/.bitmaps) by revising one of the standard icon bitmaps with the standard X **bitmap** utility:

   ```
   % mkdir $HOME/.bitmaps
   % cd $HOME/.bitmaps
   % cp $RATIONALHOME/config/ui/bitmaps/c.40 manual_page_icon.40
   % bitmap manual_page_icon.40
   ```

4. Test your work by having the File Browser display a manual page source file.

Chapter 16. Using Rational ClearCase throughout the development cycle

The previous chapters (Chapter 11, "Defining project views," on page 163 through Chapter 15, "Using element types to customize file element processing," on page 229) describe various aspects of managing a project with Rational ClearCase. This chapter describes typical usage by a developer.

About using Rational ClearCase throughout the development cycle

It is helpful to understand one way in which you can use Rational ClearCase to organize the work throughout a development cycle for a project. During a development cycle, developers create a new release and maintain the previous release.

You should understand concepts and methods to address typical organizational needs. There are many other approaches that are supported. For example, instead of using command-line tools that are described here, consider using graphic user interface (GUI) tools such as the Merge Manager to accomplish similar goals.

Project overview

Release 2.0 development of the **monet** project includes the following kinds of work:

- Patches. Several high-priority bug fixes to Release 1.0 are needed.
- Minor enhancements. Some commands need new options; some option names need to be shortened (for example, **–recursive** becomes **–r**); some algorithms need performance work.
- Major new features. A graphic user interface is required, as are many new commands and internationalization support.

These three development efforts can proceed largely in parallel (see Figure 58), but critical dependencies and milestones must be considered:

- Several Release 1.0 patch releases will ship before Release 2.0 is complete.
- New features take longer to complete than minor enhancements.
- Some new features depend on the minor enhancements.

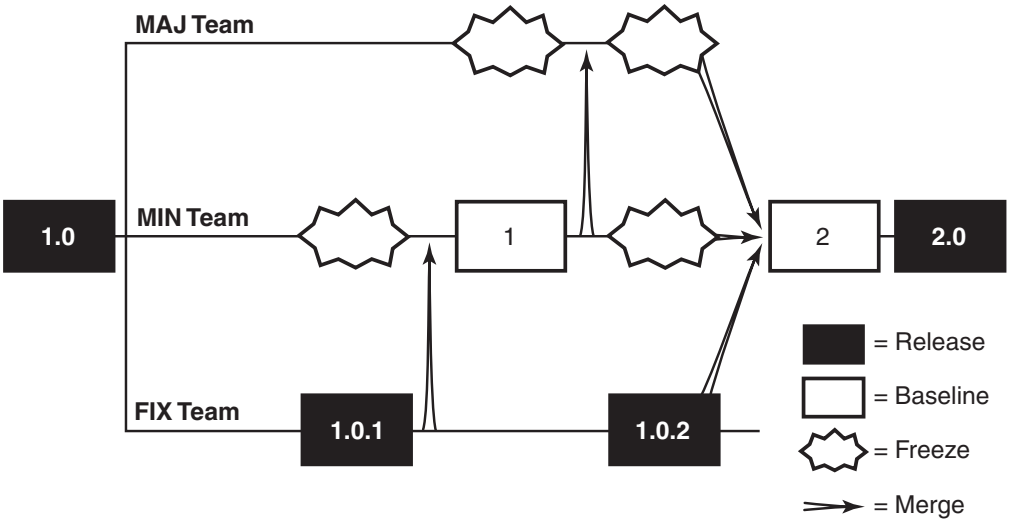


Figure 58. Project plan for Release 2.0 development

The plan uses a baseline-plus-changes approach. Periodically, developers stop writing new code, and spend some time integrating their work, building, and testing. The result is a *baseline*: a stable, working version of the application. You can integrate product enhancements incrementally and frequently. The more frequent the baselines, the easier the tasks of merging work and testing the results.

After a baseline is produced, active development resumes; any new efforts begin with the set of source versions that went into the baseline build.

You define a baseline by assigning the same version label (for example, **R2_BL1** for Release 2.0, Baseline 1) to all the versions that go into, or are produced by, the baseline build.

The project team is divided into three smaller teams, each working on a different development effort: the MAJ team (new features), the MIN team (minor enhancements), and the FIX team (Release 1.0 bug fixes and patches). Some developers may belong to multiple teams. These developers work in multiple views, each configured for the respective team tasks.

Product Note: In the examples that follow, arguments that show multicomponent VOB tags, such as /vobs/monet, do not apply to Rational ClearCase LT on the UNIX system, which recognizes only single-component VOB tags, such as /vobs_monet.

The development area for the **monet** project is shown here.

```
/vobs/monet                 (project top-level directory)
        src/                (sources)
        include/            (include files)
        lib/                (shared libraries)
```

At the beginning of Release 2.0 development, the most recent versions on the **main** branch are labeled **R1.0**.

Development strategy

This section describes the Rational ClearCase issues to be resolved before development begins.

Project manager and Rational ClearCase administrator

In most development efforts, the project manager and the system administrator are different people. The user name of the project manager is **meister**. The administrator is the **vobadm** user, who creates and owns the **monet** and **libpub** VOBs.

Use of branches

In general, different kinds of work are done on different branches. The Release 1.0 bug fixes, for example, are made on a separate branch to isolate this work from new development. The FIX team can then create patch releases that do not include any of the Release 2.0 enhancements or incompatibilities.

Because the MIN team will produce the first baseline release on its own, the project manager gives the **main** branch to this team. The MAJ team will develop new features on a subbranch, and will not be ready to integrate for a while; the FIX team will fix Release 1.0 bugs on another subbranch and can integrate its changes at any time.

Each new feature can be developed on its own subbranch, to better manage integration and testing work. For simplicity, this chapter assumes that work for new features is done on a single branch.

The project manager has created the first baseline from versions on the **main** branches of their elements. But this is not a requirement; you can create a release that uses versions on any branch, or combination of branches.

The evolution of a typical element during Release 2.0 development is shown in Figure 59.

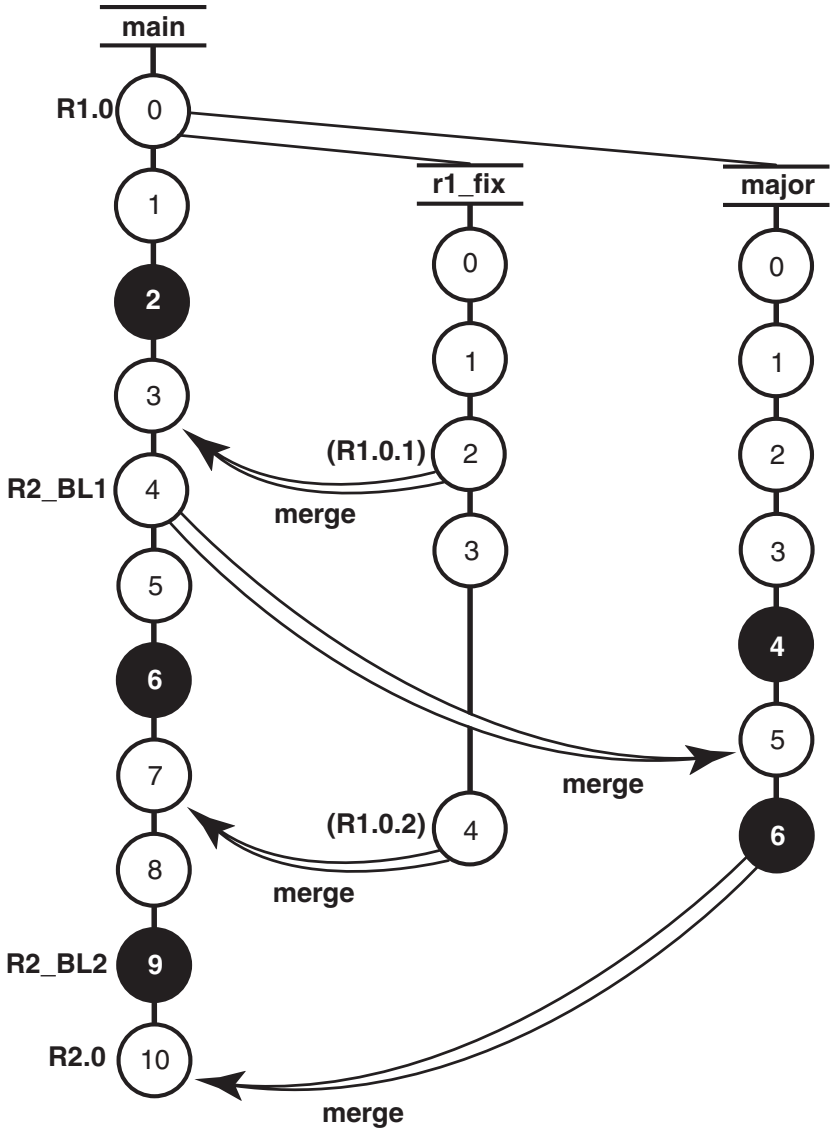


Figure 59. Development milestones: evolution of a typical element

The evolution of the element proceeds in the following steps:

1. Start minor and major enhancements, along with **R1.0** bug fixing (all branches).
2. Freeze minor enhancements work (**main** branch).
3. Merge bug fixes from Release 1.0.1 into minor enhancements (**main**).
4. Create Baseline 1 release (**main**).
5. Freeze major enhancements work (**major**).
6. Merge Baseline 1 changes into major enhancements (**major**).
7. Freeze minor enhancements work (**main**).
8. Merge additional bugfixes into minor enhancements (**main**).
9. Freeze major enhancements work (**major**).
10. Merge major enhancements work with minor enhancements work (**main**).
11. Create Baseline 2 release (**main**).
12. Begin Final testing (**main**).

13. Release 2.0 is done (**main**).

Creating project views

The MAJ team works on a branch named **major** and uses this config spec:

```
(1)     element * CHECKEDOUT
(2)     element * .../major/LATEST
(3)     element * R1.0 –mkbranch major
(4)     element * /main/LATEST –mkbranch major
```

The MIN team works on the **main** branch and uses the default config spec:

```
(1)     element * CHECKEDOUT
(2)     element * .../main/LATEST
```

The FIX team works on a branch named **r1_fix** and uses this config spec:

```
(1)     element * CHECKEDOUT
(2)     element * .../r1_fix/LATEST
(3)     element * R1.0 –mkbranch r1_fix
(4)     element * /main/LATEST –mkbranch r1_fix
```

For the MAJ and FIX teams, use of the *auto-make-branch* facility in Rule "3" on page 245 and Rule "4" on page 245 enforces consistent use of subbranches. It also relieves developers of the task of creating branches explicitly and ensures that all branches are created at the version labeled **R1.0**.

Creating branch types

The project manager creates the **major** and **r1_fix** branch types that are required for the config specs in "Creating project views" on page 245:

```
cleartool mkbrtype –c "monet R2 major enhancements" \
major@/vobs/libpub major@/vobs/monet
Created branch type "major".
Created branch type "major".

cleartool mkbrtype –c "monet R1 bugfixes" r1_fix@/vobs/libpub
r1_fix@/vobs/monet
Created branch type "r1_fix".
Created branch type "r1_fix".
```

Tip: Because each VOB has its own set of branch types, the branch types must be created separately in the **monet** VOB and the **libpub** VOB.

Creating standard config specs

To ensure that all developers in a team configure their views the same way, the project manager creates files containing standard config specs:

- /public/config_specs/MAJ contains the MAJ team's config spec.
- /public/config_specs/FIX contains the FIX team's config spec.

These config spec files are stored in a VOB but made available in a standard directory outside a VOB to ensure that all developers get the same version.

Creating, configuring, and registering views

Each developer creates a view under his or her home directory. For example, developer **arb** enters these commands:

```
% mkdir $HOME/view_store
% cleartool mkview –tag arb_major $HOME/view_store/arb_major.vws
Created view.
Host-local path: phobos:export/home/arb/view_store/arb_major.vws
Global path:     /net/phobos/export/home/arb/view_store/arb_major.vws
It has the following rights:
User : arb      : rwx
Group: user     : rwx
Other:          : r-x
```

A new view has the default config spec. Thus, developers on the MAJ and FIX teams must reconfigure their views, using the standard file for their team. Developer **arb** edits her config spec with the **cleartool edcs** command, deletes the existing lines, and adds the following line:

```
/public/config_specs/MAJ
```

If the project manager changes the standard file, **arb** must enter the command **cleartool setcs –current** to pick up the changes.

Development begins

To begin the project, a developer sets a properly configured view, checks out one or more elements, and starts work. For example, developer **david** on the MAJ team enters these commands:

```
% cleartool setview david_major
% cd /vobs/monet/src
% cleartool checkout –nc opt.c prs.c
Created branch "major" from "opt.c" version "/main/6".
Checked out "opt.c" from version "/main/major/0".
Created branch "major" from "prs.c" version "/main/7".
Checked out "prs.c" from version "/main/major/0".
```

The auto-make-branch facility causes each element to be checked out on the **major** branch (see Rule “4” on page 245 in the MAJ team’s config spec in “Creating project views” on page 245). If a developer on the MIN team enters this command, the elements are checked out on the **main** branch, with no conflict.

