MANAGING SOFTWARE PROJECTS WITH CLEARCASE

Release 4.1 and later

Windows Edition

Rational®
the e-development company™

800-023556-000

ClearCase and MultiSite Release Notes
Document Number 800-023556-000 August 2000
Rational Software Corporation 20 Maguire Road Lexington, Massachusetts 02421

Technical Acknowledgments

This software and documentation is based in part on BSD Networking Software Release 2, licensed from the Regents of the University of California. We acknowledge the role of the Computer Systems Research Group and the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley and the Other Contributors in its development.

This software and documentation is based in part on software written by Victor A. Abell while at Purdue University. We acknowledge his role in its development.

This product includes software developed by Greg Stein <gstein@lyra.org> for use in the mod_dav module for Apache (http://www.webdav.org/mod_dav/).

Rational®
the e-development company™

Contents

Part 2: Working in Base ClearCase

Figures

Tables

Preface

ClearCase, a *configuration management* system, is designed to help software development teams track the objects used in software builds. You can use base ClearCase to create a customized configuration management environment, or you can adopt the Unified Change Management (UCM) process. UCM is an out-of-the-box process, layered on base ClearCase and ClearQuest functionality, for organizing software development teams and their work products.

About This Manual

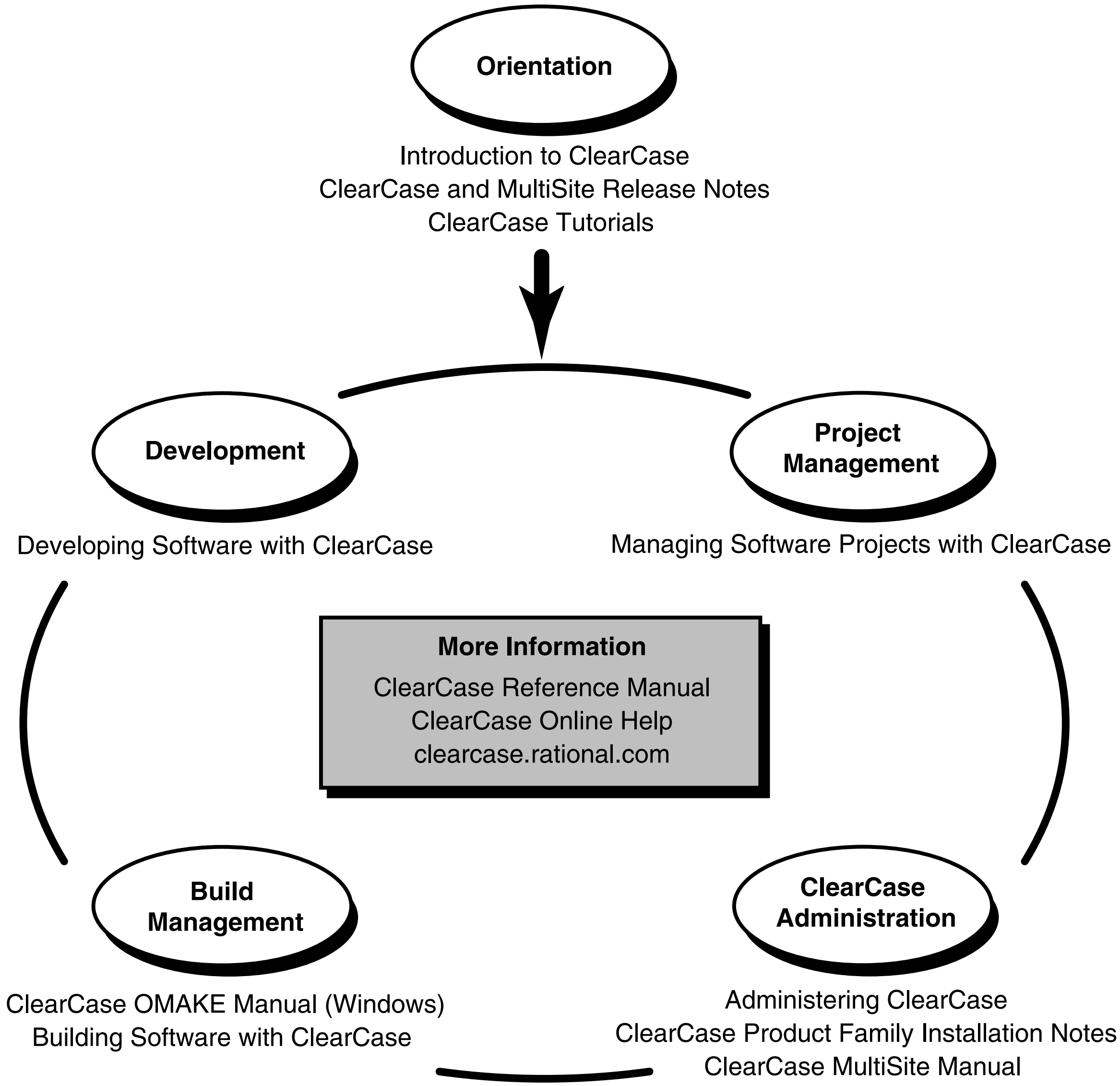
This manual shows project managers how to set up and manage a configuration management environment for their development team using either UCM or the customizable features of base ClearCase.

Organization

The manual is divided into two parts:

- *Part 1: Working in UCM*. Read this part if you plan to use UCM to implement your team's development process.
- *Part 2: 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.

ClearCase Documentation Roadmap



Typographical Conventions

This manual uses the following typographical conventions:

- *ccase-home-dir* represents the directory into which the ClearCase Product Family has been installed. By default, this directory is **/usr/atria** on UNIX and **C:\Program Files\Rational\ClearCase** on Windows.

- *attache-home-dir* represents the directory into which ClearCase Attache has been installed. By default, this directory is **C:\Program Files\Rational\Attache**, except on Windows 3.*x*, where it is **C:\RATIONAL\ATTACHE**.

- **Bold** is used for names the user can enter; for example, all command names, file names, and branch names.

- *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 are in small caps and 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, ClearCase recognizes "**...**" within a pathname as a wildcard, similar to "*" or "?". See the **wildcards_ccase** reference page for more information.

- If a command or option name has a short form, a "medial dot" (·) character indicates the shortest legal abbreviation. For example:

  lsc·heckout

  This means that you can truncate the command name to **lsc** or any of its intermediate spellings (**lsch**, **lsche**, **lschec**, and so on).

Online Documentation

The ClearCase graphical interface includes a standard Windows help system.

There are three basic ways to access the online help system: the **Help** menu, the **Help** button, or the F1 key. **Help→Help Topics** provides access to the complete set of ClearCase online documentation. For help on a particular context, press F1. Use the **Help** button on various dialog boxes to get information specific to that dialog box.

ClearCase also provides access to full "reference pages" (detailed descriptions of ClearCase commands, utilities, and data structures) with the **cleartool man** subcommand. Without any argument, **cleartool man** displays the **cleartool** overview reference page. Specifying a command name as an argument gives information about using the specified command. For example:

> **cleartool man**	*(display the cleartool overview page)*
> **cleartool man man**	*(display the cleartool man reference page)*
> **cleartool man checkout**	*(display the cleartool checkout reference page)*

ClearCase's **–help** command option or **help** command displays individual subcommand syntax. Without any argument, **cleartool help** displays the syntax for all **cleartool** commands. **help checkout** and **checkout –help** are equivalent.

> **cleartool lsprivate –help**

```
Usage: lsprivate [-tag view-tag] [-invob vob-selector] [-long | -short]
                 [-size] [-age] [-co] [-do] [-other]
```

Additionally, the online *ClearCase Tutorial* provides important information on setting up a user's environment, along with a step-by-step tour through ClearCase's most important features. To start the *ClearCase Tutorial*, choose **Tutorial** in the ClearCase folder off the **Start** menu.

Technical Support

If you have any problems with the software or documentation, please contact Rational Technical Support via telephone, fax, or electronic mail as described below. For information regarding support hours, languages spoken, or other support information, click the **Technical Support** link on the Rational Web site at **www.rational.com**.

Your Location	**Telephone**	**Facsimile**	**Electronic Mail**
North America	800-433-5444 toll free or 408-863-4000 Cupertino, CA	408-863-4194 Cupertino, CA 781-676-2460 Lexington, MA	**support@rational.com**
Europe, Middle East, and Africa	+31-(0)20-4546-200 Netherlands	+31-(0)20-4546-201 Netherlands	**support@europe.rational.com**
Asia Pacific	61-2-9419-0111 Australia	61-2-9419-0123 Australia	**support@apac.rational.com**

1

Choosing Between UCM and Base ClearCase

Before you can start to use 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 rest of this manual is organized into two parts. Part 1 describes how to manage a project using UCM. Part 2 describes how to manage a project using the various tools in base ClearCase.

1.1 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 prescribed method of using 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.

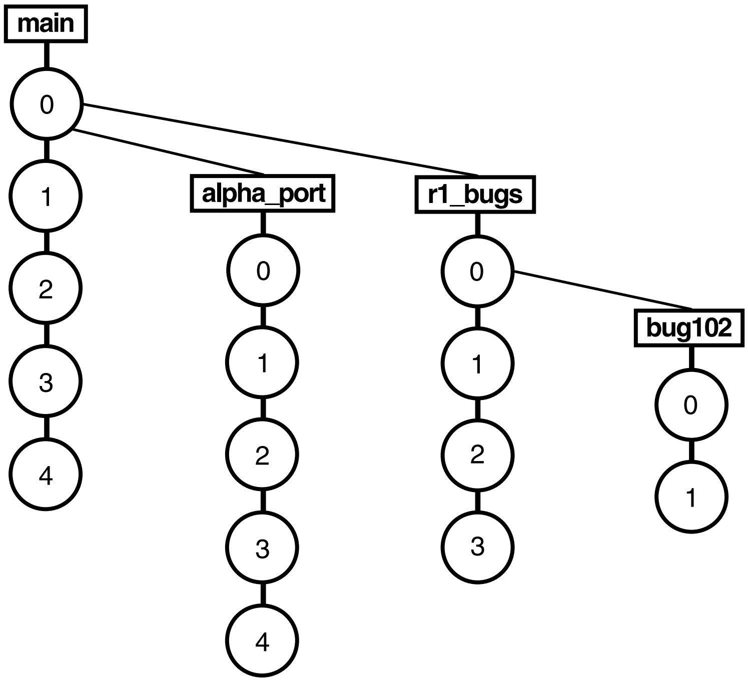
UCM offers the convenience of an out-of-the-box solution; base ClearCase offers the flexibility to implement virtually any configuration management solution that you deem appropriate for your environment.

Branching

Base ClearCase uses branches 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. Base ClearCase allows you to 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. 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.

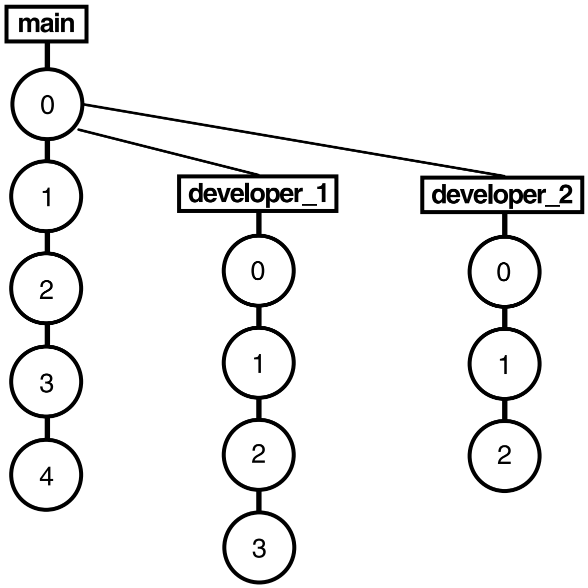
Figure 1 Branching Hierarchy in Base ClearCase



UCM uses branches also, but you do not have to manipulate them directly because it layers streams over the branches. A *stream* is a ClearCase object that maintains a list of activities and determines which versions of elements appear in a developer's view. In UCM, a project contains one *integration stream*, which records the project's shared set of elements, and multiple *development streams,* in which developers work on their parts of the project in isolation from the team. UCM does not allow for complex branch hierarchies. The project's integration stream uses one branch. Each development stream uses its own branch, which is a subbranch of the integration stream's branch. Development stream branches cannot have subbranches. Figure 2 illustrates the simple branching hierarchy that supports UCM streams.

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.

Figure 2 Branching Hierarchy Under UCM Streams



Creating and Using Baselines

Both base ClearCase and UCM allow you to create baselines. UCM automates the creation process and provides additional support for performing operations on baselines. A *baseline* identifies 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 a project's 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

In base ClearCase, you can create 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 project's recommended baseline. 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 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-ClearQuest integration, 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's recommended baseline before delivering their work to the project's shared work area.

- Developers must notify other team members by e-mail when they deliver work to the project's 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 GUI or command-line interface (CLI).

1.2 Using Base ClearCase Tools with UCM

This manual is organized into two parts: Part 1 for UCM and Part 2 for base ClearCase. If you are managing a UCM project, you may occasionally want to extend UCM by using some of the tools in base ClearCase. In particular, you may want to use ClearCase attributes, triggers, and hyperlinks to customize development policies.

Part 1: Working in UCM

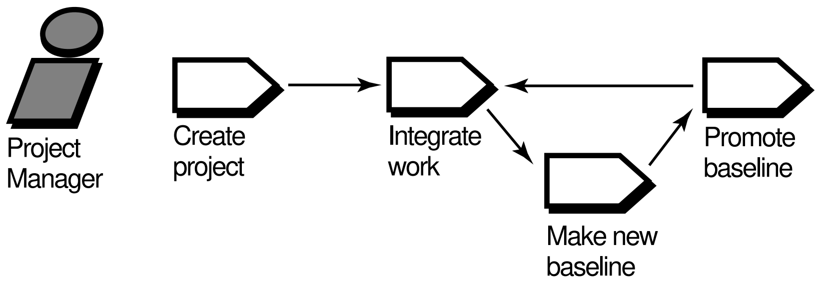
The following chapters describe how to plan, set up, and manage a UCM project to implement your team's development process.

.

2 Understanding UCM

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

2.1 The Project Management Cycle



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 shared work area and typically multiple private work areas. Private work areas allow developers to work on activities in isolation. As project manager, you are responsible for maintaining the project's shared work area. Work within a project 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 a component.

2. Developers join the project by creating their private work areas and populating them with the contents of the project's baselines.

3. Developers create activities and 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, and build and test their work in their private work areas, they share their work with the project team by performing *deliver* operations. A deliver operation merges work from the developer's private work area to the project's shared work area.

5. In the shared work area, you integrate the work delivered by developers.

6. Periodically, you create new baselines in the shared work area that incorporate the delivered work.

7. You perform quick validation tests to make sure that the new baselines build and appear to work correctly. A team of software quality engineers performs more extensive testing.