Rational ClearCase is fully compatible with standard development tools and practices. Thus, developers use the editing, compilation, and debugging tools that they prefer (including personal scripts and aliases) while working in their views.

Developers check in work periodically to make their work available to other team members (that is, those whose views select the most recent version on the team’s branch). This allows intra-team integration and testing to proceed throughout the development period.

Techniques for isolating your work

Individual developers may need or prefer to isolate their work from the changes made by other team members. To do so, they can use these techniques to configure their views:

- Time rules. When someone checks in an incompatible change, a developer can re-configure the view to select the versions at a point before those changes were made.
- Private subbranches. A developer can create a private subbranch in one or more elements (for example, `/main/major/anne_wk`). The config spec must be changed to select versions on the `/main/major/anne_wk` branch instead of versions on the `/main/major` branch.

- Viewing only their own revisions. Developers can use a Rational ClearCase query to configure a view that selects only their own revisions to the source tree.

Creating baseline 1

The MIN team has implemented and tested the first group of minor enhancements, and the FIX team has produced a patch release, whose versions are labeled **R1.0.1**. It is time to combine these efforts, to produce Baseline 1 of Release 2.0 (Figure 60).

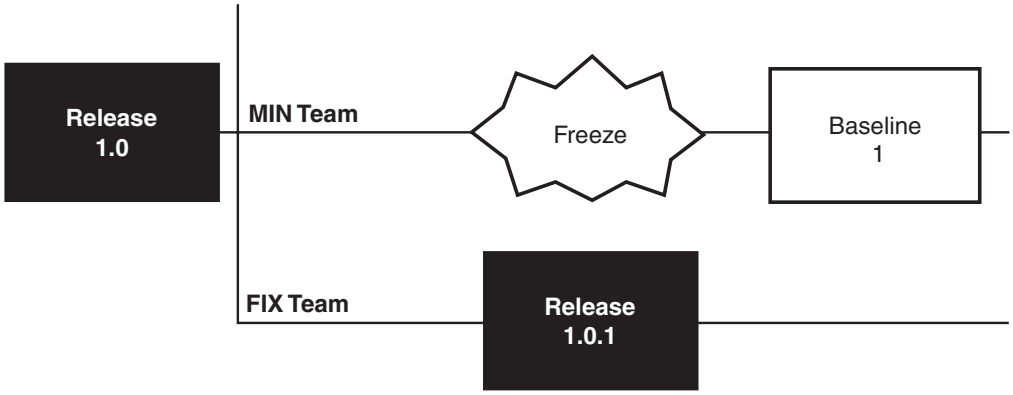


Figure 60. Creating baseline 1

Merging two branches

The project manager asks the MIN developers to merge the **R1.0.1** changes from the **r1_fix** branch to their own branch (**main**). All the changes can be merged by using the **findmerge** command once. For example:

```
% cleartool findmerge /vobs/libpub /vobs/monet/src \
   -fversion .../r1_fix/LATEST -merge -graphical
```

The output from the **findmerge** command describes the versions that are merged.

Integration and test

After the merges are complete, the `/main/LATEST` versions of certain elements represent the efforts of the MIN and FIX teams. Members of the MIN team now compile and test the **monet** application to find and fix incompatibilities in the work of both teams.

The developers on the MIN team integrate their changes in a single, shared view. The project manager creates the view storage area in a location that is accessible from all developer hosts:

```
% umask 2
% mkdir /netwide/public
% cleartool mkview -tag base1_vu /netwide/public/base1_vu.vws
Created view.
Host-local path: infinity:/netwide/public/base1_vu.vws
Global path:     /net/infinity/netwide/public/base1_vu.vws.
It has the following rights:
User : meister  : rwx
Group: mon      : rwx
Other:          : r-x
```

Because all integration work takes place on the **main** branch, there is no need to change the configuration of the new view from the Rational ClearCase default. MIN developers set this view (**cleartool setview base1_vu**) and coordinate builds

and tests of the **monet** application. Because they are sharing a single view, the developers are careful not to overwrite each other's view-private files. Any new versions that are created to fix inconsistencies (and other bugs) go onto the **main** branch.

Labeling sources

The **monet** application minor enhancements and bug fixes are now integrated, and a clean build has been performed in view **base1_vu**. To create the baseline, the project manager assigns the same version label, **R2_BL1**, to the /main/LATEST versions of all source elements. He begins by creating an appropriate label type:

```
% cleartool mklbtype –c "Release, Baseline 1" R2_BL1@/vobs/monet
R2_BL1@/vobs/libpub
Created label type "R2_BL1".
Created label type "R2_BL1".
```

He then locks the label type, preventing all developers (except himself) from using it:

```
% cleartool lock –nusers meister lbtype:R2_BL1@/vobs/monet
lbtype:R2_BL1@/vobs/libpub
Locked label type "R2_BL1".
Locked label type "R2_BL1".
```

Before applying labels, he verifies that all elements are checked in on the **main** branch (checkouts on other branches are still allowed):

```
% cleartool lscheckout –all /vobs/monet /vobs/libpub
```

No output from this command indicates that all elements for the **monet** project are checked in. Now, the project manager attaches the **R2_BL1** label to the currently selected version (/main/LATEST) of every element in the two VOBs:

```
% cleartool mklabel –recurse R2_BL1 /vobs/monet /vobs/libpub
Created label "R2_BL1" on "/vobs/monet" version "/main/1".
Created label "R2_BL1" on "/vobs/monet/src" version "/main/3".
 <many more label messages>
```

Removing the integration view

The view registered as **base1_vu** is no longer needed, so the project manager removes it:

```
% cleartool rmview –force –tag base1_vu
```

Merging ongoing development work

After Baseline 1 is created, the MAJ team merges the Baseline 1 changes into its work (Figure 61). The team now has access to the minor enhancements it needs for further development. Team members also have an early opportunity to determine whether any of their changes are incompatible.

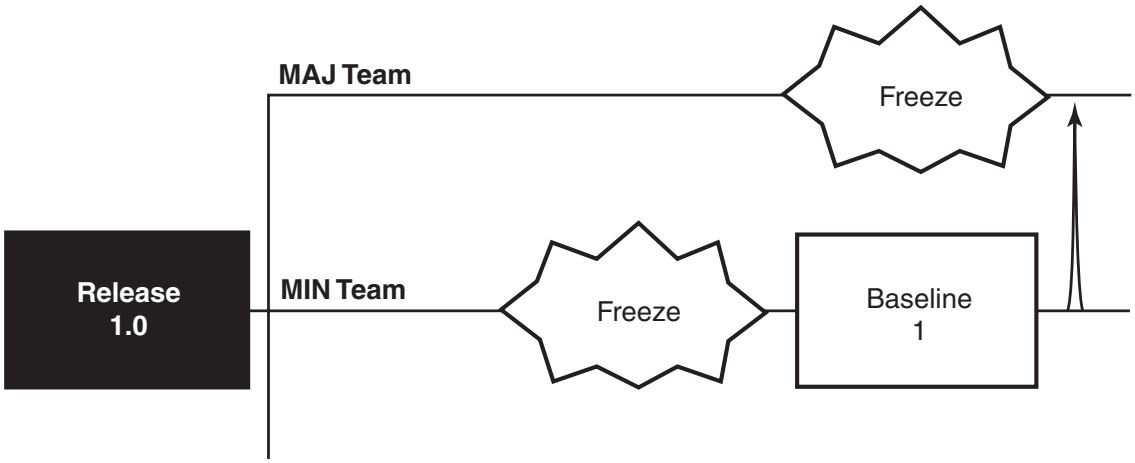


Figure 61. Updating major enhancements development

Accordingly, the project manager declares a freeze of major enhancements development. MAJ team members check in all elements and verify that the **monet** application builds and runs, making small source changes as necessary. When all such changes have been checked in, the team has a consistent set of `/main/major/LATEST` versions.

Tip: Developers working on other major enhancements branches can merge at other times, using the same merge procedures described here.

Preparing to merge

1. The project manager makes sure that no element is checked out on the **major** branch:

   ```
   % cleartool lscheckout –all /vobs/monet /vobs/libpub
   ```

   Tip: Any MAJ team members who want to continue with nonmerge work can create a subbranch at the "frozen" version (or work with a version that is checked out as unreserved).

2. The project manager performs any required directory merges:

   ```
   % cleartool setview major_vu
   ```

   Any MAJ team view can be used.

   ```
   % cleartool findmerge /vobs/monet /vobs/libpub –type d \
   –fversion /main/LATEST –merge
   Needs merge /vobs/monet/src [automatic to /main/major/3 from
   /main/LATEST]
     .
     .
     .
   Log has been written to "findmerge.log.04-Feb-04.09:58:25".
   ```

   The output log describes the **findmerge** actions.

3. After checking in the files, the project manager determines which elements need to be merged:

   ```
   % cleartool findmerge /vobs/monet /vobs/libpub –fversion /main/LATEST –print
     .
     .
     .
   A 'findmerge' log has been written to
   "findmerge.log.04-Feb-04.10:01:23"
   ```

The output log describes the **findmerge** actions. This last **findmerge** log file is in the form of a shell script: it contains a series of **cleartool findmerge** commands, each of which performs the required merge for one element:

```
% cat findmerge.log.04-Feb-04.10:01:23
cleartool findmerge /vobs/monet/src/opt.c@@/main/major/1 -fver /main/LATEST –merge
cleartool findmerge /vobs/monet/src/prs.c@@/main/major/3 -fver /main/LATEST –merge
  .
  .
cleartool findmerge /vobs/libpub/src/dcanon.c@@/main/major/3 -fver /main/LATEST -merge
cleartool findmerge /vobs/libpub/src/getcwd.c@@/main/major/2 -fver /main/LATEST -merge
cleartool findmerge /vobs/libpub/src/lineseq.c@@/main/major/10 -fver /main/LATEST -merge
```

4. The project manager locks the **major** branch, allowing it to be used only by the developers who are performing the merges:

   ```
   cleartool lock –nusers meister,arb,david,sakai brtype:major@/vobs/monet \
   brtype:major@/vobs/libpub
   Locked branch type "major".
   Locked branch type "major".
   ```

Merging work

Because the MAJ team is not contributing to a baseline soon, it is not necessary to merge work (and test the results) in a shared view. MAJ developers can continue working in their own views.

Periodically, the project manager sends an excerpt from the **findmerge** log to an individual developer, who executes the commands and monitors the results. (The developer can send the resulting log files back to the project manager, as confirmation of the merge activity.)

A merged version of an element includes changes from three development efforts: Release 1.0 bug fixing, minor enhancements, and new features (see Figure 62).

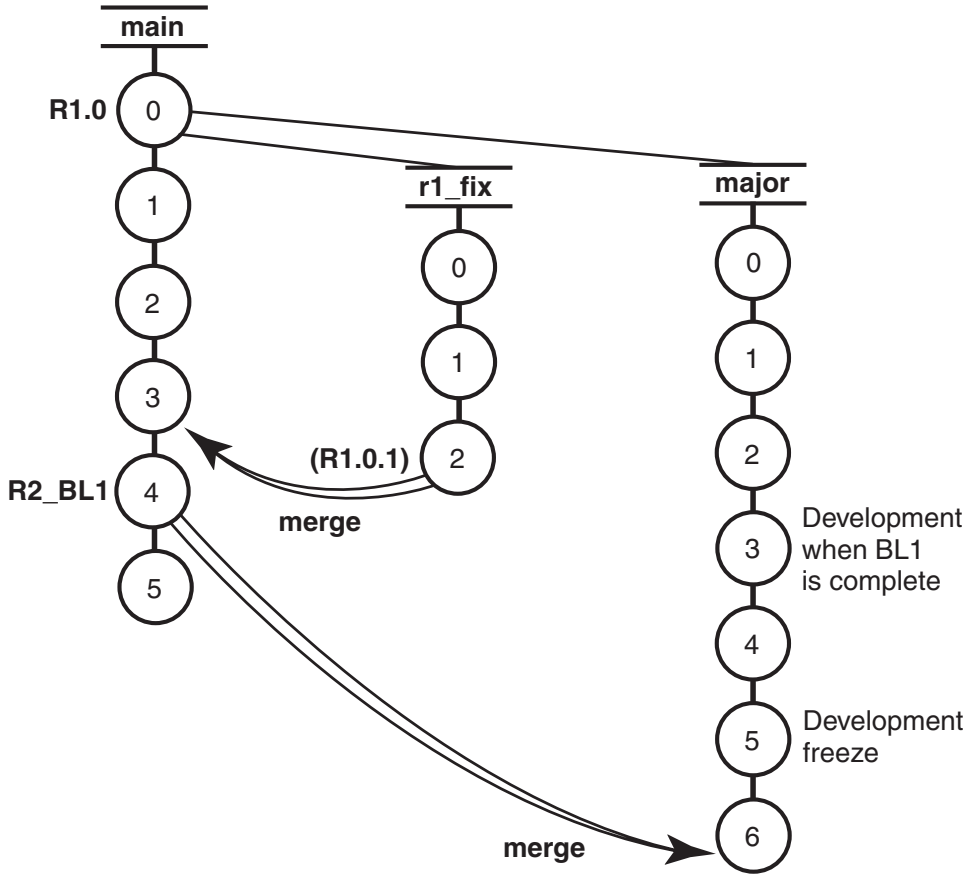


Figure 62. Merging Baseline 1 changes into the major branch

The project manager verifies that no more merges are needed, by entering a **findmerge** command with the **–whynot** option:

```
% cleartool findmerge /vobs/monet /vobs/libpub –fversion /main/LATEST –whynot –print
  .
  .
No merge "/vobs/monet/src" [/main/major/4 already merged from /main/3]
No merge "/vobs/monet/src/opt.c" [/main/major/2 already merged from
/main/12]
  .
  .
```

The merge period ends when the project manager removes the lock on the **major** branch:

```
% cleartool unlock brtype:major@/vobs/monet brtype:major@/vobs/libpub
Unlocked branch type "major".
Unlocked branch type "major".
```

Creating Baseline 2

The MIN team is ready to freeze for Baseline 2, and the MAJ team will be soon (see Figure 63).