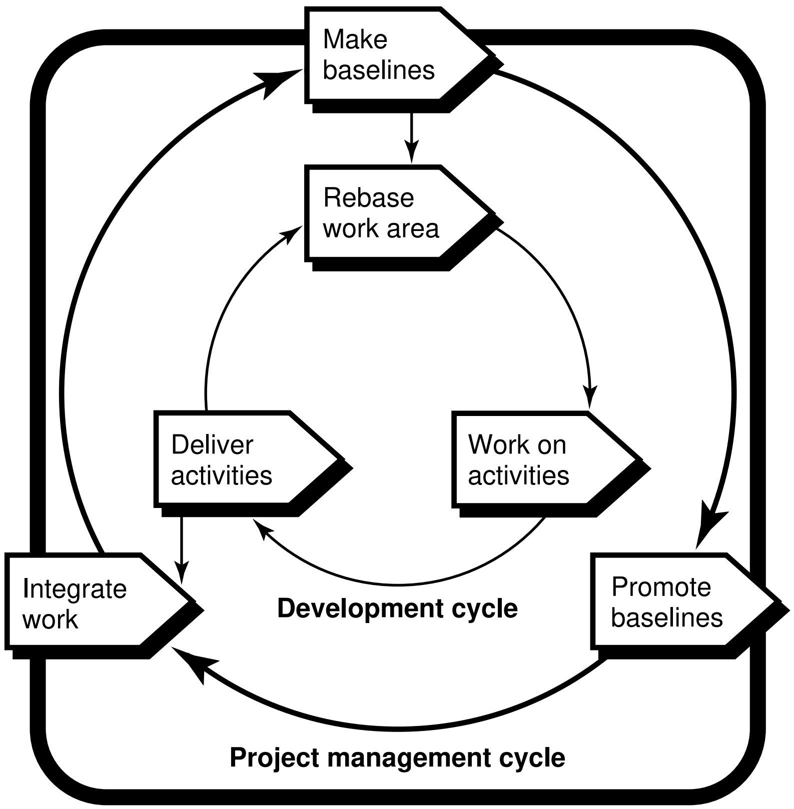
8. Periodically, as the quality and stability of baselines improve, you adjust the promotion level attribute of baselines to reflect appropriate milestones, such as Built, Tested, or Released. When the new baselines pass a sufficient level of testing, you designate them as the *recommended* set of baselines.

9. Developers perform *rebase* operations to update their private work areas to include the set of versions represented by the new recommended baselines.

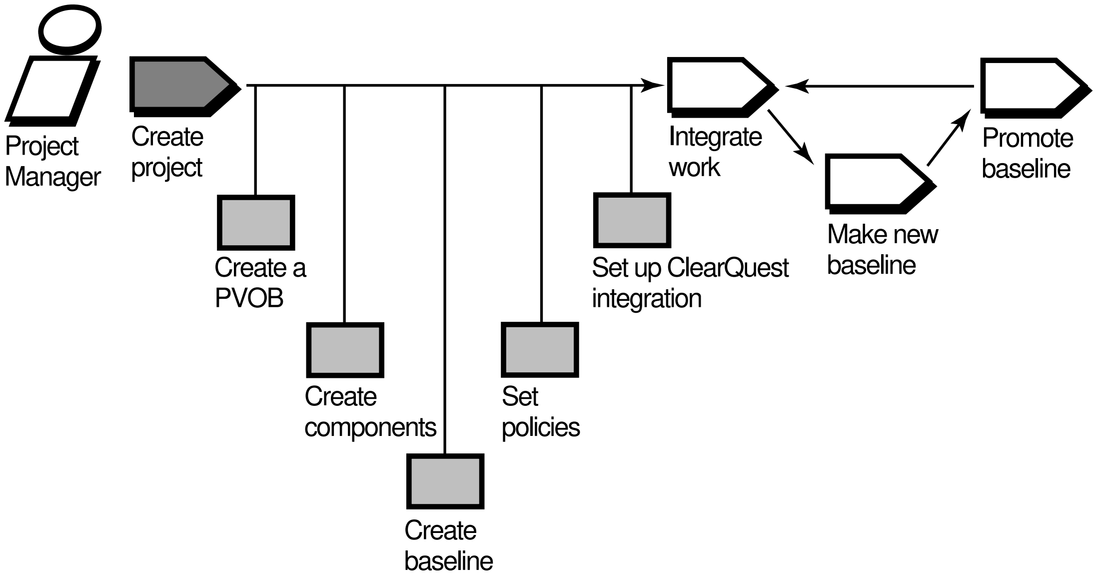
10. Developers continue the cycle of working on activities, delivering completed activities, updating their private work areas with new baselines.

This list of UCM tasks can be seen as two cycles: project management and development. Figure 3 illustrates the connection between these cycles.

Figure 3 Project Management and Development Cycles in UCM



2.2 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 the developers work on
- Create baselines that identify the versions of files with which the developers start their work
- Select the development policies you want to enforce

Creating a PVOB

ClearCase stores file elements, directory elements, derived objects, and metadata in a 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's ClearCase administrator to see whether a PVOB has already been created. For details on creating a PVOB, see *Creating the Project VOB* on page 56.

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 one large set of directories and files.

Within a component, you organize directory and file elements into a directory tree. The component's root directory must be the root directory of a VOB. You can convert existing VOBs into components, or you can create a component from scratch. For details on creating a component from scratch, see *Creating Components* on page 57. For details on converting a VOB into a component, see *Making a VOB into a Component* on page 62.

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 ClearCase object that maintains a list of activities and determines which versions of elements appear in your view.

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

Each developer on the project 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.

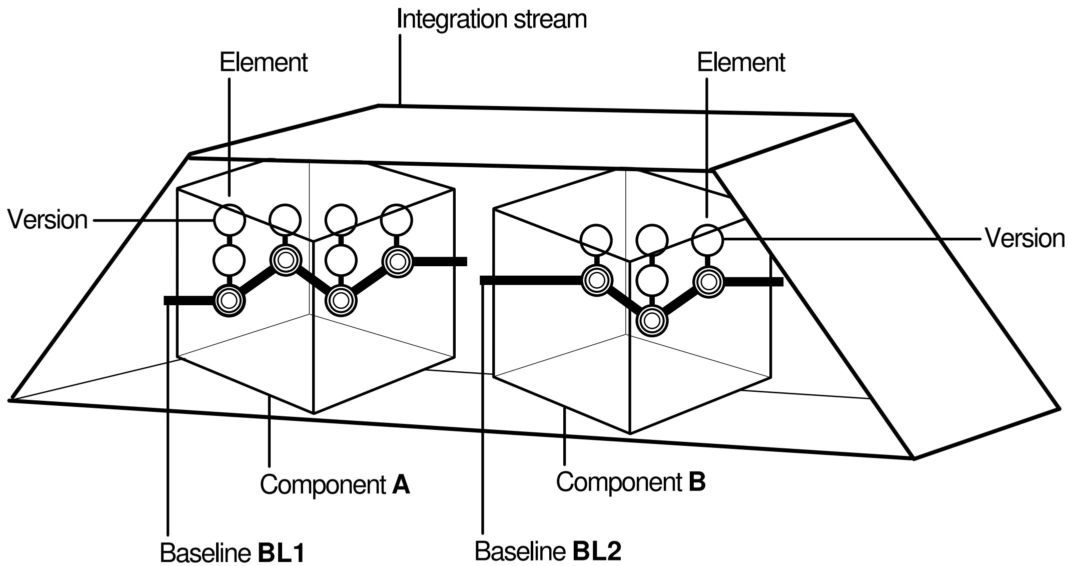
When you create a project from the UCM GUI, ClearCase creates the integration stream for you. If you create a project from the command-line interface, you need to create the integration stream explicitly. Developers create their development streams and development views when they join the project. See *Developing Software with ClearCase* for information on joining a project.

Starting from a Baseline

After you create project components or select existing components, you must identify the baseline or baselines that serve as the starting point for the team's developers. A baseline identifies one version of every element visible in a component. Figure 4 shows baselines named BL1 and BL2 that identify versions in Component A and Component B, respectively.

The project's integration stream records the baselines. When developers join the project, they populate their work areas with the versions of directory and file elements represented by the baselines. This practice ensures that all members of the project team start with the same set of files.

Figure 4 Baselines of Two Components



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 project's latest recommended baseline before they deliver work to the integration stream. 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 *Considering Which Development Policies to Enforce* on page 41.

Setting Up the UCM-ClearQuest Integration

You can use UCM without Rational ClearQuest, the change request management tool. The integration with ClearQuest adds significant project management and activity management capabilities. When you set up a UCM project to work with ClearQuest, the integration links all project activities to ClearQuest records. You can then take advantage of UCM's state transition model and ClearQuest's query, reporting, and charting features. These features allow you to do the following:

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

To set up the UCM-ClearQuest integration:

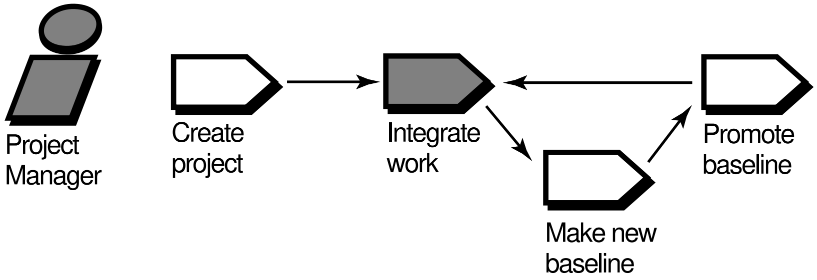
1. Enable a ClearQuest schema to work with UCM or use a predefined UCM-enabled schema.

2. Create or upgrade a ClearQuest user database to use the schema.

3. Enable your UCM project to work with ClearQuest.

See *Overview of the UCM-ClearQuest Integration* on page 20 for additional information about the integration.

2.3 Integrating Work into the Project (MultiSite)



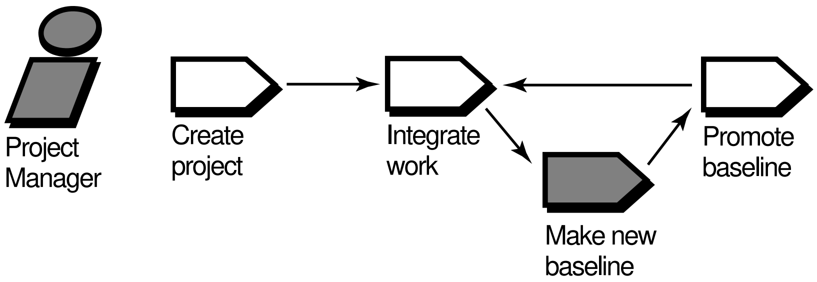
In most cases, developers complete the deliver operations that they start. If your project uses ClearCase MultiSite, you may need to complete some deliver operations. Many ClearCase customers use MultiSite, a product layered on ClearCase, to support parallel software development across geographically distributed project teams. 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, 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 MultiSite configuration, a team of developers may work at a remote site, and the project's integration stream 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 manager, you must complete these deliver operations. UCM provides a variation of the deliver operation called a *remote deliver*. When UCM detects a stream mastership situation, it makes the deliver operation a remote deliver, which starts the deliver operation but does not merge any versions. You then complete the deliver operation.

For details on completing remote deliver operations, see *Integrating the Project* on page 73.

2.4 Making a New Baseline



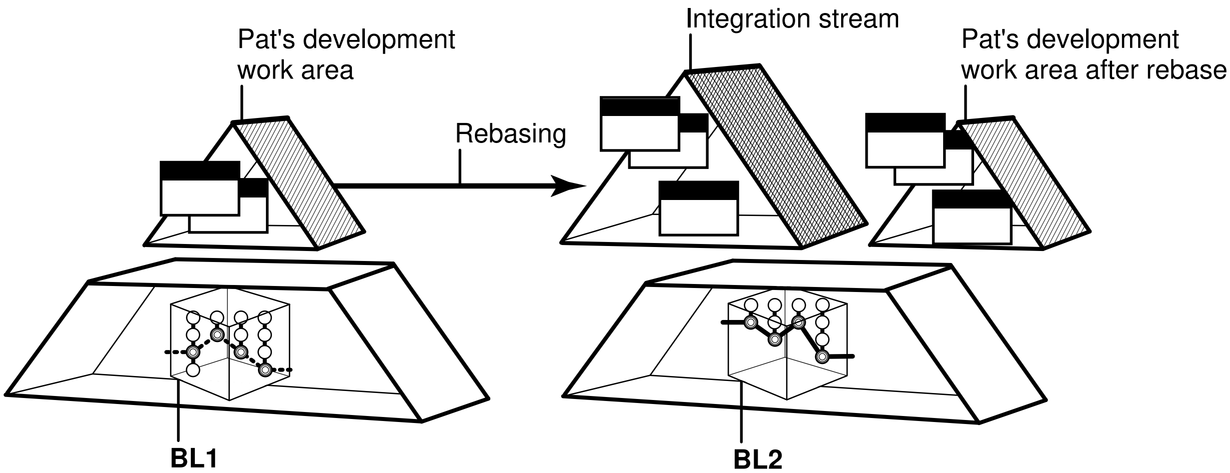
To ensure that developers stay in sync with each other's work, make new baselines regularly. A new baseline includes the work developers have delivered to the integration stream since the last baseline. To make a new baseline:

1. Lock the integration stream to prevent developers from delivering work while you create the baseline. Developers can continue to work on activities in their development streams.
2. Verify the stability of the project by building and testing its components.
3. Make the baseline.
4. Unlock the integration stream so that developers can resume delivering work.

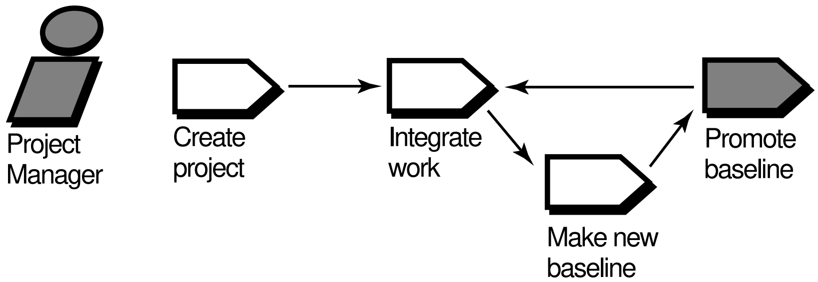
After your team of software quality engineers test the new baseline more extensively and determine that it is stable, make the baseline the *recommended* baseline. Developers then update their work areas with the new baseline by performing a rebase operation, which merges files and directories from the integration stream to the development stream.

Figure 5 illustrates a rebase operation from baseline BL1 to BL2. For details on making baselines, see *Creating a New Baseline* on page 75.

Figure 5 Rebase Operation



2.5 Promoting the Baseline

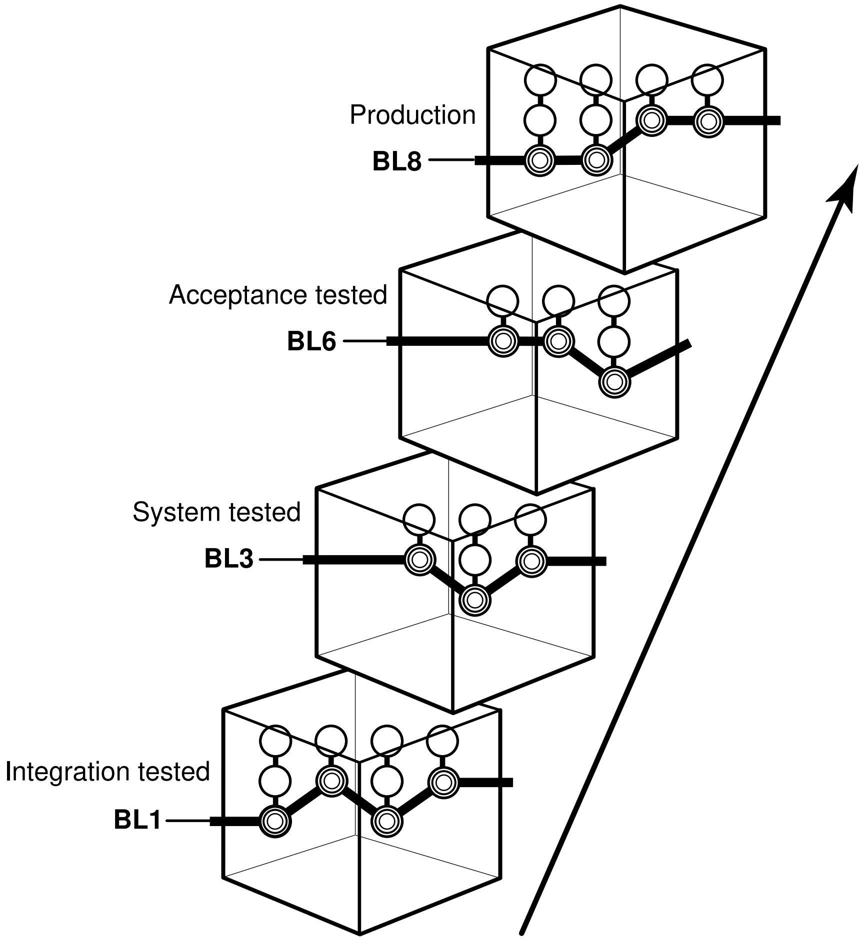


As work on your project progresses and the quality and stability of the components improve, change the baseline's promotion level attribute to reflect important milestones. The promotion level attribute typically indicates a level of testing. For example, Figure 6 shows the evolution of baselines through three levels of testing; the BL8 baseline is ready for production.

You can use promotion levels in development policies. For example, you can set a policy to make a baseline the recommended baseline when it reaches a particular promotion level, such as "tested." You can set another policy that requires developers to rebase their development streams to the set of recommended baselines before they deliver work. These policies help to ensure that developers update their work areas whenever a baseline passes an acceptable level of testing.

For details on promoting baselines, see *Promoting or Demoting the Baseline* on page 79

Figure 6 Promoting Baselines



2.6 Overview of the UCM-ClearQuest Integration

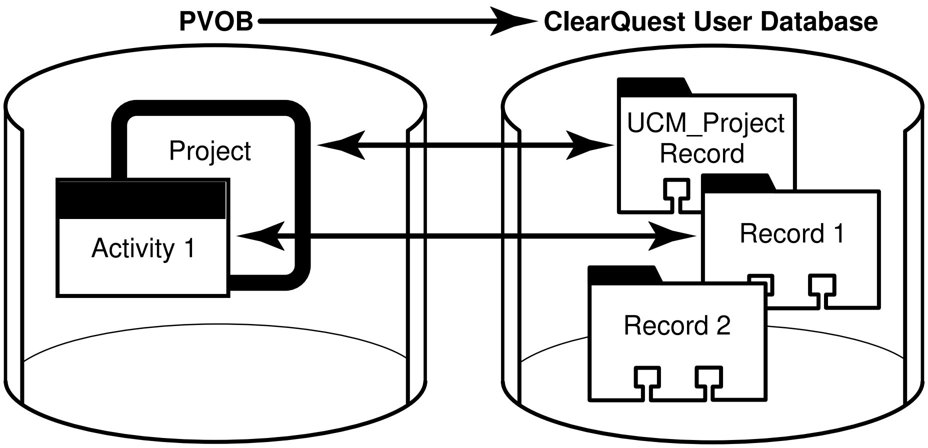
This section describes the following UCM-ClearQuest integration concepts:

- Association of UCM and ClearQuest objects
- UCM-enabled schema
- Queries
- State types

Associating UCM and ClearQuest Objects

Setting up the integration links UCM and ClearQuest objects. Figure 7 shows the bidirectional linking of these objects.

Figure 7 Association of UCM and ClearQuest Objects in Integration



When you enable a project to link to a ClearQuest user database, the integration stores a reference to that database in the project's PVOB.

Every ClearQuest-enabled project is linked to a project record of record type **UCM_Project** in the ClearQuest user database.

Every activity in a ClearQuest-enabled project is linked to a record in the database. An activity's title is linked to the headline field in its corresponding ClearQuest record. If you change an activity's title in ClearCase, the integration changes the headline in ClearQuest to match the new title, and the reverse is also true. Similarly, an activity's name is linked to the ID field in its ClearQuest record.

It is possible for a ClearQuest user database to contain some records that are linked to activities and some records that are not linked. Record 2 in Figure 7 is not linked to an activity. You may encounter this situation is if you have a ClearQuest user database in place before you adopt UCM. As you create activities, the integration creates corresponding ClearQuest records. However, any records that existed in that user database before you enabled it to work with UCM remain unlinked.

UCM-Enabled Schema

In ClearQuest, a schema is the definition of a database. To use the integration, you must create or upgrade a ClearQuest user database that is based on a UCM-enabled schema. A UCM-enabled schema contains certain fields, scripts, actions, and state types. ClearQuest includes two predefined UCM-enabled schemas, which you can use. You can also enable a custom schema or another predefined schema to work with UCM. For details on UCM-enabled schemas, see *Deciding Which Schema to Use* on page 38.

State Types

ClearQuest uses states 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 integration uses a particular state transition model. To implement this model, the integration uses state types. A *state type* is a template that defines actions and other attributes of a state. 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

For each state type, you can have multiple states. 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 49.

Queries in a UCM-Enabled ClearQuest Schema

A UCM-enabled schema includes six queries. When you create or upgrade a ClearQuest user database to use a UCM-enabled schema, the integration installs these queries in two subfolders of the Public Queries folder in the user database's 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 ClearQuest User Databases* on page 82

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. We strongly recommend that you write a configuration management plan before you begin creating projects and other UCM objects. After you create your plan, see Chapter 5, *Setting Up the Project* for information on how to implement it.

3.1 Using the System Architecture as the Starting Point

Essential to developing and maintaining high quality software is the definition of the system's architecture. The Rational Unified Process states that defining and using a system architecture is one of the six best practices to follow in developing software. A system architecture is the highest level concept of a system in its environment. The 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 a building's architecture (floor plans, electrical wiring, plumbing, and so on), a good software system architecture contains different views to represent its different aspects. The 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 Rational Unified Process suggests using five 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. Components are groups of related directory and file elements, which you develop, integrate, and release together. Large systems typically 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. To record a full picture of the project, include all files and directories connected with it. 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

- System and user documentation
- Requirements documents

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. 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:

- Size of the system
- Amount of integration required
- Whether you need to release multiple versions of the product concurrently

Size of the System

Consider the number of developers working on the system and the number of components. For a small system that consists of two or three components being developed by a dozen developers, one project probably makes sense. For a large system that consists of 20 components being developed by 100 developers, it may be wise to use several projects.

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.

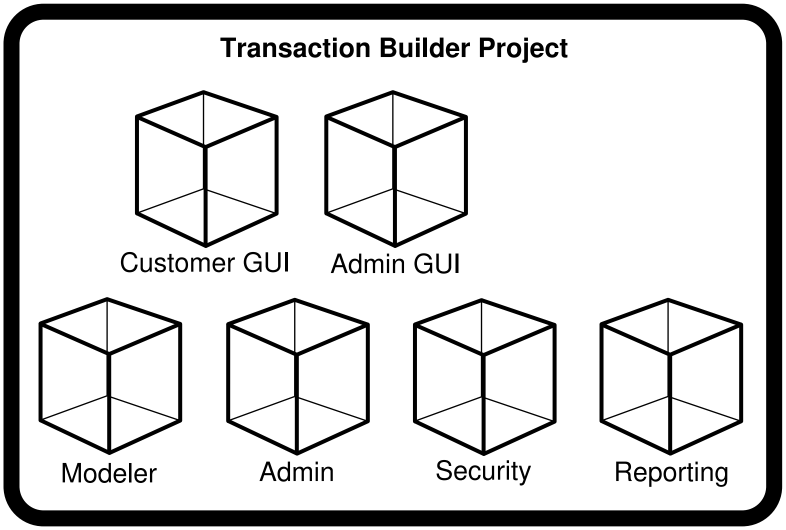
Need for Parallel Releases

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.

Example

Figure 8 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.

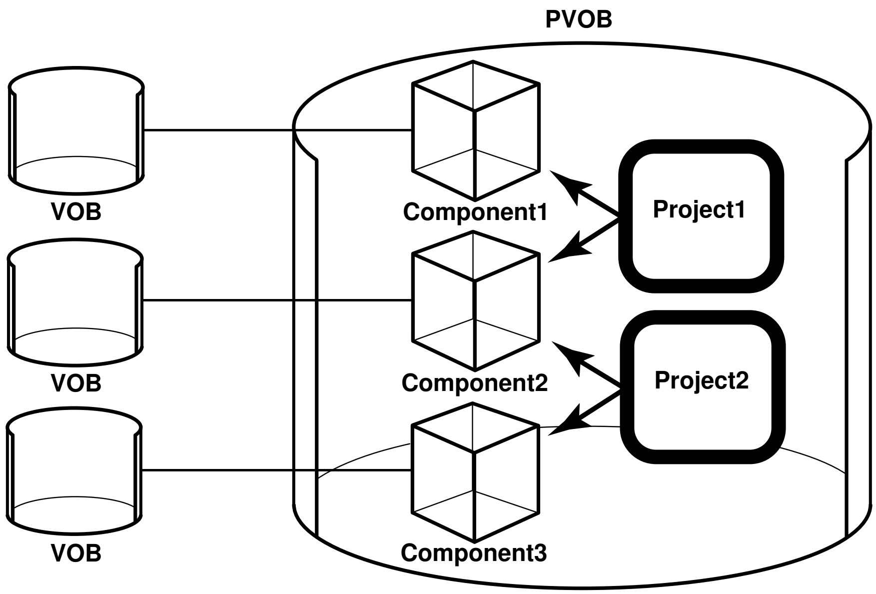
Figure 8 Components Used by Transaction Builder Project



Components and VOBs

ClearCase implements components as *versioned object bases* (VOBs), the repositories for versions of file elements, directory elements, derived objects, and metadata associated with them. Figure 9 illustrates this implementation. In Figure 9, both Project1 and Project2 use Component2.

Figure 9 Mapping Components to Projects



3.2 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:

- Ensure that your components are a suitable size for VOBs
- Identify any additional components
- Define the component directory structures
- Identify read-only components
- Identify nonmerging elements

Considering VOB Capacity

Because ClearCase implements components as VOBs, you must ensure that the contents of each planned component do not exceed the capacity of a VOB. See *Administering ClearCase* for details on VOB capacity.

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.
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.

Defining the Directory Structure

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

Table 1 Recommended Directory Structure for Components

Component	Directories	Typical Contents
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

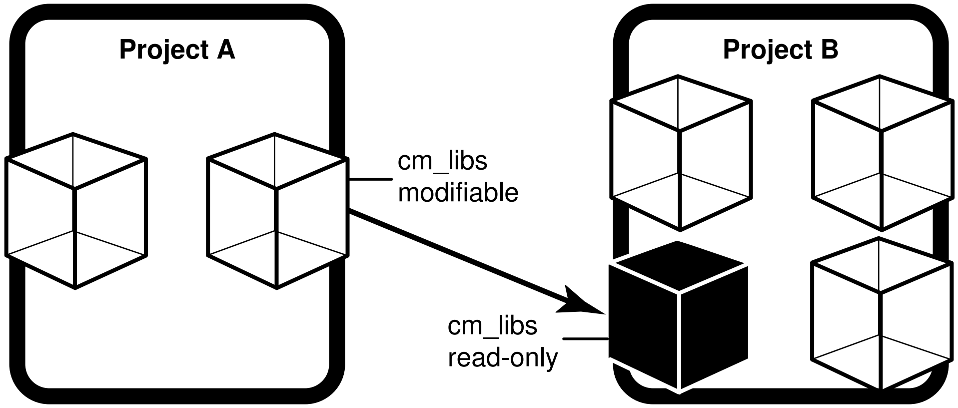
Identifying Read-Only Components

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. Therefore, one project team may be responsible for maintaining a component, and another project team may use that component to build other components.

For example, in Figure 10, Project A team members maintain a set of library files. Project B team members reference 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.

Figure 10 Using a Read-Only Component



3.3 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:

➤ When to create baselines
➤ How to name baselines
➤ The set of promotion levels
➤ How to test baselines

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 baselines of the existing project's components. 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.

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 a label type and apply it to the versions that you plan to include in your project.

Ongoing Baselines

After developers start working on the new project and making changes, create baselines 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 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 to the integration stream 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.

Defining a Naming Convention

Because baselines are an important tool for managing a project, define a meaningful convention for naming them. A useful baseline name provides this information:

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

For example: **V4.0TRANS_BL2_990519**

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. ClearCase provides the following default promotion levels:

- 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 in development policies for the project. For example, you can set a policy that makes a baseline the *recommended baseline* when it has a certain promotion level or a higher one. By default, when developers join the project or rebase their development streams, they use the recommended baselines. 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 69.

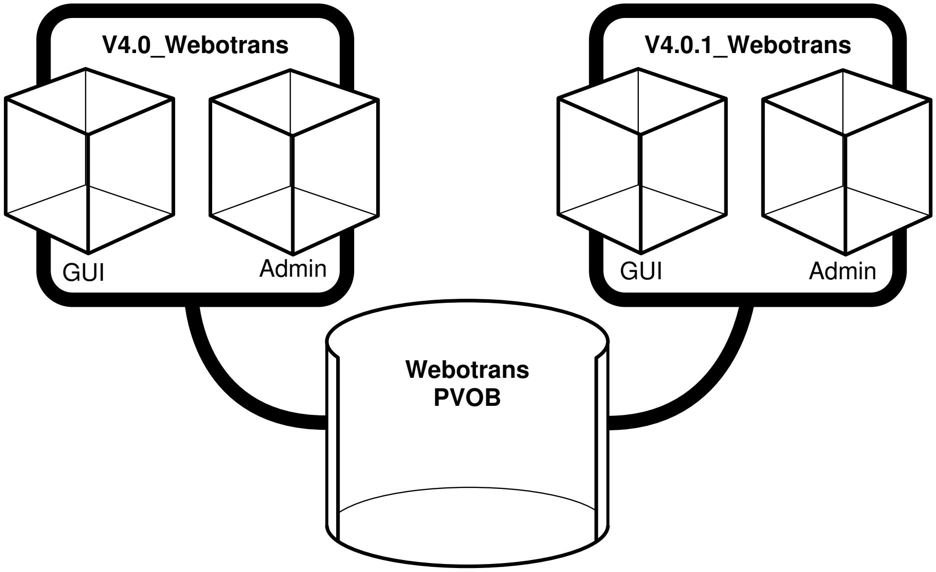
3.4 Planning PVOBs

ClearCase stores UCM objects such as projects, streams, activities, and change sets in project VOBs (PVOBs). In addition to storing objects, PVOBs can 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

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 11 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 11 Related Projects Sharing One PVOB



Consider using multiple PVOBs only when one or more of the following conditions applies:

➤ The projects do not share components and you anticipate that they will never need to share components.

➤ The projects are so large that PVOB capacity becomes an issue. For information on VOB capacity, see *Administering ClearCase*.

➤ You plan to use the UCM-ClearQuest integration, and you want to link projects to different ClearQuest user databases. See *Planning How to Use the UCM-ClearQuest Integration* on page 36 for information on the role of the PVOB in the integration.

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 when you create the PVOB. ClearCase then creates an **AdminVOB** hyperlink between the PVOB and the administrative VOB. Thereafter, when you create components, ClearCase creates **AdminVOB** hyperlinks between the components and the administrative VOB so that the components can 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, ClearCase makes **AdminVOB** hyperlinks between the components and the PVOB, and the PVOB assumes the role of administrative VOB.

For details on administrative VOBs and global types, see *Administering ClearCase*.

3.5 Identifying Special Element Types

The concept of element types allows ClearCase to handle each class of elements differently. An *element type* is a class of file elements. ClearCase includes predefined element types, such as **file** and **text_file**, and lets you define your own. 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 ClearCase encounters a merge situation during a deliver or rebase operation, it attempts to merge versions of the element. ClearCase requires user interaction only if it cannot reconcile differences between the versions. For certain types of files, you may want to impose different merging behavior.

Nonmerging Elements

Some types of files never need to be merged. For these 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. For these types of files, you can create an element type and specify **never** merge behavior.

NOTE: If you do not specify **never** merge behavior for these elements, developers could encounter problems when they attempt to deliver work to the project's integration stream. 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 current activity's change set. During a deliver operation, ClearCase attempts to merge these executable files to the integration stream unless the files are of an element type for which **never** merge behavior is specified.

Nonautomerging Elements

For some types of files, you may want to merge versions manually rather than let ClearCase merge them. 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 ClearCase change the form file during a merge operation, you want to regenerate the form file from the Visual Basic GUI.

For these types of files, you can create an element type and specify **user** merge behavior. For information on creating element types, see Chapter 12, *Using Element Types to Customize Processing of File Elements*, and the **mkeltype** reference page in *ClearCase Reference Manual*.