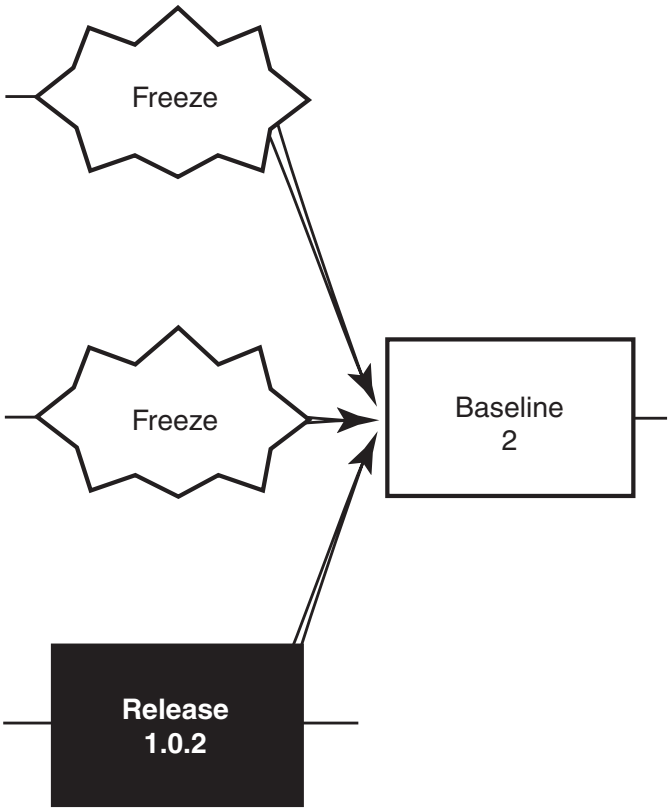


Figure 63. Baseline 2

Baseline 2 integrates all three development efforts, and thus requires two sets of merges:

- Bug fix changes from the most recent patch release (versions labeled **R1.0.2**) must be merged to the **main** branch.
- New features must be merged from the **major** branch to the **main** branch. (This is the opposite direction from the merges described in "Merging ongoing development work" on page 248.)

Merges can be done from more than two directions, so both the bug fixes and the new features can be merged to the **main** branch at the same time. In general, though, it is easier to verify the results of two-way merges.

Merging from the r1_fix branch

The first set of merges is almost identical to those described in "Merging two branches" on page 247.

Preparing to merge from the major branch

After the integration of the **r1_fix** branch is completed, the project manager prepares to manage the merges from the **major** branch. These merges are performed in a tightly controlled environment, because the Baseline 2 milestone is approaching and the **major** branch is to be abandoned.

Tip: It is probably more realistic to build and verify the application, and then apply version labels before proceeding to the next merge.

The project manager verifies that everything is checked in on both the **main** branch and **major** branches:

```
% cleartool lscheckout –brtype main –recurse /vobs/monet /vobs/libpub
% cleartool lscheckout –brtype major –recurse /vobs/monet /vobs/libpub
%
```

No output from these commands indicates that no element is checked out on either its **main** branch or its **major** branch.

Next, the project manager determines which elements require merges:

```
% cleartool setview minor_vu
```

Any MIN team view can be used.

```
% cleartool findmerge /vobs/monet /vobs/libpub –fversion .../major/LATEST –print
  .
  .
  .
A 'findmerge' log has been written to
"findmerge.log.26-Feb-99.19:18:14"
```

All development on the **major** branch will stop after this baseline. Thus, the project manager locks the **major** branch to all users, except those who are performing the merges. Locking allows the merges to be recorded with a hyperlink of type **Merge**:

```
% cleartool lock –nusers arb,david brtype:major@/vobs/monet
brtype:major@/vobs/libpub
Locked branch type "major".
Locked branch type "major".
```

Because the **main** branch will be used for Baseline 2 integration by a small group of developers, the project manager asked **vobadm** to lock the **main** branch to everyone else:

```
% cleartool lock –nusers meister,arb,david,sakai \
brtype:main@/vobs/monet brtype:main@/vobs/libpub
Locked branch type "main".
Locked branch type "main".
```

To lock the branch, you must be the branch creator, element owner, VOB owner, or *root* user (on the UNIX system) or a member of the *ClearCase administrators group* (on the Windows system). See the **lock** reference page.

Merging from the major branch

Because the **main** branch is the destination of the merges, developers work in a view with the default config spec. The situation is similar to the one described in “Preparing to merge” on page 249. This time, the merges take place in the opposite direction, from the **major** branch to the **main** branch. Accordingly, the **findmerge** command is very similar:

```
% cleartool findmerge /vobs/monet /vobs/libpub –fversion /main/major/LATEST \
–merge –graphical
  .
  .
  .
A 'findmerge' log has been written to
"findmerge.log.23-Mar-99.14:11:53"
```

After checkin, the version tree of a typical merged element appears as in Figure 64.

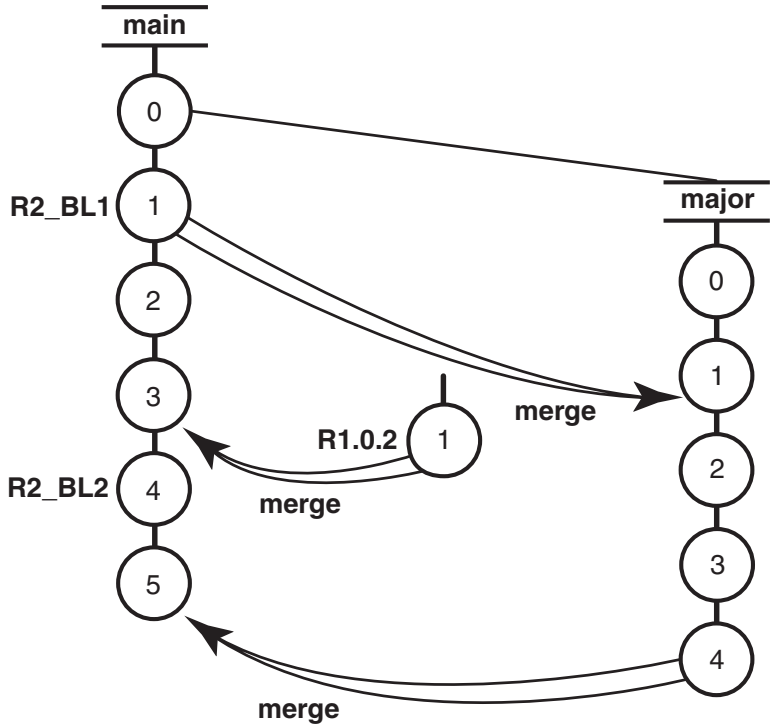


Figure 64. Element structure after the pre-Baseline-2 merge

Decommissioning the major branch

After all data has been merged to the **main** branch, development on the **major** branch will stop. The project manager enforces this policy by making the **major** branch obsolete:

```
% cleartool lock -replace -obsolete brtype:major@/vobs/monet
brtype:major@/vobs/libpub
Locked branch type "major".
Locked branch type "major".
```

Integration and test

Structurally, the Baseline 2 integration-and-test phase is identical to the one for Baseline 1 (see "Integration and test" on page 247). At the end of the integration period, the project manager attaches version label **R2_BL2** to the `/main/LATEST` version of each element in the **monet** and **libpub** VOBs. (The Baseline 1 version label was **R2_BL1**.)

Final validation: creating Release 2.0

Baseline 2 has been released internally, and further testing has found only minor bugs. These bugs have been fixed by creating new versions on the **main** branch (see Figure 65).

Figure 65. Final test and release

Before the **monet** application is shipped to customers, it goes through a validation phase:

- All editing, building, and testing is restricted to a single, shared view.
- All builds are performed from sources with a particular version label (**R2.0**).
- Only the project manager has permission to make changes involving that label.
- All labels must be moved by hand.
- Only high-priority bugs are fixed, using this procedure:
  - The project manager authorizes a particular developer to fix the bug, by granting her permission to create new versions (on the **main** branch).
  - The developer's checkin activity is tracked by a Rational ClearCase trigger.
  - After the bug is fixed, the project manager moves the **R2.0** version label to the fixed version and revokes the developer's permission to create new versions.

Labeling sources

In a view with the default config spec, the project manager creates the **R2.0** label type and locks it:

```
cleartool mklbtype –c "Release 2.0" R2.0@/vobs/monet R2.0@/vobs/libpub
Created label type "R2.0".
Created label type "R2.0".
cleartool lock –nusers meister lbtype:R2.0@/vobs/monet lbtype:R2.0@/vobs/libpub
Locked label type "R2.0".
Locked label type "R2.0".
```

The project manager labels the /main/LATEST versions throughout the entire **monet** and **libpub** development trees:

```
cleartool mklabel –recurse R2.0 /vobs/monet /vobs/libpub
```

Many label messages are displayed. During the final test phase, the project manager moves the label forward, using **mklabel –replace**, if any new versions are created.

Restricting use of the main branch

At this point, use of the **main** branch is restricted to a few users: those who performed the merges and integration leading up to Baseline 2 (see "Merging from the major branch" on page 253). Now, the project manager asks **vobadm** to close down the **main** branch to everyone except himself, **meister**:

```
% cleartool lock –replace –nusers meister brtype:main
Locked branch type "main".
```

The **main** branch is opened only for last-minute bug fixes (see "Fixing a final bug" on page 256.)

Setting up the test view

The project manager creates a new shared view, **r2_vu**, that is configured with a one-rule config spec:

```
% umask 2
% cleartool mkview –tag r2_vu /public/integrate_r2.vws
% cleartool edcs –tag r2_vu
```

This is the config spec:

```
element * R2.0
```

This config spec guarantees that only properly labeled versions are included in final validation builds.

Setting up the trigger to monitor bug-fixing

The project manager places a trigger on all elements in the **monet** and **libpub** VOBs; the trigger fires whenever a new version of any element is checked in. First, he creates a script that sends mail (for an example script, see "Notify team members of relevant changes" on page 183).

Then, he asks **vobadm** to create an all-element trigger type in the **monet** and **libpub** VOBs, specifying the script as the trigger action:

```
% cleartool mktrtype –nc -element –all –postop checkin –brtype main \
–exec /public/scripts/notify_manager.sh \
r2_checkin@/vobs/monet r2_checkin@/vobs/libpub
Created trigger type "r2_checkin".
Created trigger type "r2_checkin".
```

Only the VOB owner or **root** user (on the UNIX system) or a member of the Rational ClearCase administrators group (on the Windows system) can create trigger types.

Fixing a final bug

This section demonstrates the final validation environment in action. Developer **arb** discovers a serious bug and requests permission to fix it. The project manager grants her permission to create new versions on the **main** branch, by having **vobadm** enter this command.

```
% cleartool lock –replace –nusers arb,meister brtype:main
Locked branch type "main".
```

Developer **arb** fixes the bug in a view with the default config spec and tests the fix there. This involves creating two new versions of element **prs.c** and one new version of element **opt.c**. Each time **arb** uses the **checkin** command, the **r2_checkin** trigger sends mail to the project manager. For example:

```
Subject: Checkin /vobs/monet/src/opt.c by arb
/vobs/monet/src/opt.c@@/main/9
Checked in by arb.

Comments:
fixed bug #459: made buffer larger
```

When regression tests verify that the bug has been fixed, the project manager revokes **arb**'s permission to create new versions. Once again, the command is executed by **vobadm**:

```
% cleartool lock –replace –nusers meister brtype:main
Locked branch type "main".
```

The project manager then moves the version labels to the new versions of **prs.c** and **opt.c**, as indicated in the mail messages. For example:

```
% cleartool mklabel –replace R2.0 /vobs/monet/src/opt.c@@/main/9
Moved label "R2.0" on "prs.c" from version "/main/8" to "/main/9".
```

Rebuilding from labels

After the labels have been moved, developers rebuild the **monet** application again, to verify that a good build can be performed using only those versions labeled **R2.0.**

Wrapping up

When the final build in the **r2_vu** view passes the final test, Release 2.0 of **monet** is ready to ship. After the distribution medium has been created from derived objects in the **r2_vu** view, the project manager asks the Rational ClearCase administrator to clean up and prepare for the next release:

- The Rational ClearCase administrator deletes the all-element trigger type to remove the checkin triggers from all elements:

  ```
  cleartool rmtype trtype:r2_checkin@/vobs/monet
  trtype:r2_checkin@/vobs/libpub
  Removed trigger type "r2_checkin".
  Removed trigger type "r2_checkin".
  ```

- The Rational ClearCase administrator reopens the **main** branch:

  ```
  cleartool unlock brtype:main
  Unlocked branch type "main".
  ```

Part 4. Appendixes

Appendix A. Moving from view profiles to UCM

This appendix compares view profile features with UCM features and describes how to move a project from view profiles to UCM.