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 an administrative VOB, define the element type in the PVOB.

3.6 Planning How to Use the UCM-ClearQuest Integration

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

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

Mapping PVOBs to ClearQuest User Databases

This section describes three issues that you need to consider in deciding how many PVOBs to use for projects that link to ClearQuest user databases.

All Enabled Projects in a PVOB Must Link to the Same Database

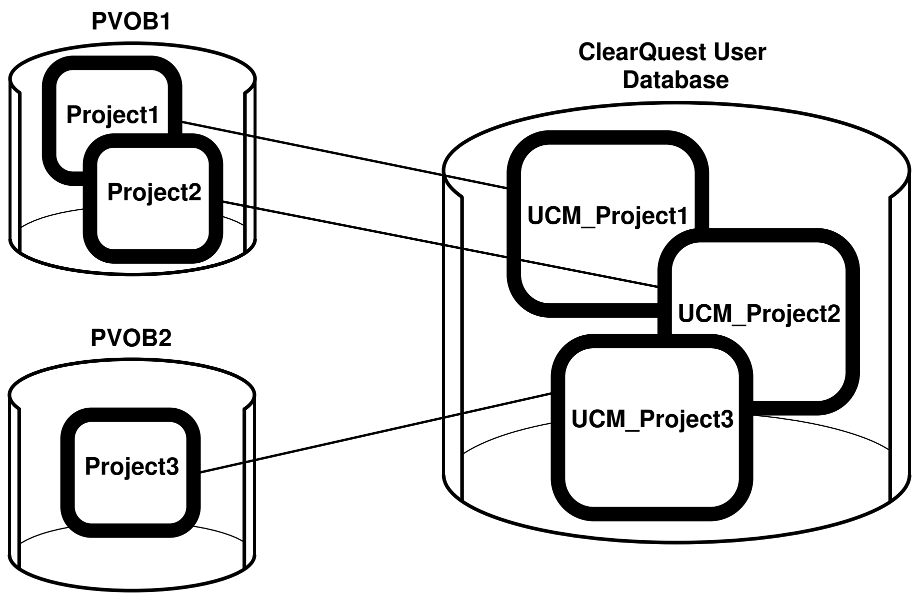
When you enable a project to link to a ClearQuest user database, the integration stores a reference to that database in the project's PVOB. Any other projects in that PVOB that you enable must use the same database. Therefore, be careful when choosing projects to store in a PVOB. If you plan to link projects to different databases, use different PVOBs.

NOTE: If you use ClearCase MultiSite, all PVOB replicas must have access to the ClearQuest user database.

Projects Linked to Same Database Must Have Unique Names

Although UCM allows you to create projects with the same name in different PVOBs, you cannot link those projects to the same ClearQuest user database. Figure 12 illustrates this naming requirement.

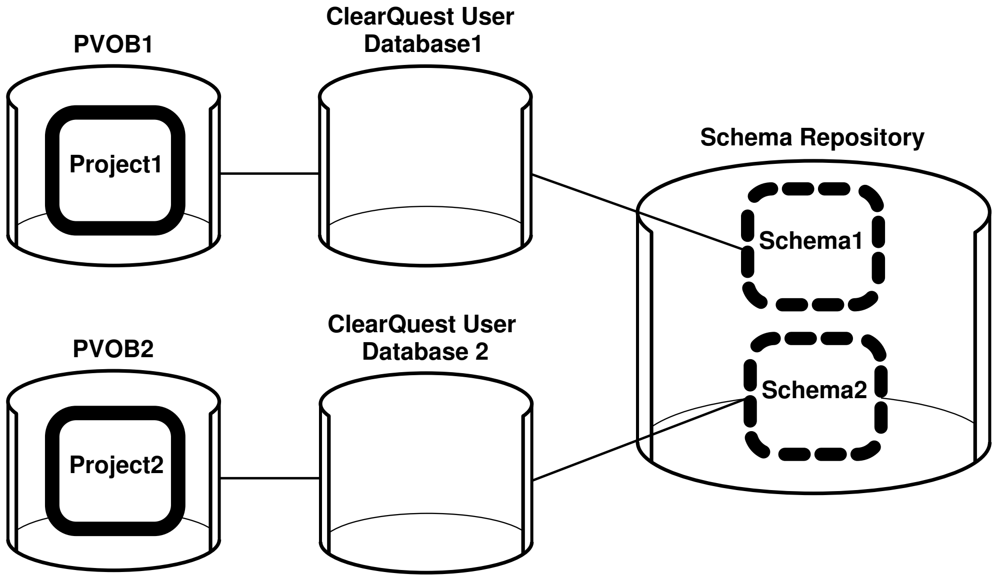
Figure 12 Projects in Multiple PVOBs Linked to the Same ClearQuest Database



Use One Schema Repository for Linked Databases

If some developers on your team work on multiple projects, we recommend that you store the schemas for the ClearQuest user databases that are linked to those projects in one schema repository, as shown in Figure 13. This allows developers to switch between projects easily. If you store the schemas in different schema repositories, developers must use the ClearQuest Maintenance Tool to connect to a different schema repository whenever they switch projects.

Figure 13 Using the Same Schema Repository for Multiple ClearQuest Databases



Deciding Which Schema to Use

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

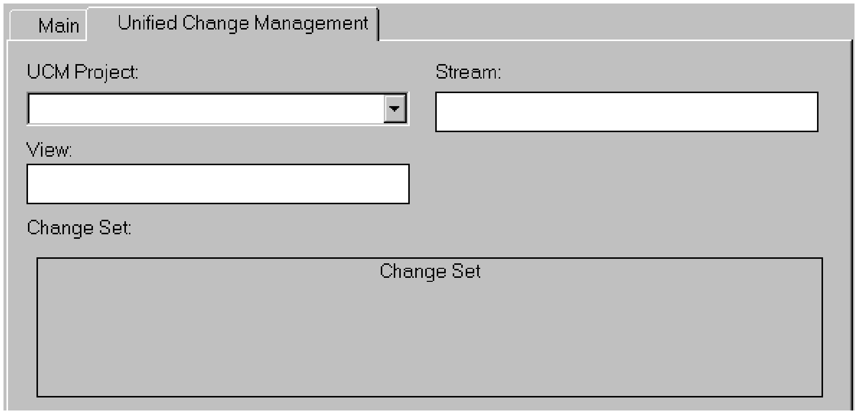
➤ The **UnifiedChangeManagement** package has been applied to the schema. A package contains metadata, such as records, fields, and states, that define specific functionality. Applying a package to a schema provides a way to add functionality quickly so that you do not have to build the functionality 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. Figure 14 shows the **Unified Change Management** tab.

➤ The **UCMPolicyScripts** package has been applied to the schema. This package contains the scripts for three ClearQuest development policies that you can enforce.

ClearQuest includes two predefined UCM-enabled schemas, named **UnifiedChangeManagement** and **Enterprise**. You can start using the integration right away by using one of these schemas, or you can use the ClearQuest Designer and the ClearQuest Package Wizard to enable a custom schema or another predefined schema to work with UCM. You can also use one of the predefined UCM-enabled schemas as a starting point and then modify it to suit your needs.

Figure 14 UCM Tab of Record Form for a UCM-Enabled Record Type



Overview of the UnifiedChangeManagement Schema

The UnifiedChangeManagement schema includes the following record types:

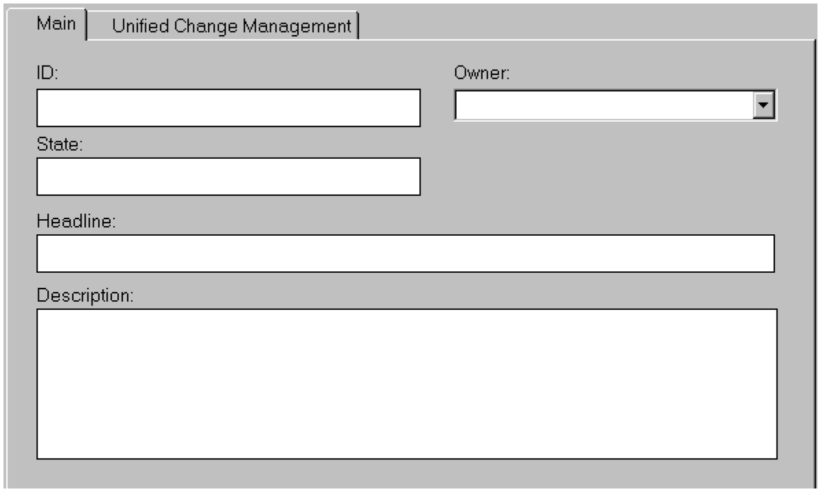
➤ **BaseCMActivity**
This is a lightweight record type that you can use to store information about activities that do not require additional fields. Figure 15 shows the **Main** tab of the BaseCMActivity record form. You may want to use this record type as a starting point and then modify it to include additional fields and states.

➤ **Defect**
This record type is identical to the record type of the same name that is included in

ClearQuest's other predefined schemas, with one exception: it is enabled to work with UCM. The Defect record type contains more fields and form tabs than the Activity 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 ClearQuest user database. You cannot modify this record type.

Figure 15 Main Tab of Record Form for the BaseCMActivity Record Type



Enabling a Schema for UCM

If you decide not to use one of the predefined UCM-enabled schemas, you need to 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 49.

  - 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 50.

- 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 *Policies Available in UCM-ClearQuest Integration* on page 43 for details.

3.7 Considering Which Development Policies to Enforce

UCM includes policies that you can set to enforce certain development practices within a project. Some policies are available only if you enable the project to work with ClearQuest.

Policies Available in UCM

This section describes the policies that are available regardless of whether you enable the project to work with ClearQuest.

Recommended Baselines

Recommended baselines are 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. Select the promotion level at which baselines become recommended baselines.

NOTE: If a component does not contain a baseline whose promotion level is at or above the recommended baseline promotion level, ClearCase uses the component's foundation baseline for the project when developers attempt to rebase their development streams or join the project.

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 29 .

Default View Types

When developers join a project, they use the Join Project Wizard to create their development views, integration views, and development streams. They use a development view and a development stream to work in isolation from the project team. They use an integration view to build and test their work against the latest work delivered to the integration stream by other developers. ClearCase provides two kinds of views: dynamic and snapshot. Decide which type of view to use as the default for development and integration views.

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

We recommend that you use dynamic views as the default view type for integration views. Dynamic views ensure that when developers deliver work to the integration 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.

Rebase Before Deliver

This policy requires developers to rebase their development streams to the project's current *recommended baselines* before they deliver work to the integration stream. 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 integration stream. This practice minimizes the amount of merging that developers must do when they perform deliver operations.

Allow Deliveries from Stream with Pending Checkouts

This policy allows developers to deliver work to the integration stream even if some files remain checked out in the development stream. If you do not set this policy, developers must check in all files in their development 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 the files associated with that activity.
2. The developer works on other activities.
3. Having completed several activities, the developer delivers them to the integration stream. Because the files associated with the first activity are still checked out, they are not included in the deliver operation. Even though the developer may build and test the changes successfully in the development work area, the changes delivered to the integration may fail because they do not include the checked-out files.

Policies Available in UCM-ClearQuest Integration

This section describes the policies that are available only when you enable the project to work with ClearQuest. ClearQuest uses scripts to implement these policies. You can modify a policy's behavior by editing its script. See *Customizing ClearQuest Project Policies* on page 52.

Check Before Work On

ClearQuest invokes this policy when a developer attempts to work on an activity. The default policy script checks to see whether the developer's user name matches the name in the ClearQuest record's Owner field. If the names match, the developer can work on the activity. If the names do not match, the Work On action 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.

Check Before ClearCase Delivery

This default policy script is a placeholder: it does nothing. ClearCase invokes this policy when a developer attempts to deliver an activity in a UCM-enabled project. We recommend that you edit the script to implement an approval process to control deliver operations. For example, you may want to add an **Approved** check box to the activity's record type and require that the project manager select it before allowing developers to deliver activities.

Do ClearQuest Action After Delivery

ClearCase calls this policy at the end of a deliver operation for each activity included in the deliver operation. The default policy script 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's record, and one of those fields is empty, the script 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's record, and resume the deliver operation.

The integration runs this script for each activity in the deliver operation. The script may return success for any number of activities before returning an error on an activity. For the successful activities, the script may change their state when it invokes the default action. When you recover from an error and rerun the deliver operation, the script looks at all activities again. For those that succeeded previously, the script does not attempt to change state. If you modify the script, be sure that it adheres to this behavior. ClearQuest returns an error if you attempt to change the state of a record to its current state.

4 Setting Up a ClearQuest User Database

This chapter describes how to set up a ClearQuest user database so that you can use the UCM-ClearQuest integration for your project. The steps in this chapter are typically completed by the ClearQuest database administrator. ClearQuest includes predefined schemas that are ready for use with UCM. You can also enable a custom schema, or another predefined schema, to work with UCM. See *Planning How to Use the UCM-ClearQuest Integration* on page 36 for information on the decisions you need to make before setting up the integration.

4.1 Using the Predefined UCM-Enabled Schemas

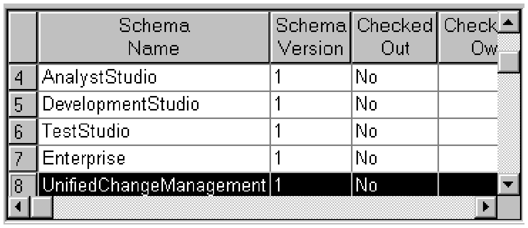
The predefined UCM schemas, named **UnifiedChangeManagement** and **Enterprise**, include the record type, field, form, state, and other definitions necessary to work with a UCM project. To set up a ClearQuest user database to work with UCM:

1. Create a user database that is associated with one of the predefined UCM-enabled schemas. In the 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 the new schema, as shown in Figure 16.

3. Click **Finish**.

Figure 16 Associating a User Database with a UCM-Enabled Schema



4.2 Enabling a Schema to Work with UCM

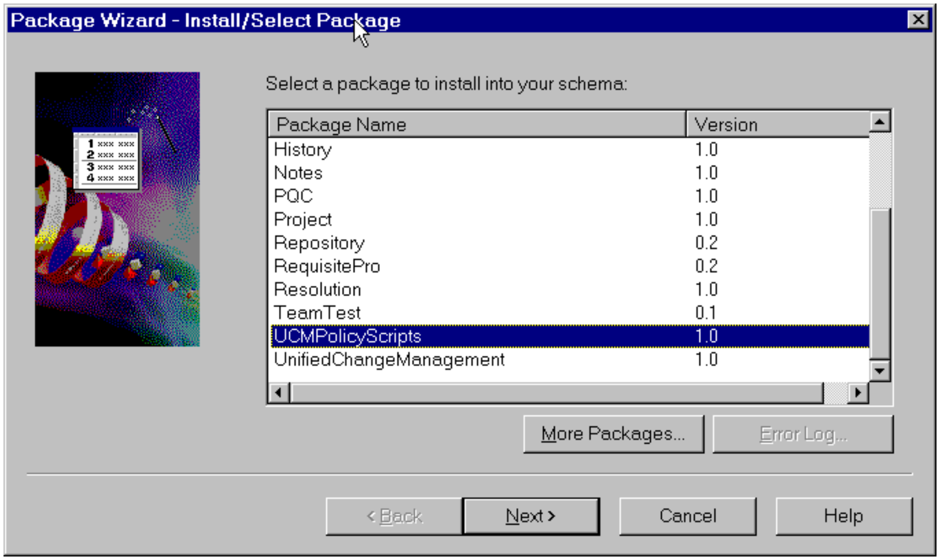
The predefined UCM schemas let you use the UCM-ClearQuest integration right away, but you may prefer to design a custom schema to track your project's activities and change requests, or you may prefer to use a different predefined schema. To enable a schema to work with UCM:

1. Ensure that the schema does not contain a record type named **UCM_Project**, which is a reserved name used by the UCM-ClearQuest integration.

2. In the ClearQuest Designer, click **Package>Package Wizard** to start the Package Wizard, as shown in Figure 17.

3. Add the **UCMPolicyScripts** package to your schema. If this package is not listed in the first page of the wizard, it has not been installed in your schema repository. To add the package to your schema repository, click **More Packages** to open the **Install Packages** dialog box; select the highest version of the package, and click **OK**. In the wizard, select the package, as shown in Figure 17. Click **Next**.

Figure 17 Adding the UCMPolicyScripts Package to a Schema



4. On the second page of the wizard, select your schema, and click **Finish**. To make the changes to the schema, ClearQuest checks out the schema for you. Check in the schema by clicking **File>Check In**. ClearQuest creates a new version of the schema.

5. Optionally, you can use the Package Wizard to apply the **BaseCMActivity** package 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 to include additional fields, states, and so on.

6. Apply the **UnifiedChangeManagement** package to the schema. Start the Package Wizard. Select **UnifiedChangeManagement**, and click **Next**.

7. In the second page of the wizard, select your schema. Click **Next**.

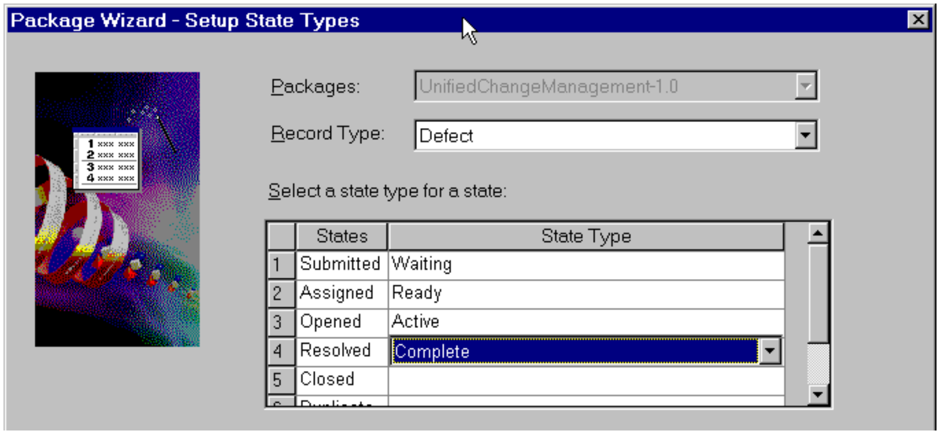
8. The third page of the wizard prompts you to specify the schema's record types. Select the check boxes of the record types that you want to enable. Click **Next**. All selected record types must meet the requirements listed in *Requirements for Enabling Custom Record Types* on page 49.

9. In the fourth page of the wizard, you must assign state types to the states for each record type that you choose to enable. For each state, click in the adjacent state type cell to display the list of available state types, as shown in Figure 18, and select one. To enable another record

type, click the arrow in the Record Type list to see the available record types. See *Setting State Types* on page 49 for a description of the four state types, and the rules for setting them.

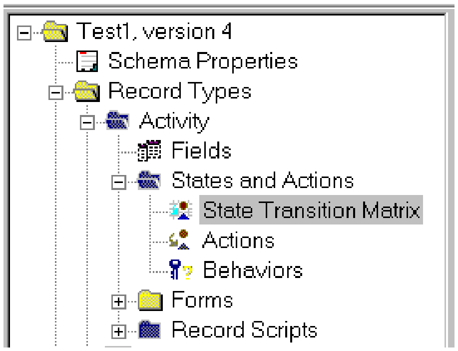
When you are finished, click **Finish** to check out the schema.

Figure 18 Assigning State Types to a Record Type's States



10. Before you can check in your schema, you must set default actions for the states of each enabled record type. Default actions are state transition actions that ClearQuest takes when a developer begins to work on an activity or delivers an activity. In the ClearQuest Designer workspace, navigate to the record type's state transition matrix, as shown in Figure 19.

Figure 19 Navigating to Record Type's State Transition Matrix



Double-click **State Transition Matrix** to display the matrix. Right-click the state column heading, and select **Properties** from its shortcut menu. Click the **Default Action** tab. Select the default action. See *State Transition Default Action Requirements for Record Types* on page 50 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**.

11. In the ClearQuest Designer workspace, navigate to the record type's **Behaviors**. Double-click **Behaviors** to display the Behaviors grid. Verify that the **Headline** field is set to **Mandatory** for all states. Verify that the **Owner** field is set to Mandatory for all Ready and Active state types.

12. Validate the schema changes by clicking **File>Validate**. Fix any errors that ClearQuest displays, and then check in the schema by clicking **File>Check In**.

13. Upgrade the user database so that it is associated with the UCM-enabled version of the schema by clicking **Database>Upgrade Database**.

Requirements for Enabling Custom Record Types

Before you can apply the **UnifiedChangeManagement** package to a custom record type, 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 ClearQuest-supplied **users** record type. 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**

- It contains an action named **Modify** of type Modify.

Setting State Types

The integration 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. 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

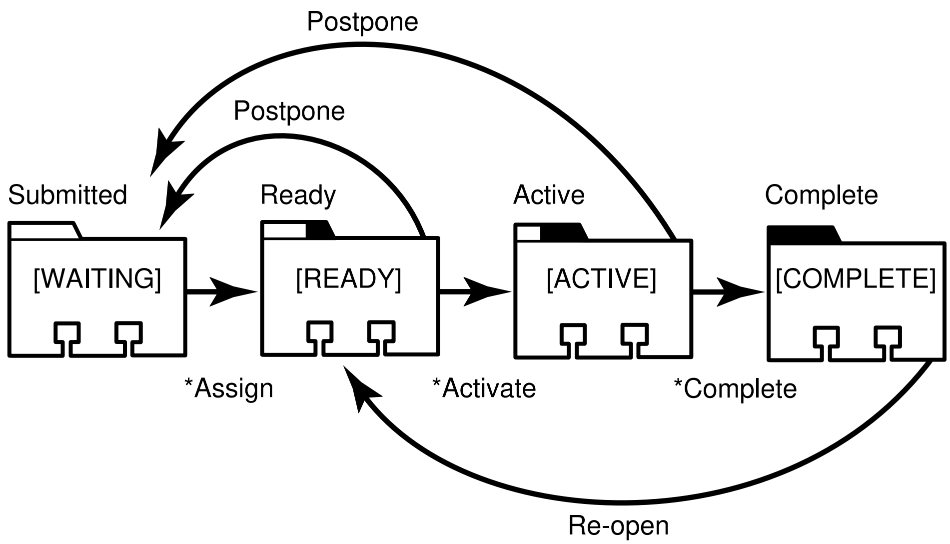
State Type	Description
Waiting	The activity is not ready to be worked on, either because it has not been assigned or it has not satisfied a dependency.
Ready	The activity is ready to be worked on. It has been assigned, and all dependencies have been satisfied.
Active	The developer has started work on the activity but has not completed it.
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. The transition from one state to the next must be made by a default action.

For example, Figure 20 shows the actions and default actions between the states defined in the UCM-enabled BaseCMActivity record type included in the predefined UCM schema. The default actions are identified with an asterisk (*). The state types are in uppercase letters enclosed in brackets. The states appear immediately above their state types.

Figure 20 State Transition Diagram for 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.

4.3 Customizing 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**

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 script calls the corresponding default script, which contains the logic for the default behavior. To modify the behavior of a policy, remove the call to the default script from the base script. Then add logic for the new behavior to the base script. Adhere to the rules stated in the base script.

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 ClearQuest clients on Windows NT , the integration uses the Visual Basic scripts. For ClearQuest clients on UNIX, the integration uses the Perl scripts. If you modify a policy's behavior and your environment includes ClearQuest clients on both platforms, be sure to make the same changes in both the Visual Basic and Perl versions of the policy's script. Otherwise, the policy will behave differently for ClearQuest clients on UNIX and Windows NT.

For descriptions of these policies, see *Policies Available in UCM-ClearQuest Integration* on page 43.

4.4 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 the product's 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/child controls in ClearQuest , you can accomplish this decomposition and tie the child activities back to the parent activity.

Using Parent/Child Controls

In ClearQuest, you use controls to display fields in record forms. A parent/child control, when used with a reference list field, lets you link related records. By adding a parent/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 ClearQuest change the state of the parent activity to Complete when all child activities have been completed, you need to write a hook. See *Administering Rational ClearQuest* for an example of such a hook.

4.5 Creating Users

Before you can assign activities to the developers on your project team, you must create user account profiles for each developer in ClearQuest. To do so:

1. In ClearQuest Designer, click **Tools>User Administration**.
2. Click **Add**.
3. Complete the **User Information** dialog box.

See *Administering Rational ClearQuest* and the ClearQuest Designer online help for details on creating user profiles.

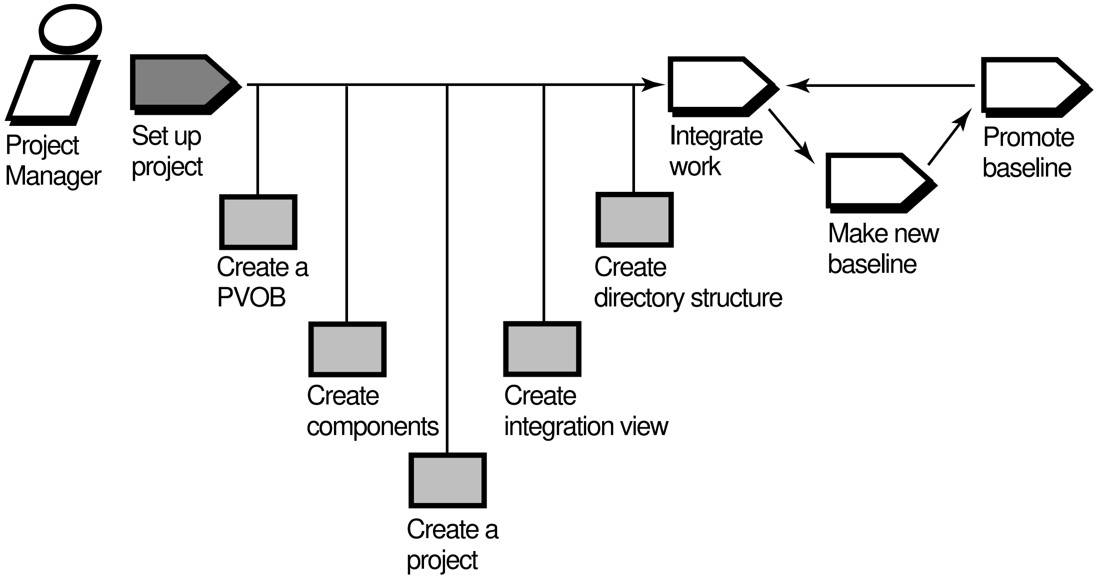
5 Setting Up the Project

This chapter describes how to 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*, for information on what to include in a configuration management plan.

The chapter presents five 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-ClearQuest integration
- Creating a development stream reserved for testing new baselines

5.1 Creating a Project from Scratch



This section describes how to create and set up a new project that is not based on an existing project or on an existing set of ClearCase VOBs.

Creating the Project VOB

To create a PVOB:

1. Click **Start>Programs>ClearCase Administration>Create VOB**. The VOB Creation Wizard appears.

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 **Create VOB as a UCM component** box clear. Although you can use one VOB as the PVOB and a component, we recommend against doing so unless your project is very small and you anticipate that it will remain small. Select the **UCM project data** check box.

3. In Step 2, specify the PVOB's storage directory. A PVOB storage directory is a directory tree that serves as the repository for the PVOB's contents. A PVOB's storage directory contains

the same subdirectories as a VOB's storage directory. (For details about VOB storage directory structure, see *Administering ClearCase*.) 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, ClearCase makes **AdminVOB** hyperlinks between the components and the PVOB, and the PVOB assumes the role of administrative VOB.

Creating Components

To create a component:

1. Start the VOB Creation Wizard.

2. In Step 1, enter a name for the component. Enter a comment to describe the purpose of the component. Select the **Create VOB as a UCM component** check box.

3. In Step 2, 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.

4. Step 3 prompts you to 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.

   ClearCase creates the component with an initial baseline that points to the **\main\0** version of the component's root directory.

Creating the Project

To create a project:

1. In the left pane of ClearCase Explorer, click **UCM** and then click **Project Explorer**. The Project Explorer appears. The Project Explorer is the graphical user interface (GUI) through which you create, manage, and view information about projects.

2. The left pane of the Project Explorer lists root folders for all PVOBs in the local ClearCase domain. Each PVOB has its own root folder. ClearCase creates the root folder using the name of the PVOB.

   ClearCase also creates a folder called **Components**, which 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 Step 1 of the New Project Wizard, enter a descriptive name for the project in the **Project Title** box. Enter a comment in the **Description** box to describe the purpose of this project.

6. Step 2 asks whether you want to create the project based on an existing project. Because you are creating a project from scratch, click **No**.

7. Step 3 asks you to choose the baselines that the project will use. These baselines are known as *foundation baselines* because they are the foundation upon which all work within the project is built.

   Click **Add** to open the **Add Baseline** dialog box. In the **Component** list, select one of the components that you previously created. The component's initial baseline appears in the **Baselines** list. Select the baseline. Be sure that the **Allow project to modify the component** check box is selected unless you want the component to be read-only. See *Identifying Read-Only Components* on page 29 for information on when you may want to use read-only components. Click **OK**. The baseline now appears in the list in Step 3. Continue to use the **Add Baseline** dialog box until the project contains its full set of foundation baselines.

8. Step 4 prompts you to specify the development policies to enforce for this project. Select the check boxes for the policies you want to enforce. See *Considering Which Development Policies to Enforce* on page 41 for information about each policy.

9. Step 5 asks whether to configure the project to work with the ClearQuest integration. To enable the project to work with ClearQuest, click **Yes**, and select a ClearQuest user database from the list. See *Enabling a Project to Use the UCM-ClearQuest Integration* on page 65 for details about the integration.

Defining Promotion Levels

ClearCase provides five baseline promotion levels. You can keep some or all of them, and you can define your own promotion levels. 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. The **Define Promotion Levels** dialog box appears. 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** dialog box appears. 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** dialog box. 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.

Creating an Integration View

When you create a project, ClearCase creates the project's integration stream for you. To see and make changes to the project's shared elements, you need an *integration view*. To create an integration view:

1. In the Project Explorer, navigate to the integration stream by moving down the object hierarchy:

   a. Root Folder

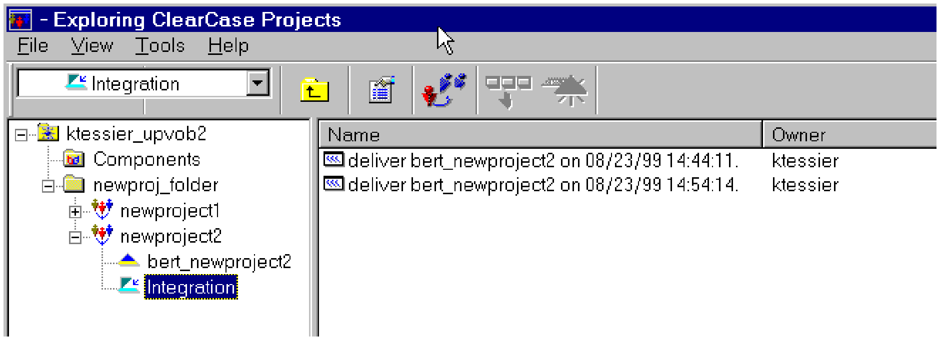
   b. Project Folder

   c. Project

   d. Stream

   Figure 21 illustrates this hierarchy.

Figure 21 Navigating to Integration Stream in Project Explorer



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

3. The View Creation Wizard appears. Accept the default values to create an integration view associated with the integration stream. By default, the View Creation Wizard uses this convention for the integration view name: *username_project-name*_**integration**.