View profiles and UCM

Product Note: View profiles are available only with Rational ClearCase on the Windows system. Rational ClearCase LT does not support view profiles.

A set of features called *view profiles* were included to automate much of the work required to set up and maintain your team's shared Rational ClearCase configuration. The Unified Change Management (UCM) process provides a more complete solution for organizing software development teams. If you currently use view profiles, you may want to move to UCM.

Feature comparison

The features of view profiles and UCM are similar in a few ways and different in many other ways.

Branches and streams

In UCM, the *project* and its *integration stream* take the place of the view profile. Views attached to the integration stream are configured to select the project shared integration branch, just as a view profile config spec selects a shared common branch.

In view profiles, developers can work independently by setting up private branches for development work. In UCM, team members join a project at which time they create their own development work areas. A development work area consists of a development stream and a development view.

Moving work among branches or streams

When working on a private branch in view profiles, there is no automated way to incorporate changes from other developers onto the private branch. In UCM, developers use the *rebase* operation to update their development work areas with the latest work delivered by other developers to the integration stream and incorporated into a baseline.

In view profiles, developers finish a private branch when they complete a task. Finishing a private branch closes that branch and merges work to the integration branch, where it is merged with other sources. In UCM, *activities* record the versions that you create to complete a development task as *change sets*. The *deliver* operation moves activities from the development stream to the integration stream or a feature-specific development stream. Your development stream remains in place after a deliver operation, and you can continue to work in it.

VOBs and components

View profiles contain a list of VOBs that hold project data. UCM projects organize directory and file elements into components, and each stream keeps a list of components.

Checkpoints and baselines

View profiles capture stable configurations of a project with checkpoints, a set of labeled versions. UCM uses baselines, which capture a set of versions per component.

Table 8 summarizes the key differences between view profiles and UCM features.

Table 8. View profile features and their UCM counterparts

View profile construct	UCM counterpart
View profile	Project and integration stream
Integration branch	Integration stream
Private branch	Development stream
Set up private branch	Create a development stream/join project
Finish private branch	Deliver work to integration stream
Branch is closed when work is completed and merged to integration branch.	Development stream is not closed after a deliver operation.
No automated support for updating private branch with work from other developers.	Rebase operation adds changes from the integration stream to private work area.
Views are configured with information from profiles.	Views are configured with information from streams.

Moving view profile information to UCM

You may have to know how to move projects from view profiles to UCM.

Preparing your view profile project

Before moving work to UCM, finish all private branches. Work on private branches cannot be moved directly to a UCM project. After work has been merged into the integration branch, create a checkpoint that labels all versions to be migrated to the UCM project.

Moving the view profile information

1. Convert each VOB of the view profile project into a component.
2. For each component, import the label used for the checkpoint created in Step 1 on page 262. By importing a label, you are creating a new baseline for each component.
3. Create a UCM project, adding each baseline created in Step 2 on page 262.

Members of the project team can now join the project, creating their own development streams and views.

For more information about creating a UCM project, see Chapter 6, “Setting up the project,” on page 85.

Appendix B. Rational ClearCase integrations with Rational ClearQuest

In Rational ClearCase, two integrations with Rational ClearQuest are supported. This appendix provides information that you need to manage both integrations in the same development environment.

Understanding the Rational ClearCase integrations with Rational ClearQuest

There are two separate integrations with Rational ClearQuest:

- The base ClearCase integration with Rational ClearQuest
- The UCM integration with Rational ClearQuest

The integrations associate one or more Rational ClearQuest user database records with one or more Rational ClearCase versions, allowing you to use features of both Rational ClearCase and Rational ClearQuest. For information on setting up the base ClearCase integration with Rational ClearQuest, see "Setting up the Rational ClearQuest user database for base ClearCase" on page 196. Note that this integration cannot be used with UCM projects.

For more information on the UCM integration with Rational ClearQuest, see Chapter 5, "Setting up a Rational ClearQuest user database for UCM," on page 75.

In general, you should use the base ClearCase integration with Rational ClearQuest and the UCM integration with Rational ClearQuest separately, and avoid using a common Rational ClearQuest user database. However, it is possible for both integrations to use the same Rational ClearQuest user database. This can be useful if you are moving a project to UCM and have a substantial amount of information in a Rational ClearQuest user database that was created with the base ClearCase integration with Rational ClearQuest. You may want the new work in UCM to be reflected in new Rational ClearQuest records in the same Rational ClearQuest user database.

You should be aware of the considerations in managing the coexistence of the base ClearCase integration with Rational ClearQuest and the UCM integration with Rational ClearQuest.

Managing coexisting integrations

When a Rational ClearQuest user database that had been integrated with Rational ClearCase previously is configured for integration with UCM, the existing change sets are preserved intact in the Rational ClearQuest user database, but cannot be migrated to the UCM integration with Rational ClearQuest.

Change sets of existing records in the Rational ClearQuest user database are preserved, and you can access them from a Rational ClearQuest client. To continue work on a task in a project that has been migrated to UCM, create a new, corresponding, UCM activity and continue work there.

See "Planning how to use the UCM integration with Rational ClearQuest" on page 57 for related information.

Schema usage with both integrations

A Rational ClearQuest schema can contain modifications from both the base ClearCase integration with Rational ClearQuest and the UCM integration with Rational ClearQuest. A record type in such a schema would include both the Rational ClearCase package and the Unified Change Management package.

An individual record of that record type can store either Rational ClearCase or UCM change set information, but not both.

Presentation

The form for a record type that uses both integrations includes two tabs to show the change set information associated with each integration. The **Unified Change Management** tab lists the change set for a UCM activity. The **ClearCase** tab shows the change set associated with a Rational ClearQuest record.

Appendix C. Customizing Rational ClearCase Reports

This appendix explains how to customize Rational ClearCase Reports. Specifically, it introduces the Rational ClearCase Reports Programming Interface and gives examples of how you can customize the report procedures and the user interface.

Product Note: Rational ClearCase Reports is available only with the Windows versions of Rational ClearCase and Rational ClearCase LT.

How Rational ClearCase Reports works

Rational ClearCase Reports consists of two parts:

- The report procedures, which you can modify
- The Rational ClearCase Reports applications (Report Builder and Report Viewer), which you cannot modify

The report procedures are hooks into the applications; they implement all the operations necessary to generate and view a specific report. The applications collect user input, interpret it, and run the appropriate report procedure. At run time, Rational ClearCase Reports executes as two applications: Report Builder and Report Viewer. The Report Builder is used to select and define report parameters; the Report Viewer is used to view the report output.

All report procedures require an interface specification. This specification determines the user interface information presented to users in the Report Builder and Report Viewer. When users select a folder, the Report Builder scans the interface specification of each report in the associated subdirectory and places the contents in a temporary buffer. When users select a specific report, Report Builder extracts from this buffer the interface information associated with the report that is displayed in the Report Builder and Report Viewer. After users provide the required report parameters, the Report Builder generates the report and passes the data to the Report Viewer.

The commands that the Report Builder uses include an **-i** option, which extracts the interface specification from the report procedure. If the report procedure does not include an interface specification or if the structure and contents of that specification are not what the Report Builder expects, report processing stops.

For more information on the processing sequence between the Rational ClearCase Reports applications and the report procedures, see "Run-Time processing sequence for Reports programming interface" on page 266.

What you can customize in Rational ClearCase Reports

The Rational ClearCase programming interface enables you to customize four parts of the Report Builder user interface and two parts of the Report Viewer. You can customize by adding, changing, or removing information for the changeable areas of the Report Builder:

- The name of the folders in the tree pane.
- The directory organization displayed in the tree pane.
- The report description.

- The report parameters

As with the Report Builder, you can customize the Report Viewer. Add, change, or remove information for the changeable areas as follows:

- The position of a column heading can be moved, a column heading name can be added, modified, or deleted and a default sort order can be added or removed from any column heading.
- The commands on the pop-up menu.

For programming examples that demonstrate how you can make these customizations, see "Report programming examples" on page 276.

Run-Time processing sequence for Reports programming interface

Before you begin to customize report procedures, it is important to understand the run-time processing flow for Report Builder and Report Viewer. The processing sequence occurs in the following phases.

- In phase 1, the user opens one of the subfolders in the Reports folder.

  The Report Builder processes the interface specification of all report procedures associated with the reports in that subfolder and presents the description of each report in the reports pane of the Report Builder. The parameters associated with the first report listed appear in the parameters pane. This processing is done with the command that uses the **-i** option.
- In phase 2, the user selects a report in the reports pane.

  The Report Builder populates the parameters pane with the parameters required for that report. When the user clicks a parameter, the associated parameter chooser prompts the user to provide a value. When all parameters have values, the user can run the report. (The **Run Report** button is not available until all parameters have values.)
- In phase 3, the report is generated.

  A command line, whose parameters are defined in the interface specification, is passed to the Report Viewer, with the parameter values. The Report Viewer runs the report procedure and uses either **cleartool** or the Rational ClearCase Automation Library (CAL) interface to retrieve information from the VOB. The report procedure returns the information to the Report Viewer, which sorts, formats, and displays it. The right-click behavior for all rows in the report (as defined in the interface specification) is now enabled, and the user can also manipulate the report data.

Figure 66 illustrates this processing sequence.

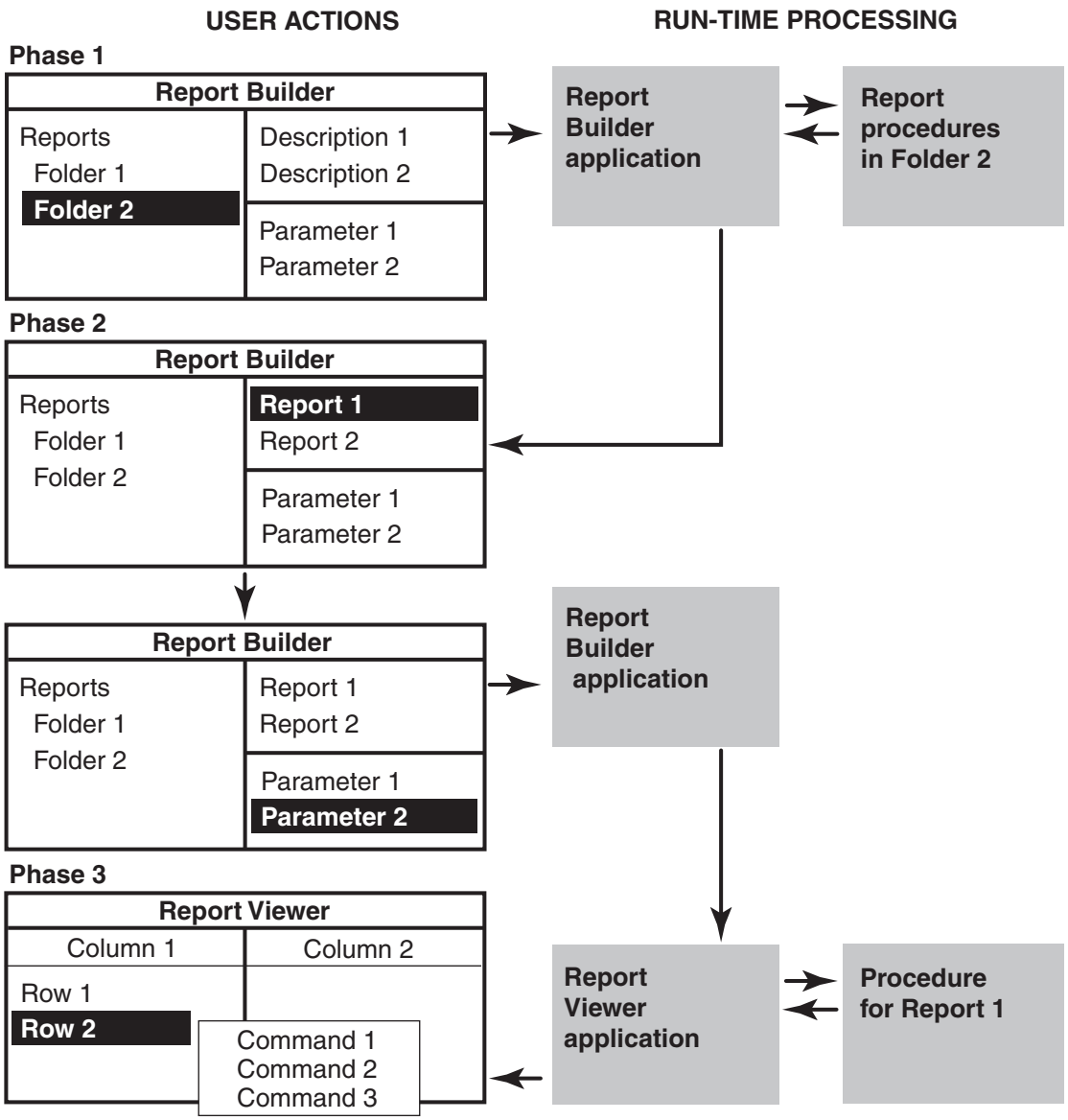


Figure 66. Run-time processing sequence

To execute these processing phases correctly, a report procedure must meet the following requirements:

- The directory that contains the report procedure must be found at known location. The Report Builder reads the \reports\scripts directory to determine the report procedure file names. When a user clicks the associated directory folder, Report Builder calls the associated report procedure.
- The report procedure must have a valid interface specification. If the expected format is not present, the report does not run.
- The interface specification in the report procedure must use parameters and choosers supplied by Rational ClearCase Reports. See Table 9 on page 271.
- The report procedure must support a command line interface that the Report Viewer can use to pass user-defined parameter values to the report procedure.