ClearCase supports two kinds of views:

- Dynamic views, which use the ClearCase multiversion file system (MVFS) to provide immediate, transparent access to files and directories stored in VOBs. ClearCase maps a dynamic view to a drive letter in Windows Explorer.

- Snapshot views, which copy files and directories from VOBs to a directory on your computer.

We recommend that you make the integration view a dynamic view to 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 with ClearCase*. Dynamic views are not available on Windows.

Creating the Directory Structure

Because you are creating the project from scratch, you need to create the directory elements within the project's components to implement the directory structure that you define during the planning phase. See *Defining the Directory Structure* on page 28. To add a directory element to a component:

1. In Windows Explorer, navigate to the integration view. Double-click the component to display its contents.
2. Create a folder.
3. Right-click the folder to display the shortcut menu. Click **ClearCase>Add to Source Control**.
4. When prompted, specify an activity to be associated with the addition of the new directory element.

For additional information about creating directory and file elements, see *Developing Software with ClearCase* and the **mkelem** reference page.

Importing Directories and Files from Outside ClearCase

If you have a large number of files and directories that you want to place under ClearCase version control, you can speed the process by using the **clearexport** and **clearimport** command-line utilities. These two utilities allow you to migrate an existing set of directories and files from another version control software system, such as SourceSafe or PVCS, to ClearCase. You can also use **clearexport** and **clearimport** to place directories and files that are not currently under any version control under ClearCase control.

To migrate source files into a component:

1. Create and set a non-UCM view by using the View Creation Wizard. To start the View Creation Wizard, from ClearCase Explorer click **Base ClearCase>Create View**.
2. From within the view, run **clearexport** to generate a data file from your source files.
3. From within the view, 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. See *Making a Baseline from a Label* on page 63 for details.

As an alternative, you can use **clearexport** and **clearimport** on VOBs, and then convert the VOBs to components. See *Creating a Project Based on an Existing ClearCase Configuration* on page 62 for details on converting VOBs into components.

For details on using **clearexport** and **clearimport**, see *Administering ClearCase* and the **clearexport** and **clearimport** reference pages.

5.2 Creating a Project Based on an Existing ClearCase Configuration

If you have existing VOBs, you may want to convert them into components so that you can include them in projects. This section describes how to set up a project based on existing VOBs.

Creating the PVOB

Use the VOB Creation Wizard, as described in *Creating the Project VOB* on page 56, to create the PVOB. In Step 3, if you currently use an administrative VOB, select it in the list. ClearCase creates an **Admin VOB** hyperlink 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**.

Making a VOB into a Component

To make a VOB into a component:

1. In the Project Explorer, select the PVOB. Click **Tools>Import VOB**. The **Import VOB** dialog box appears.

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. When you are finished, click **Import**.

Making a Baseline from a Label

After you convert an existing VOB 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 the baseline:

1. 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, click **Start>Programs>ClearCase>Apply Label**.
2. In the Project Explorer, select the PVOB. Click **Tools>Import Label**. Step 1 of the Import Label Wizard appears.
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. InStep #2, 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's promotion level.

Creating the Project

Use the New Project Wizard to create the project as described in *Creating the Project* on page 57.

Creating an Integration View

Create an integration view as described in *Creating an Integration View* on page 59.

5.3 Creating a Project Based on an Existing Project

As you create new projects, you may need to create new versions 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.

Reusing Existing PVOB and Components

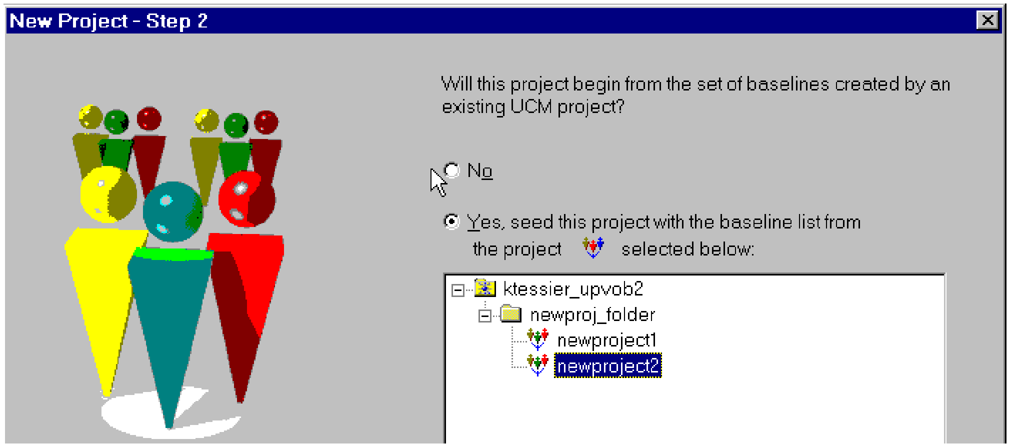
Because your project is a new version of an existing project and uses the same components as the existing project, do not create a new PVOB for this project. Continue to use the existing PVOB.

Creating the Project

Start the New Project Wizard, as described in *Creating the Project* on page 57, to create the project.

In Step 2 of the wizard, select **Yes** to indicate that the project begins from the baselines in an existing project. Then navigate to the project that contains those baselines. Figure 22 shows that the new project is based on the baselines in **newproject2**.

Figure 22   Step 2 of New Project Wizard



Step 3 lists the latest baselines in the project that you select in Step 2. You can add baselines from components that are not part of the existing project by clicking **Add** to open the **Add Baseline** dialog box. Similarly, you can remove a baseline by selecting it and clicking **Remove**.

Finish the remaining steps in the wizard as described in *Creating the Project* on page 57.

Creating an Integration View

When you create a new project, ClearCase creates a new integration stream 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 59.

5.4 Enabling a Project to Use the UCM-ClearQuest Integration

Before you can connect a project to a ClearQuest user database, you must set up the database to use a UCM-enabled schema. See Chapter 4, *Setting Up a ClearQuest User Database*.

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

1. In the left pane of the Project Explorer, right-click the project to display its shortcut menu. Click **Properties** to display its property sheet.

2. Click the **ClearQuest** tab and then select the **Project is ClearQuest-enabled** check box. Select the user database from the list, as shown in Figure 23. The first time that you enable a project, ClearQuest opens its **Login** dialog box. Enter your user name, password, and the name of the database to which you are linking the project.

3. Select the development policies that you want to enforce. See *Policies Available in UCM-ClearQuest Integration* on page 43 for a description of these policies. Click **OK** when you are finished.

If you are creating a new project, you can enable the project to work with ClearQuest by selecting **Project is ClearQuest-enabled** and selecting the user database in Step 5 of the New Project Wizard.

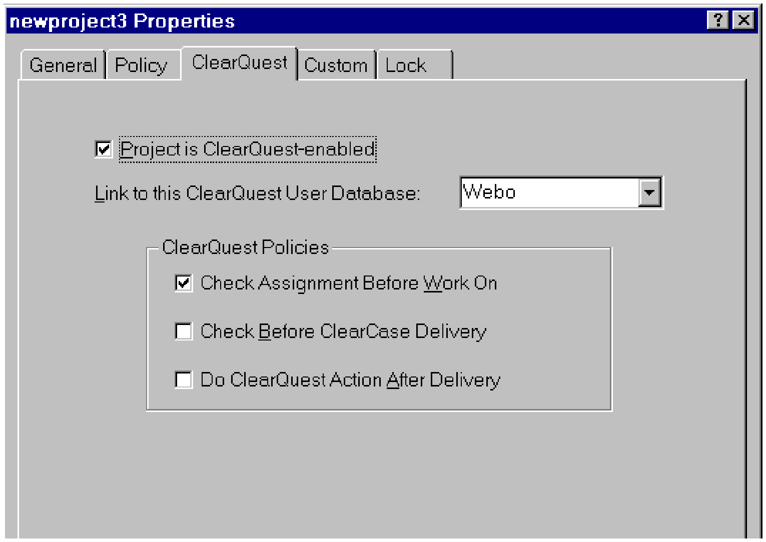
ClearCase does not require you to enable all projects in the PVOB to work with ClearQuest. However, all enabled projects in the same PVOB must use the same ClearQuest database. Therefore, give careful consideration to choosing the ClearQuest database. See *Mapping PVOBs to ClearQuest User Databases* on page 37 for details.

After you enable a UCM project to work with a ClearQuest user database, you may decide to link the project to a different user database. You can switch databases by selecting a different one on the ClearQuest tab of the project's property sheet if no other project in the same PVOB is ClearQuest-enabled, and no activities have been created.

Migrating Activities

If your project contains activities when you enable it to work with a ClearQuest database, the integration creates records for each of those activities by using the **UCMUtilityActivity** record type. If you want to store all of your project's 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.

Figure 23 Enabling a Project to Work with a ClearQuest User Database



Setting Project Policies

A UCM-enabled schema includes three policies that you can set from either ClearCase or ClearQuest.

In ClearCase, set the policies by selecting check boxes on the ClearQuest tab of the project's property sheet, as shown in Figure 23.

To set policies from ClearQuest:

1. Start the ClearQuest client by clicking **Start>Programs>Rational ClearQuest 2.0>Rational ClearQuest**. In the ClearQuest client workspace, navigate to the **UCMProjects** query, as shown in Figure 24.

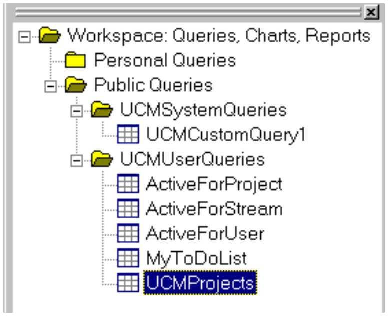
2. Double-click the query to display all UCM-enabled projects.

3. Select a project from the **Results** set. The project's form appears.

4. On the form, click **Actions** and select **Modify**. Select the check boxes for the policies you want to set.

See *Policies Available in UCM-ClearQuest Integration* on page 43 for descriptions of the policies.

Figure 24 Navigating to the UCMProjects Query



Assigning Activities

To create and assign activities in ClearQuest:

1. Start the ClearQuest client, and log in to the user database connected to the project.

2. Click **Actions>New**. The **Choose a record type** dialog box appears. Select a UCM-enabled record type, and click **OK**.

3. The **Submit** form appears. Fill in the boxes on each tab. On the **Main** tab, you must fill in at least the **Headline** and **Owner** boxes. On the **Unified Change Management** tab, select the project. When you finish filling in the boxes, click **OK**. ClearQuest creates the record.

User account profiles must exist in ClearQuest for the developers to whom you assign activities. See *Creating Users* on page 53 for details on creating user account profiles.

Disabling the Link Between a Project and a ClearQuest User Database

There may be times when you want to disable the link between a project and a ClearQuest user database. If another project in the same PVOB is ClearQuest-enabled or if activities have been created, you must first disable the link between each ClearQuest-enabled project in the PVOB and the user database. To disable the links:

1. On the **ClearQuest** tab of the project's property sheet, clear the **Project is ClearQuest-enabled** check box.

2. Click **OK** on the **ClearQuest** tab. The integration disables the link between the project and the ClearQuest database. The integration also removes any existing links between activities and their corresponding ClearQuest records.

3. Display the project's property sheet again, select the **Project is ClearQuest-enabled** check box, and select another user database if you want to link the project to a different user database.

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

Fixing Projects That Contain Linked and Unlinked Activities

It is possible that after you enable a project to work with ClearQuest, some of the project's activities remain unlinked to ClearQuest records. Similarly, when you disable the link between a project and ClearQuest, some activities may remain linked. Two 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 project's PVOB is in a ClearCase MultiSite configuration, and unlinked activities were added by a MultiSite synchronization operation to the local PVOB's project, which is enabled to work with ClearQuest.

Detecting the Problem

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 the Problem

To restore the project to a consistent state:

1. In the Project Explorer, display the project's property sheet, and click the **ClearQuest** tab.

2. Click **Ensure all Activities are Linked**. The integration checks all the project's activities. If the project is enabled, the integration links any unlinked activities. 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 **Ensure all Activities are Linked** operation again to correct the problem.

5.5 Creating a Development Stream for Testing Baselines

When you make a new baseline, we recommend that you 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.

To create a development stream:

1. In ClearCase Project Explorer, right-click the project, and select **New>Stream** from the shortcut menu.

   The **Create a Development Stream** dialog box appears.

2. By default, ClearCase uses the set of *recommended baselines* 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** dialog box appears.

3. In the **Change Baseline** dialog box, select the component that contains the baseline you want to test. Click **Change**.

   A second **Change Baseline** dialog box appears listing all baselines for the component.

4. 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** dialog box and repeat the process. When you are finished, click OK in the first **Change Baseline** dialog box.

5. In the **Create a Development Stream** dialog box, be sure that the **Prompt me to create a View for this stream** check box is selected. Click **OK**.

   The View Creation Wizard appears.

6. Complete the steps of the View Creation Wizard to create a view for the development stream.

Now the development stream is configured so that 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.

6 Managing the Project

After you create and set up a project, developers join the project, work on activities, and deliver completed activities to the integration stream. As project manager, you need to maintain the project so that developers do not get out of sync with each other's work. This chapter describes the following maintenance tasks:

- Adding components
- Integrating work delivered by the remote deliver model
- Making new baselines
- Testing baselines
- Promoting and demoting baselines
- Tracking the progress of the project
- Cleaning up the project

6.1 Adding Components

Over time, the scope of your project typically broadens, and you may need to add components. To add a component to a project's integration stream:

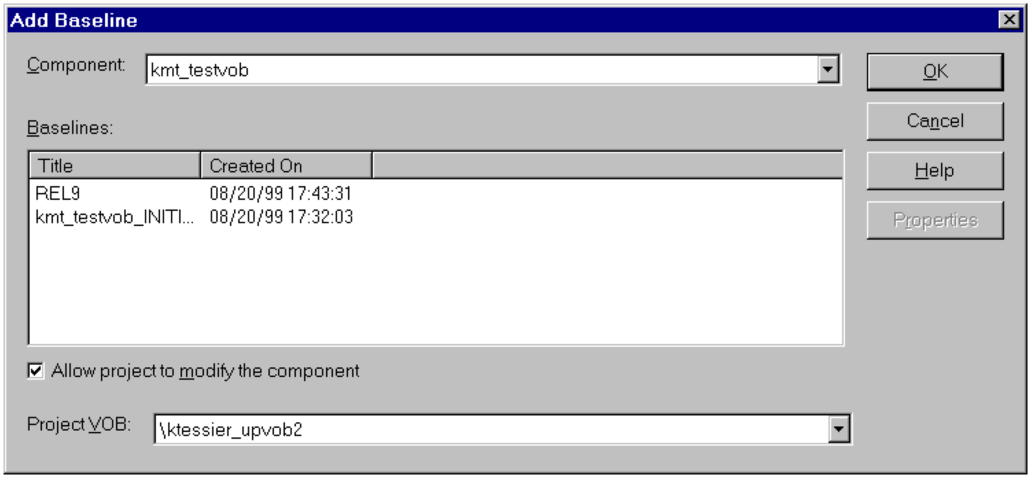
1. In the left pane of ClearCase Explorer, click **UCM** and then click **Project Explorer**.

2. In the right pane of the Project Explorer, right-click the project's integration stream to display its shortcut menu. Click **Properties** to open the integration stream's **Properties** dialog box.

3. Click the **Configuration** tab, and then click **Add**. The **Add Baseline** dialog box appears.

4. In the **Component** list, select the component that you want to add. The component's baselines appear in the Baselines list. Figure 25 shows the baselines available in the **kmt_testvob** component.

Figure 25 Add Baseline Dialog Box



5. In the **Baselines** list, select the baseline that you want to add to the project.

6. Click **OK**. The **Add Baseline** dialog box closes, and the baseline that you chose appears on the **Configuration** tab.

7. Click **OK** to close the integration stream's **Properties** dialog box.

   The **Rebase Stream Preview** dialog box appears. To modify the integration stream's configuration to include the new foundation baseline, UCM needs to rebase the integration stream.

8. Click **OK** in the **Rebase Stream Preview** dialog box.

9. Click **Complete** to finish the rebase operation.

Updating Snapshot View Load Rules

If your integration view is a snapshot view, you need to edit the view's load rules to include the components that you add to the integration stream. A snapshot view's load rules specify which components ClearCase loads into the view. To edit the integration view's load rules:

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

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, ClearCase updates the snapshot view's config spec, but it does not update the view's load rules. When you add a component, notify developers who use snapshot views that they need to update the load rules for their development views after they rebase their development streams to the new baseline.

6.2 Integrating the Project

In most cases, developers complete the deliver operations that they start. However, in a MultiSite configuration where the project's integration stream is mastered at a different replica than the developer's development stream, the developer cannot complete deliver operations. When ClearCase detects such a stream mastership situation, it makes the deliver operation a *remote deliver* operation.

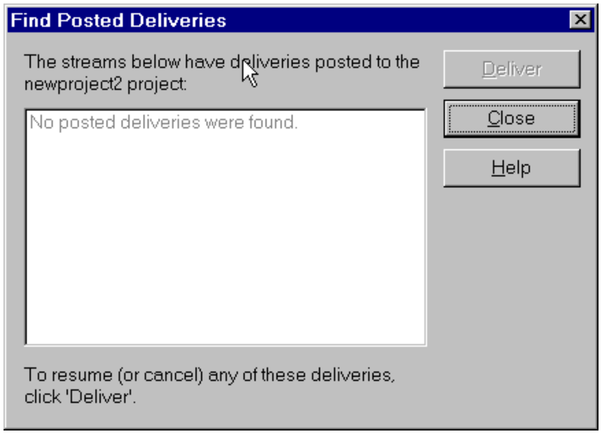
In a remote deliver operation, ClearCase starts the deliver operation but leaves it in the posted state. It is up to you, as project manager, to find and complete deliver operations in the posted state. Developers who have deliver operations in the posted state cannot deliver from, or rebase, their development streams until you complete or cancel their deliver operations.

Finding WorkThat is Ready to Be Delivered

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**. The **Find Posted Deliveries** dialog box appears, as shown in Figure 26, and lists all development streams within the project that contain deliver operations in the posted state.

Figure 26 Find Posted Deliveries Dialog Box



Completing Remote Deliver Operations

To complete remote deliver operations for a development stream:

1. Select the development stream from the list in the **Find Posted Deliveries** dialog box.

2. Click **Deliver**. The **Deliver** dialog box appears. Click **Resume** to resume the deliver operation. Click **Cancel** to cancel the deliver operation. See *Developing Software with ClearCase* for details on completing the deliver operation.

Undoing a Deliver Operation

In addition to the remote deliver scenario, there is another case where you may need to help developers with their deliver operations. At any time before completing the deliver operation, developers can back out of the deliver operation and undo any changes made during the operation. However, if developers check in their versions to the integration view, they cannot easily undo the changes. When this happens, you may need to remove the checked in versions by using the **cleartool rmver –xhlink** command.

NOTE: The **rmver** command erases part of your organization's 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 *ClearCase Reference Manual* for details.

Note that 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.

6.3 Creating a New Baseline

As developers deliver work to the integration stream, it is important that you frequently make new baselines that record the changes. Developers can then rebase to the new baselines and stay current with each other's changes.

Locking the Integration Stream

Before you make a new baseline, lock the integration stream to prevent developers from delivering work. This ensures that you are dealing with a static set of files. To lock the integration stream:

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

Verifying That the Code Base Is Stable

After you lock the integration stream, we recommend that you build and test the project's 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 *Building Software with ClearCase*. Because you lock the integration stream when you build and test in it, we recommend that you use a separate development stream for extensive testing of new baselines. Perform only quick validation tests in the integration stream so that it is not locked for an extended period of time.

See *Testing the Baseline* on page 77 for information about using a development stream for testing new baselines.

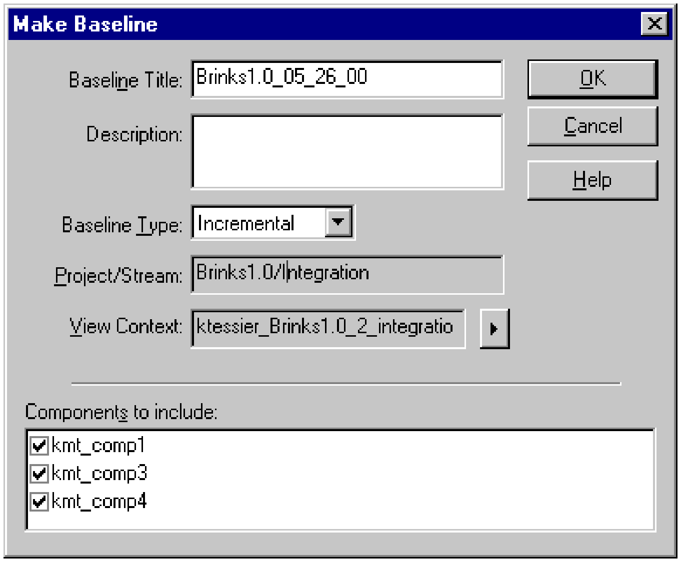
Making the New Baseline

To make a new baseline:

1. In the Project Explorer, select the project's integration stream.

2. Click **Tools>Make Baseline**. The **Make Baseline** dialog box appears, as shown in Figure 27.

Figure 27 Make Baseline Dialog Box



3. Enter a name in the **Baseline Title** box. By default, ClearCase names the baseline by appending the date to the project's name.

4. Choose the type of baseline to create.

   An *incremental baseline* is a baseline that ClearCase creates 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 ClearCase creates by recording all versions below the component's root directory.

Generally, incremental baselines are faster to create than full baselines; however, ClearCase can look up the contents of a full baseline faster than it can look up the contents of an incremental baseline.

5. Specify which components to include in the baseline. By default, ClearCase applies the baseline to all project components. If a component has not changed since the current baseline, ClearCase does not create a new baseline for it.

Making a Baseline For a Set of Activities

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. To do so, use the **cleartool mkbl** command and specify the **activities** parameter. See the **mkbl** page in *ClearCase Reference Manual* for details.

Unlocking the Integration Stream

After you create a new baseline, unlock the integration stream so that developers can resume delivering work to the integration stream. To unlock the integration stream:

1. In the Project Explorer, select the integration stream.
2. Click **File>Properties** to display the integration stream's property sheet.
3. Click the **Lock** tab.
4. Click **Unlocked** and then click **OK**.

6.4 Testing the Baseline

To avoid locking the integration stream for an extended period of time, we recommend that you 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 69 for information on creating a development stream.

After you create a new baseline and verify that it builds and passes an initial validation test in the integration stream, rebase the development stream:

1. In the Project Explorer, select the development stream and click **Tools>Rebase Stream**.

   The **Rebase Stream Preview** dialog box appears.

2. By default, ClearCase rebases your development stream 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 **Change**.

   The **Change Rebase Configuration** dialog box appears.

3. Select a component that contains a baseline you want to test. Click **Change**.

   The **Change Baseline** dialog box appears, 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** dialog box and repeat the process. When you finish selecting baselines, click **OK** to dismiss the **Change Rebase Configuration** dialog box.

6. Click **OK** in the **Rebase Stream Preview** dialog box to continue the rebase operation. See online help or *Developing Software with ClearCase* for details on rebasing a development stream. When you finish rebasing the development stream, you are ready to begin testing the new baselines.

Fixing Problems

If you discover a problem with a baseline while testing it, fix the affected files and deliver the changes to the integration stream as follows:

1. From the development view attached to the development stream, check out the files you need to fix. When you check out a file, you need to specify an activity.

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, deliver the activity to the integration stream.

5. In the Project Explorer, make a new baseline that includes the fixes you delivered plus changes that other developers have delivered since you created the last baseline. See *Creating a New Baseline* on page 75.

6.5 Promoting or Demoting the Baseline

As work on your project progresses, and the quality and stability of the components improve, change the baseline's promotion level attribute to reflect a level of testing that the baseline has passed.

To promote a baseline's promotion level to the level specified for recommended baselines:

1. In the Project Explorer, select the integration stream.
2. Click **Tools>Recommend Baselines**.

To change a baseline's promotion level to something other than the level specified for recommended baselines:

1. In the Project Explorer, right-click the project's integration stream to display its shortcut menu. Click **Properties** to open the integration stream's **Properties** dialog box.
2. Click the **Baselines** tab.
3. In the **Components** list, select the component that contains the baseline you want to promote. In the **Baselines** list, select the baseline. Click **Properties**. The baseline's **Properties** dialog box appears.
4. Click the arrow in the **Promotion Level** list to display all available promotion levels. Select the new promotion level.

On occasion, you may need to demote a baseline by changing its promotion level to one that is lower in the promotion level order. For example, suppose that after you create a new baseline, you discover that it contains a major bug. To prevent developers from introducing this bug to their development streams by rebasing, you can demote the baseline to a Rejected level.

6.6 Tracking the Project

ClearCase provides several tools to help you track the progress of your project. This section describes how to use those tools.

Comparing Baselines

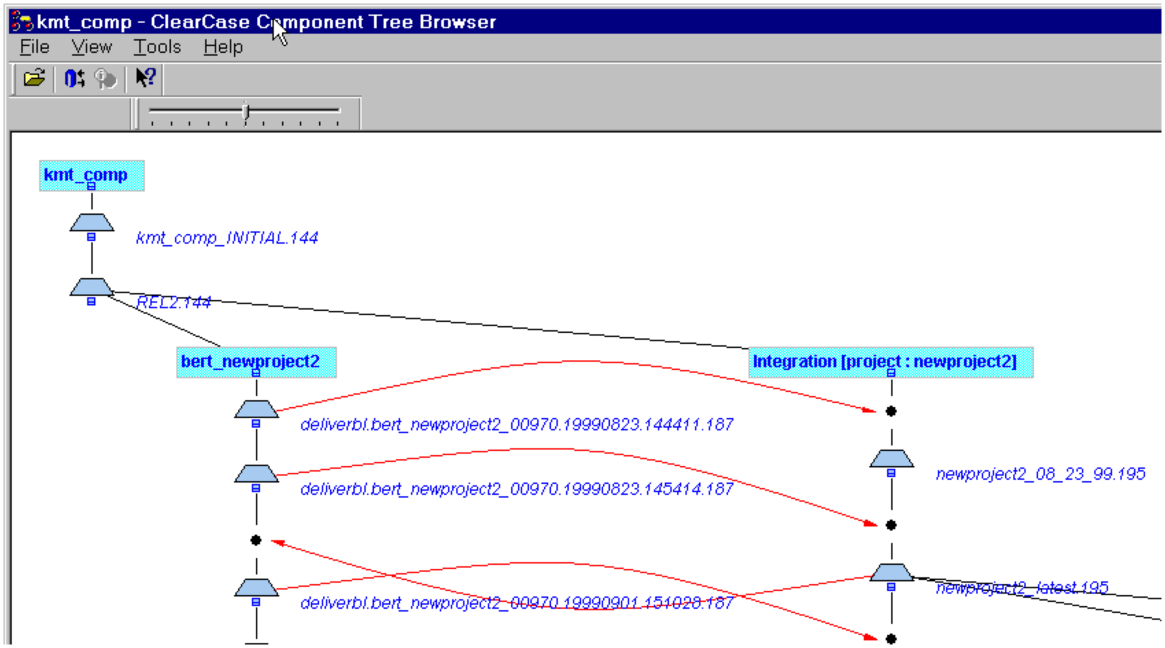
The ClearCase Component Tree Browser is a GUI that displays the baseline history of a component. You can use it to compare the contents of two baselines. To start the Component Tree Browser:

1. Start the Project Explorer, and navigate to the component whose baseline history you want to see.

2. Right-click the component to display its shortcut menu. Select **Browse Baselines**.

The Component Tree Browser appears, as shown in Figure 28.

Figure 28 ClearCase Component Tree Browser



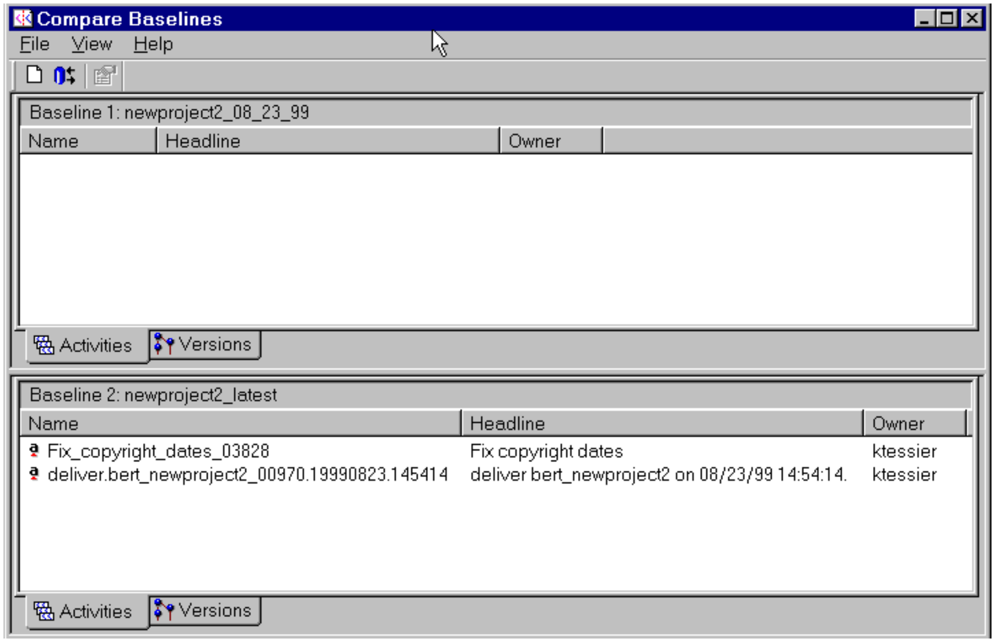
The Component Tree Browser shows the lines of development for the component and each stream that uses the component. In Figure 28, **kmt_comp_INITIAL.144** is the initial baseline that was created when the project manager created the **kmt_comp** component. The **.144** is a unique identifier that ClearCase appends to the baseline. **REL2.144** is the first baseline that the project manager created after creating the component. It is the *foundation baseline* for the integration stream and the development stream named **bert_newproject2**. When Bert joined the project, ClearCase populated his development work area with the contents of the **REL2.144** baseline.

The **deliverbl.bert_newproject2_***n* entries are baselines that ClearCase creates in the development stream during deliver operations. The integration arrows from the development stream to the integration stream represent deliver operations. The **newproject2_08_23_99.195** baseline includes the work from the first deliver operation. The **newproject2_latest.195** baseline includes the work from the second deliver operation.

The integration arrow from the integration stream to the development stream represents a rebase operation.

To compare two baselines, select a baseline by clicking its icon. Then click **Tools>Compare>with Another Baseline**. Click the second baseline's icon. The Compare Baselines window appears, as shown in Figure 29. Alternatively, you can click **Tools>Compare>with Previous Baseline** to compare a baseline with its immediate predecessor.

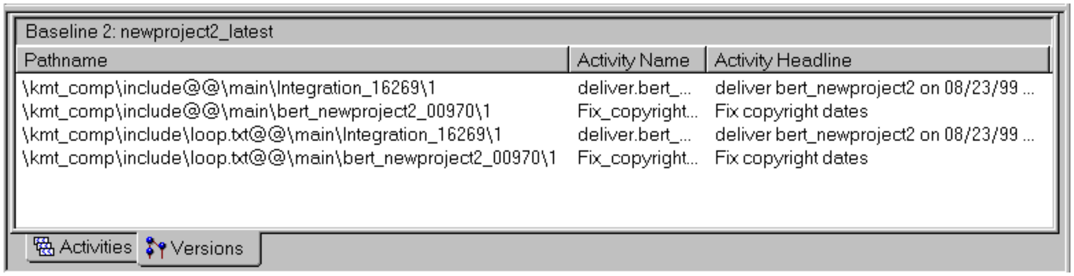
Figure 29 Comparing Baselines by Activity



The Compare Baselines window in Figure 29 shows the results of a comparison of the **newproject2_08_23_99.195** and **newproject2_latest.195** baselines. The more recent baseline contains the **Fix copyright dates** activity. The Compare Baselines window also lists the integration activity that ClearCase created during the deliver operation.