Configuring shared report directories

When Rational ClearCase is installed on the client, the files for Rational ClearCase Reports are stored in *ccase–home–dir*\reports. Before you modify the contents of this directory, create a copy of it in a shared location. You can then delete or rename folders and add or modify report procedures.

To create the copy, do one of the following:

- Copy the files to a new directory.
- Place a copy of the files under source control and create a Rational ClearCase view to serve as the shared location.

You must remove the .dll and .exe files from the customization directory. The subdirectories for \scripts,\script_tools, and \scripts_rightclick must be present. The \scripts directory becomes the root node **Reports** in the Report Builder tree pane; you can modify this directory tree. Do not delete any files that are in \script_tools and \scripts_rightclick. You may add your own folders, of course.

The help files that are used by the reports cannot be modified and are not included in the \reports directory. The help file for Rational ClearCase Reports is located in *ccase–home–dir*\bin\cc_reports.hlp.

Adding report procedures to source control

To place a copy of *ccase–home–dir*\reports under source control:

1. Copy all files to a temporary directory.
2. In the temporary directory, enter a command of this form:

   ```
   clearfsimport -recurse source-name target-VOB-directory
   ```

3. Create a dynamic or snapshot view for the reports data that is now under source control.

Setting the Report Builder to the customized directory

After you copy the installed files for Rational ClearCase Reports from*ccase–home–dir*\reports to a shared directory location, you can set Report Builder to use this location:

1. In the Report Builder window, click **Report > Set Scripts Location** to open the Configure Reports Directory window.
2. In the window, do one of the following:
   - Type the directory path for the customized directory in the field.
   - Click **...** and, in the Browse for scripts location window, navigate to and select a directory location.

   Tip: After changing the Rational ClearCase Reports user interface, you must restart Report Builder to activate the changes.

Default directory structure for Rational ClearCase Reports

All files for Rational ClearCase Reports are stored in *ccase–home–dir*\reports. This is the directory structure:

```
reports\
   ccreportbuilder.exe
   ccreportviewer.exe
   cctypechooser.dll
   ccpathchooser.dll
   scripts\
      ClearCase_Tools\
      Elements\
```

```
      Attributes\
      Branches\
      Labels\
      Triggers\
   UCM_Projects\
   UCM_Streams\
   Views\
   VOBs\
scripts_rightclick\
script_tools\
```

Populating the Report Builder tree pane

The Report Builder window contains the following panes:

- The tree pane (left).
- The reports pane (top-right).
- The parameter pane (bottom-right).

When the user clicks any folder in the tree pane, the Report Builder runs the associated report procedures from the command line. The **-i** option in the command line enables the Report Builder to use a discovery algorithm to collect the user interface information for Report Builder.

The Report Builder accesses the \scripts subdirectory. Directories in the tree appear as folders in the tree pane. Any files whose extensions match those listed below are listed in the reports pane.

.exe	Typically a Visual C++ application that uses Rational ClearCase Automation Library (CAL) to extract data
.js	JavaScript™, run under Windows Scripting Host (cscript.exe)
.pl	Perl, executed under perl.exe from user's PATH environment variable, for example, ActiveState Perl
.prl	ccperl
.vbs	VBScript, run under Windows Scripting Host (cscript.exe)

All other files are ignored. The file name extension of report procedures supplied with Rational ClearCase Reports is .prl, which the Report Builder associates with ccperl.exe.

At run time, the Report Builder displays all folder names, substituting a space for the underscore and dropping the file name extension. There is one exception: the root directory is always named Reports. This convention cannot be changed.

For example, you have the following on-disk directory tree:

```
scripts\
   Admin_Reports
      view_aging.prl
      all_views.prl
   UCM_Reports\
      lagging_streams.prl
      completed_acts.prl
```

The Report Builder displays text in the tree pane as the following folders:

```
\Reports
   Admin Reports\
   UCM Reports\
```

Report Procedure interface specifications

As the Report Builder finds report procedures in the customized directory, it queries each report procedure for its interface specification. Report Builder starts a separate process with **CreateProcess()**. A valid report procedure must implement an interface specification and return formatted text to STDOUT that conforms to this specification:

```
description : ["<text to display in description pane for this report>"]

id : <numeric help id>

helpfile : ["<full path to user-written help file for what's this
report help>"]

parameters : [<parameter_spec_1>] [<parameter_spec_2>] ...
[<parameter_spec_N>]

rightclick : [<rightclick_spec_1>] [<rightclick_spec_2>] ...
[<rightclick_spec_N>]

fields : [<field_spec_1>] [<field_spec_2> ... [<field_spec_N>]
```

If a serious parsing error occurs in processing the interface specification, the report does not appear in the reports pane. The **helpfile** specification is reserved for future use and is not supported in this release. For information on troubleshooting parsing errors, see "Troubleshooting customization" on page 292.

The examples in "Interface specification for All_Views.prl" through "Parameter choosers" on page 274 show how the interface specification is defined in specific report procedures.

Interface specification for All_Views.prl

The Report Builder uses this command to run All_Views.prl:

```
ccperl "D:\Program Files\Rational\Clearcase\Reports\scripts\Views\All_Views.prl" -i
```

This is the interface specification:

```
description : "All Views"
id : 2001
helpfile :
parameters :
rightclick : Properties_of_View(single)
fields : "View Tag"(view_tag, rightclick, initial_width 30, sort 1)
"View Owner"(user_dq)
```

The report interface attaches the Report Viewer to the **View Tag** and **View Owner** fields; the right-click event in the Report Viewer window calls Properties _of_View.prl, which is based on a data stream from the **View Tag** field.

Description specification

The **description** is the only required part of an interface specification. When only **description** is defined, a report procedure can run other graphical user interfaces (for example, **clearprompt**) or otherwise interact with the user. The reports in the \ClearCase_Tools folder define **description** only.

Descriptions can contain anything other than the delimiter, a double quote ("). There is no maximum length for this definition, but long strings do not wrap in the reports pane.

Help files

Help files are supplied for the Report Builder user interface. The help files for Rational ClearCase Reports are in *ccase–home–dir*\bin\cc_reports.hlp. You cannot add an ID for your own report.

Parameters specification

When specifying parameters, you can use only those supplied with Rational ClearCase Reports. Each parameter has an associated chooser control, parameter text, and a help ID (see Table 9).

Table 9. Parameters supplied with Rational ClearCase Reports

Parameter	Default text displayed in the parameter pane	Help ID	Chooser	Selection
PROJECTS	Select projects in UCM Process VOB	1	Path (UCM)	Multiple
STREAMS	Select streams in UCM Process VOB	2	Path (UCM)	Multiple
ACTIVITIES	Select activities in UCM Process VOB	3	Path (UCM)	Multiple
PROJECT	Select project in UCM Process VOB	4	Path (UCM)	Single
STREAM	Select stream in UCM Process VOB	5	Path (UCM)	Single
ACTIVITY	Select activity in UCM Process VOB	6	Path (UCM)	Single
ISTREAM	Select Integration Stream in UCM Process VOB	7	Path (UCM)	Single
PVOB	Select one Process VOB Tag	8	Path (file selection)	Single
COMPONENT	Type a single UCM component object selector (no verification performed)	9	Text	Single
BASELEVEL	Type a single UCM baseline object selector (no verification performed)	10	Text	Single
ISTREAMS	Select Integration Streams in UCM Process VOB	11	Path (UCM)	Multiple
PVOBS	Select Process VOBs Tags	12	Path (file selection)	Multiple
COMPONENTS	Type a list of UCM components object selectors (no verification performed)	13	Text	Multiple
BASELEVELS	Type a list of UCM baselines object selectors (no verification performed)	14	Text	Multiple
LOOKIN	Select paths in view to report on	15	Path (file selection)	Multiple
USER	Associated with user (values are non-domain-qualified)	17	Text	Single
GROUP	Associated with group (values are non-domain-qualified)	18	Text	Single
LABEL	With label	19	Type	Single
ATTRIBUTE	With attribute	20	Type	Single

Table 9. Parameters supplied with Rational ClearCase Reports (continued)

Parameter	Default text displayed in the parameter pane	Help ID	Chooser	Selection
ATTRIBUTE_VALUE	With value for attribute	21	Text	Single
TRIGGER	With trigger	22	Type	Single
BRANCH	With branch	23	Type	Single
ELTYPE	With element type	24	Text	Single
HLTYPE	With hyperlink type	25	Type	Single
CCTIME	Since date/time	26	Date/time	Single
BRANCHLEVELS	With integer levels of branching	27	Text	Single
FILE_NAME	With filename	28	Text	Single
PATH	Enter path	29	Text	Single
STRING	With string	30	Text	Single
INTEGER	Enter integer	31	Text	Single
REGULAR_ EXPRESSION	Enter regular expression	32	Text	Single

When you use one of the parameters that is listed in Table 9, naming it is all that is required. For example, this is the **parameters** specification for the Elements Changed Between Two Labels report:

```
parameters : LOOKIN LABEL LABEL
```

The order of parameters is important. They are displayed in the parameter pane in the order of the specification. (Each parameter appears as a link. When users click the link, they are prompted to enter a parameter value.) At run time, the Report Viewer calls the report procedure, which must handle the parameter values in the same order as defined in the specification.

The parameters in Table 9 that are associated with the Type Chooser must also include the **LOOKIN** parameter in the interface specification. The **LOOKIN** parameter must have a value before any values for other parameters that use the Type Chooser can be specified. The paths that are the values for the **LOOKIN** parameter are used to build the set of VOBs that types can be read from. At run time, if a user attempts to set a type parameter in reverse order, the Report Builder displays this error message:

```
Before this parameter can be set, you must first set a value for the
"Select pathnames in view to report on" parameter.
```

Rightclick specification

The **rightclick** specification is a list of commands available on the pop-up menu in the Report Viewer. All right-click events are supported by a list of scripts in the \scripts_rightclick directory. This specification allows you to control the text on the pop-up menu. At run time, underscores in these text strings are replaced by spaces.

```
rightclick : properties_of_view delete_view
```

By default, the commands are valid for both single and multiple selections of result records in the Report Viewer. This behavior can be controlled by using the **single** modifier:

```
rightclick : properties_of_view(single) delete_view(single)
```

A special string, **sep**, allows visual separators to group commands. At run time, these commands appear on the pop-up menu in the order specified.

Fields specification

The **fields** specification defines the names of the field headings and a number of modifiers to describe the results a report procedure returns to the Report Viewer. Table 10 describes the supported modifiers.

Table 10. Fields modifiers

Modifier	Description
sort N	Optional. Specifies the sort order for returned records. If specified, this modifier must be a sequence of integers that begin with 1. If no sort specification is made, the records remain in the same order as returned from the report procedure.
Inital_width N	Optional. Overrides the default width for the field.
<field_type>	Required.
hidden	Optional. Prevents display of values for this field in the Report Viewer. If this modifier is used, there is usually an associated **sort N** modifier for the field.
rightclick	Optional. The field value stream that is sent where any right-click action occurs in the Report Viewer. Only one field can be designated as the **rightclick** field.

For example, the following **fields** specification describes a single field with the minimum specification allowed. The **field_type** modifier is required.

```
fields: "view tag"(view_tag)
```

In this example, the **fields** specification defines two fields, **view tag** and **last mod time**, with all the allowable modifiers:

```
fields: "view tag"(view_tag, rightclick, initial_width 10) "last mod
time"(time_t, hidden, sort 1)
```

field_type conventions

Table 11 lists the names for **field_type**s and the kind of data represented. Use these definitions in your own report procedures wherever possible; but you can use your own definitions.

Table 11. Field type supplied with Rational ClearCase Reports

Field name	Data description	Example
project	UCM Project headline name	V4.1
project_objsel	UCM project object selector	Project:v4.1@\projects
stream	UCM Stream headline name	George_v4.1
stream_objsel	UCM Stream object selector	George_v4.1@\projects
activity	UCM Activity headline name	My activity
activity_objsel	UCM Activity object selector	Activity:my_act@\projects

Table 11. Field type supplied with Rational ClearCase Reports (continued)

Field name	Data description	Example
view_tag	View-tag such as returned by **lsview**	main_latest_view
time_t	Integer ticks since 1/1/1970	946934277
cctime	Readable time, format is `%dfmt_ccase`	20-Dec-99.16:01:12
User	User name	georgem
User_dq	Domain-qualified user name	rational\georgem
string	Random text	hello world
Host	Host name	georgemnt
Hpath	Local machine path to view/VOB directory	D:\ClearCase_Storage\views\jet
View_sttrs	View attributes	snapshot, ucmview
Element_xpn	Full path to element ending in @@	S:\frontpage\accts\web\photo.htm@@
Element_pn	Full path to element without @@	S:\frontpage\accts\web\photo.htm
Version_pn	Version specifier, after @@	\main\v4.0.bl5_main\2
label	Label instance name	V4.0
Integer	Integer number	5
Yes_no	**yes** or **no** enumerated string	Yes
Branch_xpn	Full path to branch	S:\frontpage\accts\web\photo.htm@@\main
version_xpn	Full path to version	S:\frontpage\accts\web\photo.htm@@\main\3
branch	Branch name	main
Attribute	Attribute name	normalize_html
Objsel	Object selector	VOB:\my_vob
Trigger	Trigger name	post_ci
Eltype	Element type	text_file
Vob_tag	VOB Tag	\projects

Depending on the column width that is required to display for a user-defined **field_type**, the **fields** specification in a report procedure may need to adjust the display column size with the **Inital_width N** modifier.

Parameter choosers

When a user opens a folder in the Report Builder tree pane, the reports pane is populated with the list of descriptions that the Report Builder discovered in the interface specification. When the user selects a report, the associated parameters are loaded in the Report Builder. Each parameter in the interface specification has associated parameter text, a help ID, and a chooser. All parameters have an associated chooser (Table 9).

These choosers are supplied with Rational ClearCase Reports:

- Path Chooser
- UCM Targets Chooser
- Types Chooser
- Date/Time Chooser
- Text Chooser

For user information, click **Help** in any chooser window.

Path chooser

The Path Chooser is associated with the **LOOKIN** parameter. It presents a list of view paths for users to select, and then sends the selected paths to the report procedure. It is also used for the **PVOB** and **PVOBS** parameters to choose the VOB tag of a UCM project VOB.

UCM targets chooser

The UCM Targets Chooser is associated with the **PROJECT**, **PROJECTS**, **STREAM**, **STREAMS**, **ACTIVITY**, **ACTIVITIES**, **ISTREAM**, and **ISTREAMS** parameters and allows you to select UCM objects.

Type chooser

The Type Chooser presents values for the **BRANCH**, **ATTRIBUTE**, **LABEL**, **HYPERLINK**, and **TRIGGER** parameters. All parameters that the Type Chooser supports require an initial value **LOOKIN** parameter.

Date/time chooser

The Date/Time Chooser is used to select date/time values for the **CCTIME** parameter.

Text chooser

The Text Chooser presents values for these parameters: **COMPONENT**, **COMPONENTS**, **BASELINE**, **BASELINES**, **USER**, **GROUP**, **ATTRIBUTE_VALUE**, **ELTYPE**, **BRANCHLEVELS**, **FILE_NAME**, **PATH**, **STRING**, **INTEGER**, and **REGULAR_EXPRESSION**.

Data typed into the Text Chooser is not validated or parsed in any way by the Report Builder or Report Viewer. The report procedure that accepts the parameter value must perform any validation required.

For most parameters, the text above the field is **Enter value for user**. For parameters that require the name of a baseline, a component, or an element type, the text changes to reflect the parameter. For example: **Enter value for baseline**.

BASELINE baseline:<bl>@ \<pvob>

COMPONENT
component:<comp>@\<pvob>

ELTYPE <text_file>

Viewing the report

When all required parameters have values, clicking **Run Report** opens the Report Viewer window. The Report Builder creates a command line to pass the user-defined parameters, in the order defined by interface specification. For example, if a report procedure asks for parameters **LOOKIN LABEL**, the Report Viewer passes these values as follows:

```
ccperl elements_with_label.prl %LOOKIN='s:\frontpage\acctst';%LABEL=V4.0;
```

The Report Viewer creates a process to run the report procedure using ccperl.exe for .prl, perl for .pl, cscript.exe for .js and .vbs, and default activation for .exe. The report procedure returns results to STDOUT. The results are separated by semicolons, in the same order, number, and type specified in the **fields** definition in the interface specification.

When the report procedure has collected all its data, it exits. The report procedure must return records to STDOUT in the most efficient manner possible; the Report Viewer sorts the results and formats them for display. At run time, users can change the default sorting order by clicking the column headings in the Report Viewer. Simple text sorting is used for all fields except those whose **field_type** is **time_t**, **integer**, or **cctime**. For these three fields only, Report Viewer uses numeric sorting.

Saving report data

Clicking **Save As** in the Report Viewer window opens a standard file selection window to prompt the user to save the results in one of the following output formats:

.CSV	Comma-separated, for import into Access or Excel
.HTML	For viewing in a Web browser
.XML	For viewing in Internet Explorer 5 using XSL style sheets

Saving the file is performed by the save_results.prl script in \script_tools. This script supports two switches, **-html** and **-csv**, and the header, followed by semicolon-separated data rows. This script also needs a *path* value for the **-out** option, where *path* is the value that the Report Viewer passes from the Path Chooser.

XML output is supported directly by the Report Viewer. You can re-implement the **.CSV** and **.HTML** output by modifying save_result.prl. You can also define additional XSL style sheets that can be referred to in XML output. Start with the style sheet supplied with Rational ClearCase Reports (\script_tools\table.xsl) to create customized XSL files.

Report programming examples

All report procedures supplied with Rational ClearCase Reports are written in ccperl. The programming examples presented in this section are modifications of these report procedures. Report procedures can be written in many other scripts and programming languages; report procedures that use other programming languages are available in the **T0046** package that you can obtain from the Rational ClearCase Customer Web site at IBM Rational Support (see "Obtaining the T0046 package" on page 293). The following programming examples are presented in this section:

- Example 1: Adding a new column to the report for Versions_byDate.prl.
- Example 2: Changing the directory organization and report description, modifying the version path to a use different field name, and adding an element type column to report output for Elements_with_New_Versions_Since_Date.prl.
- Example 3: Changing the report description, parameter types, and report output for Elements_Created_by_User.prl.
- Example 4: Changing the order of commands and adding a command to the pop-up menu for Element_with_Labels.prl.

- Example 5: Adding a user-defined command to the pop-up menu for Element_with_Branches.prl.

In the source code listings that accompany each example, the string `### customization change` marks the changes to the original report that accomplish the task.

Example 1: Adding a column to report output

The Versions by Date report lists all versions that exist for the path that the user specified. This report includes the following columns:

- Version Path
- Version Creation Time

The change to this report adds a column that lists the user name associated with each version. The report procedure is located in *ccase–home–dir*\Reports\Scripts\Elements\Versions_by_Date.prl.

Processing logic

The processing logic of Versions_by_Date.prl is as follows:

1. The **LOOKIN** parameter, which is the sole parameter for this function, is received in a string of this form:

   ```
   LOOKIN = "<path1> [<path2> ...]"
   ```

   This parameter specifies the list of paths with which the **cleartool find** command is to be invoked.
2. When the routine is invoked, it extracts the paths from the **LOOKIN** string and passes them to the **check_lookin()** routine (located in common_script.prl).
3. The routine **check_lookin()** then puts the paths into the global variable **$ctfind_paths** and encloses each path in double quotes; it also performs simple validations on the paths received.
4. The report procedure calls **cleartool lshistory**, passing **$ctfind_paths** as the paths parameter, and with a **-fmt** parameter to return the necessary information.
5. The report procedure executes a **print** statement with parameters (that is, the items to print) of the same number and order as the list passed during interface specification processing. The Report Builder has the information required to set up the column headings; the report procedure must conform to this specification to print its output.

Interface specification

This is the existing interface specification for Versions_by_Date.prl:

```
if (/^-i/) {
   print "description : 'Versions by Date'\n";
   print "id : 2018\n";
   print "helpfile :\n";
   print "parameters : ";
   print "LOOKIN ";
   print "\n";
   print_version_rightclick();
   print "fields : ";
   print "\"Version Path\"(version_xpn, rightclick, sort 2) ";
   print "\"Version Creation Time\"(cctime) ";
   print "\"Version Creation Time\"(time_t, sort 1, hidden) ";
   print "\n";
   exit(0);
}
```

Changes required

To add an additional column of report output:

1. Add a properly coded **print** statement to the interface specification that the Report Builder can pass to the Report Viewer.
2. Add a **%Fu;** to the **-fmt** parameter in the **cleartool lshist** call to get this information from the Rational ClearCase configuration.
3. Properly extract the user information into some variable after the **cleartool lshist** call returns its output, so that it can be printed.
4. Print the user variable in the same order as it appeared in the interface specification so that it appears under the correct column heading.

Modified report procedure

Here is the modified version of Versions_by_Date.prl. This report procedure is example1.prl in the**T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
$start_dir = $0; $start_dir =~ s/\\scripts\\.*/\\scripts/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts/$1\\script_tools/;

$cc = ""; if ($cc) {;};
$ct = ""; if ($ct) {;};
$debug = ""; if ($debug) {;};
$skip_path_checks = ""; if ($skip_path_checks) {;};
$CLEARCASE_XN_SFX = ""; if ($CLEARCASE_XN_SFX) {;};
$ctfind_paths = ""; if ($ctfind_paths) {;};
$skip_path_checks = "yes"; if ($skip_path_checks) {;};
$debug = "no"; if ($debug) {;};

sub do_exit {
   $err = join(" ", @_);
   if ("$err" != "") {
      print STDERR "$err\n";
   }
   sleep(2);
   if ("$err" != "") {
      exit(1);
      } else {
      exit(0);
   }
}

open(INCLUDE, "<$common_dir\\common_script.prl") or do_exit("error
opening include file '$common_dir\\common.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common.prl'");

my $args = $ARGV[0];
$args =~ s/%/ /g;
@args = split(";", $args);
$required_args = 0;
foreach(@args) {

   s/^[    ]+//;
   s/[    ]+$//;
   validate_arg_length($_);
   if (/^-i/) {
      print "description : 'Versions by Date'\n";
      print "id : 2018\n";
```

```
      print "helpfile :\n";
      print "parameters : ";
      print "LOOKIN ";
      print "\n";
      print_version_rightclick();
      print "fields : ";
         print "\"Version Path\"(version_xpn, rightclick, sort 2) ";
         print "\"Version Creation Time\"(cctime) ";
         print "\"Version Creation Time\"(time_t, sort 1, hidden) ";
### customization change *** added following line
         print "\"User'(user) ";
      print "\n";
      exit(0);
}
if (/^LOOKIN[    ]*=[    ]*('.*')/) {
   check_lookin($1);
   $required_args++;
   next;
}
print STDERR "unrecognized argument: $_\n";
print STDERR "  ccperl $0 -i\n";
print STDERR "    for script's interface.\n";
do_exit("\n");
}
if ($required_args != 1) {
   print STDERR "usage: not all required arguments specified.\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
   do_exit("\n");
}
$ENV{"d;"} = "äd;ä";
open(CTHIST, "cleartool lshist -fmt '%d;%e;%n\\n' -recurse -nco
$ctfind_paths |");
while(<CTHIST>) {
   chomp;
   if (/create directory version/ || /create version/) {
      ($date, $event, $xpn) = split  /;/, $_, 3;
      if ($date) {;}
      if ($event) {;}
      if ($xpn) {;}
      $timet = time_to_ticks($date);
### customization change *** added following line
      $user = 'cleartool desc -fmt '%Fu' '$xpn'';
### customization change *** added ";$user" to following line
      print "$xpn;$date;$timet;$user\n";
   }
}
do_exit();
```

Example 2: changing directory organization, description, and output

The Elements with New Versions Since Date report lists all new versions for the path and since the date and time specified by the user. This report includes the following columns:

- Element Path
- Version Path
- Version Creation Time

The changes to the report procedure do the following:

- Display in the Report Builder tree pane a new directory named *ccase–home–dir*\Reports\Scripts\Elements\New_Versions directory.

- Display a new report description: **Types of Elements with New Versions Since Date**.
- Display the version path information in the **version_xpn** field in a different format.
- Add a column in the report output to display a new column for **Element Type**.

The report procedure is located in:

ccase-home-dir\Reports\Scripts\Elements\Elements_with_New_Versions_Since_Date.pr

Processing logic

The processing logic of Elements_with_New_Versions_Since_Date.prl is as follows:

1. When the Report Builder processes the interface specification, the report procedure yields two parameters:

   ```
   LOOKIN
   CCTIME
   ```

   The mechanics of the **LOOKIN** parameter are described in "Example 1: Adding a column to report output" on page 277. When the report procedure receives **CCTIME**, it is a string of this form:

   ```
   CCTIME = "time"
   ```

   This parameter specifies the times that the **cleartool find** command uses.
2. When the report procedure is invoked by the Report Viewer using a fully qualified command line, it extracts the values from the **CCTIME** string and passes them to the **chooser_time_to_cctime()** subroutine (located in common.prl). This routine converts the string to the correct format (for passing to **cleartool**) and returns it.
3. The report procedure opens a pipe from a **cleartool find -print** command, with the converted **cctime** value passed in as a **created_since(<cctime>)** string. The value **created_since** is a query_language(1) predicate, which is frequently used in conjunction with the **find** command.
4. As the values from the **cleartool find** command are returned, the report procedure calls **cleartool describe** on the output to get the version-creation time. The routine calls the **time_to_ticks()** routine (in common.prl) to get the time equivalent in ticks.
5. The report procedure gets the path and version ID from the **cleartool find** output, splitting it on the value of the $CLEARCASE_XN_SFX extended naming symbol for the host. Finally, the report procedure prints the information in the same order as defined in the interface specification.

Interface specification

This is the existing interface specification for Elements_with_New_Versions_Since_Date.prl:

```
if (/^-i/) {
   print "description : 'Elements with New Versions Since Date'\n";
   print "id : 2017\n";
   print "helpfile :\n";
   print "parameters : ";
   print "LOOKIN CCTIME";
   print "\n";
   print_element_rightclick();
   print "fields : ";
   print "\"Element Path\"(element_pn, sort 2, rightclick) ";
   print "\"Version Path\"(version_pn) ";
   print "\"Version Creation Time\"(cctime) ";
```

```
   print "\"Version Creation Time\"(time_t, hidden, sort 1) ";
   print "\n";
   exit(0);
}
```

Changes required

To change the directory organization and report description, to modify the version path to use a different field name, and to add an element type column to the report output:

1. Create a new folder, **New_Versions**, and move the report procedure there.
2. Add a properly coded **print** statement to the interface specification that does the following:
   - Specifies how to display the report description information in the Report Builder
   - Specifies how to display the report in the Report Viewer
3. Add additional processing to the **cleartool find** output as required to get the desired information for element type.
4. Properly extract the new information for element type into a variable.
5. Print the new information in the proper position so that it appears under the correct column heading.

Modified report procedure

Here is the modified version of Elements_with_New_Versions_Since_Date.prl. This report procedure is example2.prl in the**T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
$start_dir = $0; $start_dir =~ s/\\scripts\\.*/\\scripts/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts/$1\\script_tools/;

$cc = ""; if ($cc) {;};
$ct = ""; if ($ct) {;};
$debug = ""; if ($debug) {;};
$skip_path_checks = ""; if ($skip_path_checks) {;};
$CLEARCASE_XN_SFX = ""; if ($CLEARCASE_XN_SFX) {;};
$ctfind_paths = ""; if ($ctfind_paths) {;};
$skip_path_checks = "yes"; if ($skip_path_checks) {;};
$debug = "no"; if ($debug) {;};

sub do_exit {
   $err = join(" ", @_);
   if ("$err" != "") {
      print STDERR "$err\n";
   }
   sleep(2);
   if ("$err" != "") {
      exit(1);
      } else {
exit(0);
}
}
open(INCLUDE, "<$common_dir\\common_script.prl") or do_exit("error
opening include file '$common_dir\\common.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common.prl'");

my $args = $ARGV[0];
```

```
$args =~ s/%/ /g;
@args = split(";", $args);
my $cctime = "";
$required_args = 0;
foreach(@args) {
   s/^[    ]+//;
   s/[    ]+$//;
   validate_arg_length($_);
   if (/^-i/) {
### customization change *** changed following line
      print "description : 'Types of Elements with New Versions Since
      Date'\n";
      print "id : 2017\n";
      print "helpfile :\n";
      print "parameters : ";
         print "LOOKIN CCTIME";
      print "\n";
      print_element_rightclick();
      print "fields : ";
         print "\"Element Path\"(element_pn, sort 2, rightclick) ";
### customization change *** changed following line
         print "\"Version Path\"(version_xpn) ";
         print "\"Version Creation Time\"(cctime) ";
         print "\"Version Creation Time\"(time_t, hidden, sort 1) ";
### customization change *** added following line
         print "\"Element Type\"(eltype) ";
      print "\n";
      exit(0);
}
   if (/^LOOKIN[    ]*=[    ]*('.*')/) {
      check_lookin($1);
      $required_args++;
      next;
   }
   if (/^CCTIME[    ]*=[    ]*'*([^']*)'*/)  {
   $cctime = chooser_time_to_cctime($1);
   $required_args++;
   next;
   }
   print STDERR "unrecognized argument: $_\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
   do_exit("\n");
}
if ($required_args != 2) {
   print STDERR "usage: not all required arguments specified.\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
   do_exit("\n");
}
open(CTFIND, "cleartool find $ctfind_paths -version
'created_since($cctime)' -print |");
while(<CTFIND>) {
   chomp;
   if (/CHECKEDOUT/) {next;}
   $vertime = 'cleartool desc -fmt '%d' '$_'';
### customization change *** added following line
   $eltype = 'cleartool desc -fmt '%[type]p' '$_'';
   $vertime_t = time_to_ticks($vertime);
   ($path, $verid) = split $CLEARCASE_XN_SFX, $_, 2;
### customization change *** changed following line
   print "$_;$verid;$vertime;$vertime_t;$eltype\n";
   #print "$path;$verid;$vertime;$vertime_t\n";
}
do_exit();
```

Example 3: changing description, parameter types, and output

The Elements Created by User report lists all elements created by the user-defined user name. This report includes the following columns:

- Element Path
- Creating User

The changes to this report do the following:

- Display a new report description: **Elements with Group**.
- Remove the existing user parameter and add a new parameters for group.
- Compare the group associated with an element and the group specified in a user-defined group parameter.
- Add a column in the report output for **Group** and **Yes/No**. The **Yes/No** column will reflect the result of the comparing whether the group associated with each element is the same as the value of the user-defined group parameter.

The script is located in *ccase–home–dir*\Reports\Scripts\Elements\Elements_Created_by_User.prl.

Processing logic

The processing logic of Elements_Created_by_User.prl is as follows:

1. When the Report Builder processes the interface specification, the report procedure yields two parameters:

   ```
   LOOKIN
   USER
   ```

   The mechanics of the **LOOKIN** parameter are described in *"Example 1: Adding a column to report output" on page 277*. The report procedure receives **USER** as a string of this form:

   ```
   USER= "user-name"
   ```

   This parameter specifies the user name that the **cleartool** subcommand uses.
2. The **USER** string is extracted and stored as **$ccuser**. It is then passed to the **created_by($ccuser)**.
3. The **created_by ($ccuser)** query language primitive filters the paths specified to **cleartool find** and returns only those that match the predicate, in this case, those created by the user by setting a parameter value for **USER**.
4. The user variable is printed in the same order specified in the interface specification so that it appears under the correct column heading.

Interface specification

This is the existing interface specification for Elements_Created_by_User.prl:

```
if (/^-i/) {
   print "description : 'Elements Created by User'\n";
   print "id : 2016\n";
   print "helpfile :\n";
   print "parameters : ";
   print "LOOKIN USER";
   print "\n";
   print_element_rightclick();
   print "fields : ";
   print "\"Element Path\"(element_xpn, sort 2, rightclick) ";
   print "\"Creating User\"(user, sort 1) ";
   print "\n";
   exit(0);
}
```

Changes required

To remove the user parameter, to add parameters for group and date/time, and to adjust the report output for group and date/time information:

1. Change the interface specification of the report procedure to correspond to required interface changes.
2. Change the logic in the report procedure to handle data requests for group information; add a **%Gu;** to the **-fmt** parameter in the**cleartool describe** call to get group information from the Rational ClearCase configuration.
3. Properly extract the group information into a variable after the **cleartool describe** call returns its output, so that it can be printed.
4. Determine whether the element group is the same group parameter value entered by the user and print the result of this comparison as a column heading.
5. Print the group variables in the order specified in the interface specification so that they appear under the correct column heading.

Modified report procedure

Here is the modified version of Elements_Created_by_User.prl. This report procedure is example3.prl in the**T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
$start_dir = $0; $start_dir =~ s/\\scripts\\.*/\\scripts/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts/$1\\script_tools/;

$cc = ""; if ($cc) {;};
$ct = ""; if ($ct) {;};
$debug = ""; if ($debug) {;};
$skip_path_checks = ""; if ($skip_path_checks) {;};
$CLEARCASE_XN_SFX = ""; if ($CLEARCASE_XN_SFX) {;};
$ctfind_paths = ""; if ($ctfind_paths) {;};
$skip_path_checks = "yes"; if ($skip_path_checks) {;};
$debug = "no"; if ($debug) {;};

sub do_exit {
   $err = join(" ", @_);
   if ("$err" != "") {
   print STDERR "$err\n";
   }
   sleep(2);
   if ("$err" != "") {
      exit(1);
   } else {
   exit(0);
   }
}
open(INCLUDE, "<$common_dir\\common_script.prl") or do_exit("error
opening include file '$common_dir\\common.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common.prl'");

my $args = $ARGV[0];
$args =~ s/%/ /g;
@args = split(";", $args);
my $ccuser = "";
$required_args = 0;
foreach(@args) {
```

```
   s/^[    ]+//;
   s/[    ]+$//;
   validate_arg_length($_);
   if (/^-i/) {
### customization change *** changed following line
      print "description : 'Elements With Group'\n";
      print "id : 2016\n";
      print "helpfile :\n";
      print "parameters : ";
### customization change *** changed following line
         print "LOOKIN GROUP";
      print "\n";
      print_element_rightclick();
      print "fields : ";
         print "\"Element Path\"(element_xpn, sort 2, rightclick) ";
### customization change *** added following 2 lines
         print "\"Element's Group\"(group, sort 1) ";
         print "\"Same\"(yes_no) ";
### customization change *** deleted following line
         #print "\"Creating User\"(user, sort 1) ";
      print "\n";
   exit(0);
}
if (/^LOOKIN[    ]*=[    ]*('.*')/) {
   check_lookin($1);
   $required_args++;
   next;
}
### customization change *** deleted following 2 lines
   #if (/^USER[    ]*=[\t   ]*\"*([^\"]*)\"*/) {
      #$ccuser = $1;
### customization change *** added following 2 lines
if (/^GROUP[    ]*=[\t   ]*\"*([^\"]*)\"*/)  {
   $ccgroup = $1;
   $required_args++;
### customization change *** deleted following line
   #validate_user($ccuser);
   next;
}
print STDERR "unrecognized argument: $_\n";
print STDERR "  ccperl $0 -i\n";
print STDERR "    for script's interface.\n";
do_exit("\n");
}
if ($required_args != 2) {
   print STDERR "usage: not all required arguments specified.\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
   do_exit("\n");
}
### customization change *** deleted following 3 lines
#if ($ccuser =~ /[ ]+/) {
#     do_clearprompt("cleartool find does not allow spaces in user names;
#   cannot proceed.");
#}

### customization change *** changed following line
open(CTFIND, "cleartool find $ctfind_paths -nxname -print |");
while(<CTFIND>) {
   chomp;
### customization change *** added following 6 lines
   $grp = 'cleartool desc -fmt '%Gu' '$_'';
   if ($grp eq $ccgroup) {
      $same = "yes";
      } else {
      $same = "no";
```

```
   }
### customization change *** changed following line
   print "$_;$grp;$same\n";
   #print "$_;$ccuser;\n";
}
do_exit();
```

Example 4: changing the pop-up menu for right-click handling

The Elements with Labels report lists all elements with labels for a user-defined path. This report includes one column:

- Element Path

The change to this report adds the **Compare with Previous Version** command to the pop-up menu. Currently, these commands appear on the pop-up menu:

- Properties of Element
- Version Tree
- History

The report procedure is located in *ccase–home–dir*\Reports\Scripts\Elements\Labels\Elements_with_Labels.prl.

Interface specification

This is the existing interface specification for Elements_with_Labels.prl:

```
if (/^-i/) {
   print "description : ";
   print "'Elements with Labels'";
   print "\n";
   print "id : 2003\n";
   print "helpfile :\n";
   print "parameters : ";
   print "LOOKIN ";
   print "LABEL ";
   print "\n";
   print_element_rightclick();
   print "fields : ";
   print "\"Element Path\"(element_pn, rightclick, sort 1)";
   print "\n";
   exit(0);
}
```

Note the call to **print_element_rightclick()** in the middle of the interface specification. The code for this routine is located in \script_tools\common.prl:

```
sub print_element_rightclick {
   print "rightclick : ";
   print "Properties_of_Element(single) ";
   print "sep ";
   print "Version_Tree(single) ";
   print "History(single) ";
   print "\n";
}
```

Changes required

A convention used in the report procedures is to put the same commands on pop-up menus for all reports that use the same primary sort field. For example, all the reports whose primary sort key is **element** or **element_xpn** display the same set of commands.

To make an additional command available for all reports whose primary sort key is **element** or **element_xpn**, modify the routines stored in \script_rightclick and then edit the associated routine in \script_tools\common.prl.

To change the report procedure, copy the contents of **sub print_element_rightclick** (located in \script_tools\common.prl) and paste it into the appropriate part of the interface specification. Then, add a declaration to display the new command.

Modified report procedure

Here is the modified version of Elements_with_Labels.prl. This report procedure is example4.prl in the**T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
$start_dir = $0; $start_dir =~ s/\\scripts\\.*/\\scripts/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts/$1\\script_tools/;

$cc = ""; if ($cc) {;};
$ct = ""; if ($ct) {;};
$debug = ""; if ($debug) {;};
$skip_path_checks = ""; if ($skip_path_checks) {;};
$CLEARCASE_XN_SFX = ""; if ($CLEARCASE_XN_SFX) {;};
$ctfind_paths = ""; if ($ctfind_paths) {;};
$skip_path_checks = "yes"; if ($skip_path_checks) {;};
$debug = "no"; if ($debug) {;};

sub do_exit {
   $err = join(" ", @_);
   if ("$err" != "") {
      print STDERR "$err\n";
   }
   sleep(2);
   if ("$err" != "") {
      exit(1);
      } else {
      exit(0);
   }
}
open(INCLUDE, "<$common_dir\\common_script.prl") or do_exit("error
opening include file '$common_dir\\common.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common.prl'");

my $args = $ARGV[0];
$args =~ s/%/ /g;
@args = split(";", $args);
my $cclabel = "";
$required_args = 0;
foreach(@args) {
   s/^[    ]+//;
   s/[    ]+$//;
   validate_arg_length($_);
   if (/^-i/) {
      print "description : ";
         print "'Elements with Labels'";
      print "\n";
      print "id : 2003\n";
      print "helpfile :\n";
      print "parameters : ";
         print "LOOKIN ";
         print "LABEL ";
```

```
      print "\n";
### customization change *** deleted following line
   #print_element_rightclick();
### customization change *** added following 7 lines
      print "rightclick : ";
         print "Properties_of_Element(single) ";
         print "sep ";
         print "Compare_with_Previous_Version(single) ";
         print "Version_Tree(single) ";
         print "History(single) ";
      print "\n";
      print "fields : ";
         print "\"Element Path\"(element_pn, rightclick, sort 1)";
      print "\n";
   exit(0);
}
if (/^LOOKIN[    ]*=[    ]*('.*')/) {
   #print "paths are $1\n";
   check_lookin($1);
   $required_args++;
   next;
}
if (/^LABEL[    ]*=[    ]*'*([^']*)'*/)  {
   $cclabel = $1;
   #print "label is $cclabel\n";
   $required_args++;
   next;
}
print STDERR "unrecognized argument: $_\n";
print STDERR "  ccperl $0 -i\n";
print STDERR "    for script's interface.\n";
do_exit("\n");
}
if ($required_args != 2) {
   print STDERR "usage: not all required arguments specified.\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
   do_exit("\n");

}
open(CTFIND, "cleartool find $ctfind_paths -element
'lbtype_sub($cclabel)' -print |");
while(<CTFIND>) {
   chomp;
   ($path, $rest) = split $CLEARCASE_XN_SFX, $_, 2;
   if ($rest) {;}
   print "$path;\n";
}
do_exit();
```

Example 5: adding a new command to Report Viewer pop-up menu

The Elements with Branches report lists all elements associated with a branch and path that the user provides. This report includes the following columns:

- Element Path
- Branch

The report procedure is located in *ccase–home–dir*\Reports\Scripts\Elements\Branches\Elements_with_Branches.prl.

The change to this report adds the **Merge Manager** command to the pop-up menu. This command is not supplied with Rational ClearCase Reports, so the work

required to included it is different from that described in "Example 4: changing the pop-up menu for right-click handling" on page 286.

These commands currently appear on the pop-up menu:

- Properties of Element
- Version Tree
- History

Interface specification

This is the existing interface specification for Elements_with_Branches.prl:

```
if (/^-i/) {
   print "description : 'Elements with Branches'\n";
   print "id : 2013\n";
   print "helpfile :\n";
   print "parameters : ";
   print "LOOKIN BRANCH";
   print "\n";
   print_element_rightclick();
   print "fields : ";
   print "\"Element Path\"(element_xpn, sort 1, rightclick) ";
   print "\"Branch\"(branch) ";
   print "\n";
   exit(0);
}
```

Changes required

Making this modification requires a new script for the new command functions.

You must place this script in the \scripts_rightclick directory. (The script can be written in any of the supported programming languages.) The script must be coded to receive a stream on input from STDIN from a field that is designated by a **rightclick** modifier in the interface specification of the report procedure. For example, to create my_rc.prl, which starts clearmrgman.exe (Merge Manager), you must place my_rc.prl in \scripts_rightclick.

Modified report procedure

Here is the modified version of Elements_with_Branches.prl. This report procedure is example5.prl in the**T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
$start_dir = $0; $start_dir =~ s/\\scripts\\.*/\\scripts/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts/$1\\script_tools/;

$cc = ""; if ($cc) {;};
$ct = ""; if ($ct) {;};
$debug = ""; if ($debug) {;};
$skip_path_checks = ""; if ($skip_path_checks) {;};
$CLEARCASE_XN_SFX = ""; if ($CLEARCASE_XN_SFX) {;};
$ctfind_paths = ""; if ($ctfind_paths) {;};
$skip_path_checks = "yes"; if ($skip_path_checks) {;};
$debug = "no"; if ($debug) {;};

sub do_exit {
   $err = join(" ", @_);
   if ("$err" != "") {
      print STDERR "$err\n";
   }
   sleep(2);
   if ("$err" != "") {
      exit(1);
      } else {
```

```
   exit(0);
   }
}
open(INCLUDE, "<$common_dir\\common_script.prl") or do_exit("error
opening include file '$common_dir\\common.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common.prl'");

my $args = $ARGV[0];
$args =~ s/%/ /g;
@args = split(";", $args);
my $ccbranch = "";
$required_args = 0;
foreach(@args) {
   s/^[    ]+//;
   s/[    ]+$//;
   validate_arg_length($_);
   if (/^-i/) {
      print "description : 'Elements with Branches'\n";
      print "id : 2013\n";
      print "helpfile :\n";
      print "parameters : ";
         print "LOOKIN BRANCH";
      print "\n";
### customization change *** deleted following line
   #print_element_rightclick();
### customization change *** added following 8 lines
      print "rightclick : ";
         print "my_rc(single) ";
         print "Properties_of_Element(single) ";
         print "sep ";
         print "Compare_with_Previous_Version(single) ";
         print "Version_Tree(single) ";
         print "History(single) ";
      print "\n";
      print "fields : ";
         print "\"Element Path\"(element_xpn, sort 1, rightclick) ";
         print "\"Branch\"(branch) ";
      print "\n";
   exit(0);
}

if (/^LOOKIN[    ]*=[    ]*('.*')/) {
   #print "paths are $1\n";
   check_lookin($1);
   $required_args++;
   next;
}
if (/^BRANCH[    ]*=[    ]*'*([^']*)'*/)  {
   $ccbranch = $1;
   $required_args++;
   next;
}
print STDERR "unrecognized argument: $_\n";
print STDERR "  ccperl $0 -i\n";
print STDERR "    for script's interface.\n";
do_exit("\n");
}
if ($required_args != 2) {
   print STDERR "usage: not all required arguments specified.\n";
   print STDERR "  ccperl $0 -i\n";
   print STDERR "    for script's interface.\n";
```

```
   do_exit("\n");
}
open(CTFIND, "cleartool find $ctfind_paths -nxname -branch
'brtype($ccbranch)' -print |");
while(<CTFIND>) {
   chomp;
   print "$_;$ccbranch;\n";
}
do_exit();
```

Here is the new command of my_rc.prl that has been created to support a new pop-up menu command for starting Merge Manager. This report procedure is available in the **T0046** package, which is available at IBM Rational Support (see "Obtaining the T0046 package" on page 293).

```
# these are all set by set_record_vars in common_rightclick.prl
#
$CLEARCASE_PN = "", $CLEARCASE_XN_SFX = "", $CLEARCASE_ID_STR = "",
$CLEARCASE_XPN = "";
$CLEARCASE_BRANCH_PATH = "", $CLEARCASE_VERSION_NUMBER = "";
$ELEMENT_RESULTS = "", $BRANCH_RESULTS = "", $VERSION_RESULTS = "";
$results = "";

$debug = "no";

$start_dir = $0; $start_dir =~
s/\\scripts_rightclick\\.*/\\scripts_rightclick/;
$common_dir = $start_dir;
$common_dir =~ s/(.*)\\scripts_rightclick/$1\\script_tools/;

open(INCLUDE, "<$common_dir\\common_rightclick.prl") or do_exit("error
opening include file '$common_dir\\common_rightclick.prl'");
$buf = "";
while(<INCLUDE>) {
   $buf = $buf . $_;
}
close(INCLUDE);
eval $buf || do_exit("error on eval of include file
'$common_dir\\common_rightclick.prl'");

if ($CLEARCASE_PN) {;}
if ($CLEARCASE_XN_SFX) {;}
if ($CLEARCASE_ID_STR) {;}
if ($CLEARCASE_XPN) {;}
if ($CLEARCASE_BRANCH_PATH) {;}
if ($CLEARCASE_VERSION_NUMBER) {;}
if ($ELEMENT_RESULTS) {;}
if ($BRANCH_RESULTS) {;}
if ($VERSION_RESULTS) {;}
if ($debug) {;}

$first = "yes";

while(<STDIN>) {
   chomp;
   set_record_vars($_);
#####################################################################
# things to be done a record at a time are done here
   if ($first eq "yes") {
      $first = "no";
      open(COMMAND, "clearmrgman |");
      while(<COMMAND>) {;}
      close(COMMAND);
   }
#####################################################################
}
# things to be done with the result set as a whole go here
```

```
$results =~ s/ $//;

#print "results are $results\n";
```

Troubleshooting customization

There are two primary areas that you may need to troubleshoot:

- Errors in the interface specification
- Coding high-level languages other than ccperl

Errors in the interface specification

These are the common errors you may make when coding the interface specification for your report procedure:

- The interface syntax used in your program does not conform to the interface specification.
- Invalid parameter names are used for the **parameter** specification.
- The **rightclick** specification calls a routine that does not exist in the \right_click folder.
- The **print** statements to STDOUT are in a different order from that defined by the **fields** specification.

You can identify errors in the interface specification easily by using the testing script, ifaces.prl. This script checks customized report procedures that have been written in ccperl. It is available with the **T0046** package (see "Coding high-level languages other than ccperl" on page 293).

To start the testing script, use a command of this form:

ccperl ifaces.prl *<path-to-script-or-directory-tree>*

Test your report procedures *before* you check them in to the shared directory tree that you have configured.

If you do not run the testing script before using your report in Report Builder and a parsing error occurs in processing the interface specification, the new report is not displayed in the list of reports in the reports pane. There is no feedback; you see the report description in the reports pane or you see nothing. If you do not see a description, the parsing error is serious. If you do see a description, the interface specification is somewhat correct, but you may still be using an invalid parameter, referring to a nonexistent right-click routine, or sending output in the wrong order to STDOUT.

The Report Builder does not check for valid parameters. For example, consider the interface specification for a new report procedure, my_custom_report.prl, with the following interface specification:

```
description : "This test report asks for a three known parameters and
two unknown parameters"

id : 2500

parameters : LOOKIN UNKNOWN_1 STREAMS FOO PROJECT

rightclick :

fields : "field 1"(string)
```

The second and fourth parameters of this interface specification are invalid. At run time, the description for this report appears in the Report Builder reports pane, but the second and fourth parameters are displayed as blank lines in the parameter pane.

However, the testing script detects these errors because these parameter names are not supplied with ClearCase Reports (see Table 9):

```
my_custom_report.prl:

desc: this test report asks for a three known parameter and two unknown
parameters

id: 2500

parm: LOOKIN

**********************

ERROR: illegal parameter: UNKNOWN_1

**********************

continue? (y/n) > y

UNKNOWN_1 STREAMS

**********************

ERROR: illegal parameter: FOOBAR

**********************

continue? (y/n) > y
```

Coding high-level languages other than ccperl

When coding report procedures in languages other than ccperl, for example, Visual C++, Java, Javascript or Visual Basic, refer to the programming examples available in the **T0046** package.

Obtaining the T0046 package

Obtain the **T0046** package at the following URL:

https://www6.software.ibm.com/reg/rational/rational-i

The site explains which browser types and versions are supported. At the Rational Download and Licensing Center page, in **Search**, enter T0046 and click **Search**. At the Rational ClearCase Add-ins page, find the T0046 entry.

Appendix D. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department BCFB
20 Maguire Road
Lexington, MA 02421
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Trademarks

AIX, ClearCase, ClearQuest, DB2, IBM, Rational, RequisitePro, and XDE are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Readers' Comments — We'd Like to Hear from You

ClearCase and Rational ClearCase LT
Guide to Managing Software Projects
Version 7.0.0

Publication No. GI11-6712-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
GI11-6712-00

IBM

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Attn: Dept CZLA
20 Maguire Road
Lexington, MA 02421-3112

Fold and Tape | **Please do not staple** | Fold and Tape

GI11-6712-00

Cut or Fold
Along Line

Printed in USA

GI11-6712-00