To see the change sets associated with the activities, click **Versions**. Figure 30 shows the versions associated with the **Fix copyright dates** and integration activities.

Figure 30 Comparing Baselines by Version



Querying ClearQuest User Databases

If you use the UCM-ClearQuest integration, you can use ClearQuest queries to retrieve information about the state of your project. When you create or upgrade a ClearQuest user database to use a UCM-enabled schema, the integration installs six queries in two subfolders of the Public Queries folder in the user database's 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 3 lists and describes the queries.

Table 3 Queries in 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.

Table 3 Queries in UCM-Enabled Schema

Query	Description
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 ClearQuest user database.
UCMCustomQuery1	This query is not intended to be used 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 also create your own queries by clicking **Query>New Query** within the ClearQuest client. In the **Choose a record type** dialog box that appears, select **All_UCM_Activities** if you want the query to search all UCM-enabled record types.

Using ClearCase Reports

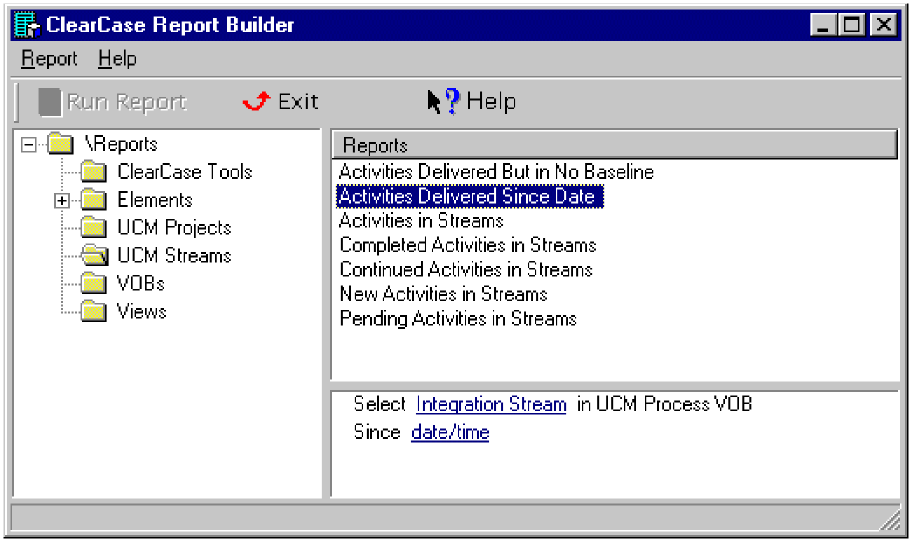
The ClearCase Reports applications (Report Builder and Report Viewer) allow you to generate and view reports specific to your project environment. Use the Report Builder to select and define a report's parameters. Use the Report Viewer to see the report output. To start the ClearCase Report Builder, click **Start>Programs>ClearCase Administration>ClearCase Report Builder**.

The ClearCase 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 example, in Figure 31, with the **Activities Delivered Since Date** report selected, the Report Builder prompts for the name of an integration stream and a date.

For details on using the Report Builder and the Report Viewer, see their online help.

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 ClearCase Reports Programming Interface allows you to customize report procedures. For details on doing so, see Appendix C, *Customizing ClearCase Reports*.

Figure 31 ClearCase Report Builder



6.7 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.

To delete a project, stream, component, or activity, select the object in the Project Explorer, and click **File>Delete**. To delete a baseline, use the **cleartool rmbl** command.

Projects

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 *ClearCase Reference Manual*.

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 *ClearCase Reference Manual*.

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 component's initial baseline does not serve as a *foundation baseline* for another stream.

For more information on removing components, see the **rmcomp** reference page in *ClearCase Reference Manual*.

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's 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 on removing baselines, see the **rmbl** reference page in *ClearCase Reference Manual*.

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 on removing activities, see the **rmactivity** reference page in *ClearCase Reference Manual*.

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.

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 you have made obsolete, click **View>Show Obsolete Items** in the Project Explorer.

7 Managing Parallel Releases of Multiple Projects

The previous chapters describe how to 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. This chapter describes how to accomplish that merging in two common scenarios:

- Managing a current project and a follow-on project simultaneously
- Incorporating a patch release into a new release of the project

This chapter also describes other scenarios in which you can use these merging techniques between projects.

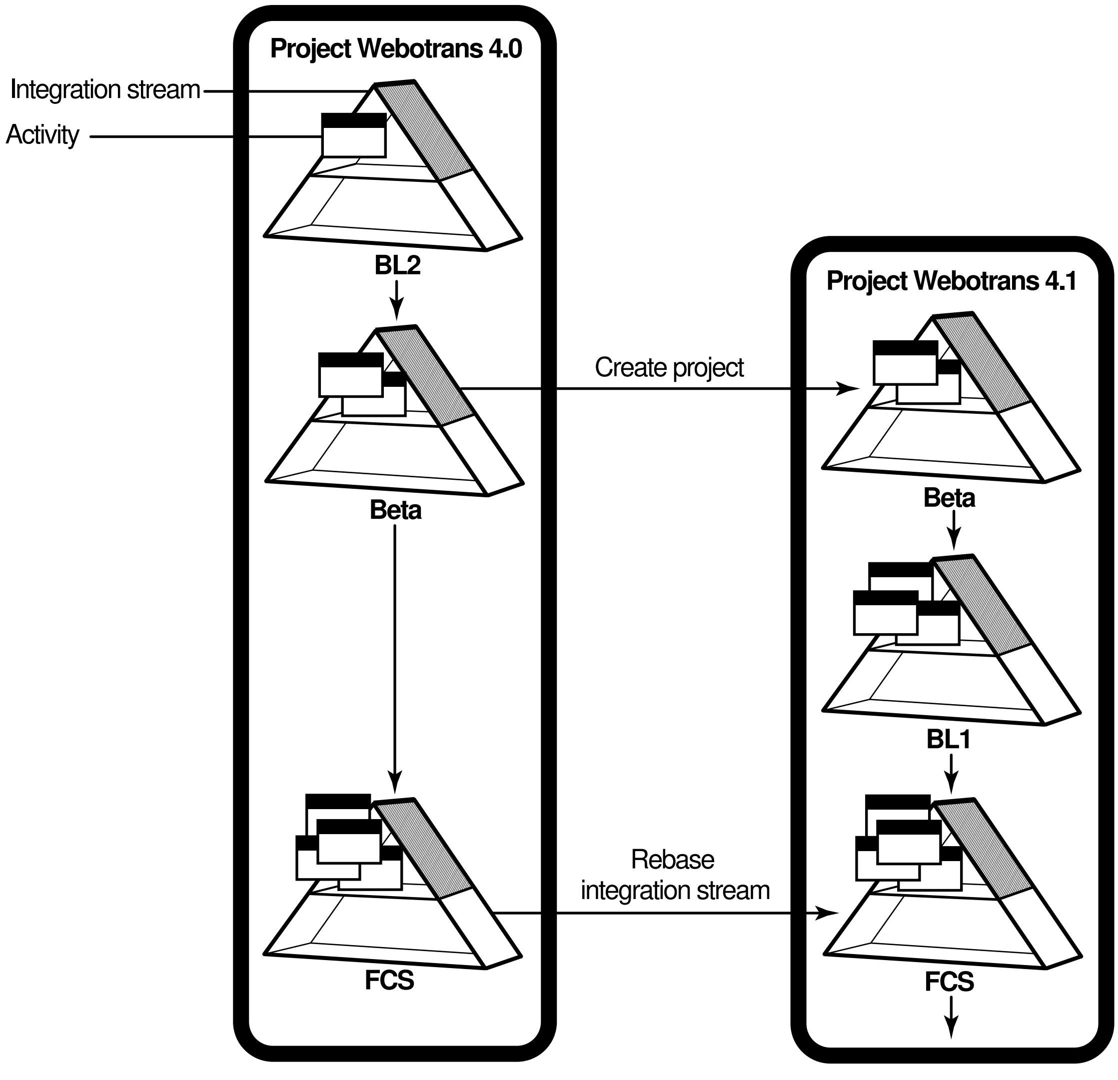
7.1 Managing a Current Project and a Follow-on Project Simultaneously

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.

Example

Figure 32 illustrates the flow of a current project, Webotrans 4.0, and a follow-on project, Webotrans 4.1.

Figure 32 Managing a Follow-on Release



In this example:

- 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 project managers continued to create new baselines that incorporate those changes.
- When the 4.0 team completed its work, the project manager created the final baselines, named FCS. The 4.1 project manager then rebased the 4.1 integration stream to the FCS baselines.

Performing Interproject Rebase Operations

To rebase an integration stream to a set of baselines in another project's integration stream:

1. Navigate to an integration view attached to the integration stream that you want to rebase.
2. For each component, issue the **cleartool rebase** command, specifying the component's baseline. For example:

   ```
   % cleartool rebase –baseline FCS.195 –gmerge
   Changed config spec for view "webotrans4.1_integration" to reflect its
   stream's new configuration.
   Build and test are necessary to ensure that the merges were completed
   correctly.

   When build and test are confirmed, run "cleartool rebase —resume —
   complete".
   ```

3. ClearCase merges nonconflicting changes automatically. You must resolve the changes that ClearCase cannot merge automatically. The **–gmerge** option directs ClearCase to start its Diff Merge graphical tool to help you resolve conflicting changes. For details on using Diff Merge, see the Diff Merge online help and *Developing Software with ClearCase*.
4. When you finish the merge, build and test the changes before completing the rebase operation.
5. Complete the rebase operation. For example:

   ```
   % cleartool rebase –resume –complete
   ```

In the example shown above, **FCS.195** is the full name of the baseline for one of the components in the Webotrans 4.0 integration stream. To determine a baseline's full name:

1. In the Project Explorer, select the integration stream.
2. Click **File>Properties**. The integration stream's property sheet appears.
3. Click the **Baselines** tab.
4. In the **Components** list, select the component that contains the desired baseline.
5. In the **Baselines** list, select the root name of the baseline. Click **Properties**. The baseline's property sheet appears. The **Name** box identifies the full baseline name.

Note that you can rebase your project's integration stream only if the baseline to which you are rebasing is a successor of your integration stream's current *foundation baseline*. In the above example, the FCS baseline is a successor to the Beta baseline, which is the current foundation baseline for the Webotrans 4.1 integration stream.

7.2 Incorporating a Patch Release into a New Version of the Project

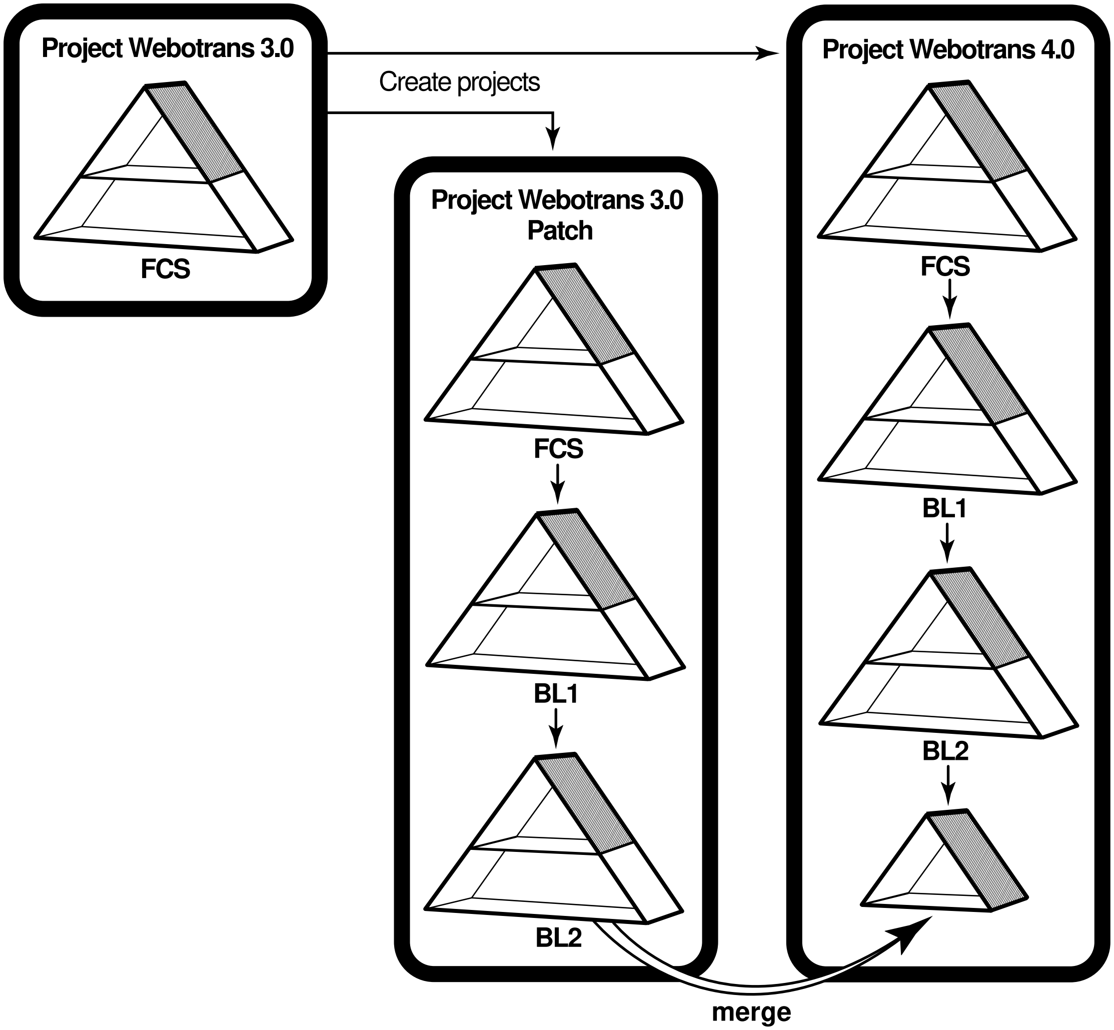
Another common parallel development scenario involves working on a patch release and a new release of a project at the same time. This section describes this scenario.

Example

Figure 33 illustrates the flow of a patch release and a new release. In this example:

- 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 project managers creating baselines periodically.
- The developers working on the 3.0 Patch project finish their work, and the project manager incorporates the final changes in the BL2 baseline. The project manager then needs to merge those changes from the 3.0 Patch integration stream to the 4.0 integration stream so that the 4.0 project contains the fix.

Figure 33 Incorporating a Patch Release



Merging Work to Another Project

UCM does not support interproject deliver operations. However, you can simulate a deliver operation by running a script such as the one shown here, which uses base ClearCase functionality to merge changes.

```
# Sample Perl script for delivering contents of one UCM project to another,
# or to a nonUCM project. Run this script while set to the integration
# view of the destination project.
#
# 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 integration stream. For details on **findmerge**, see the **findmerge** reference page.

We recommend that you add error handling and other logic appropriate for your site to this script before using it.

7.3 Additional Merging Scenarios

This section describes two additional scenarios for which you may want to use a script similar to the one shown in *Merging Work to Another Project* on page 92.

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 project, and the rest of the team works in base ClearCase. If you are a longtime 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 project's integration stream to the branch that serves as the integration branch for the system. To do so, use a script similar to the one shown in *Merging Work to Another Project* on page 92.

Merging to a System Project

If you have a very large system, you may decide to use multiple projects with each project team working on a different part of the system. In this scenario, you would typically create an additional project to serve as the system project. The system project is the integration location for the system. When project managers for each of the projects create stable baselines, the system's project manager merges those baselines to the system project's integration stream.

To do this merging, use the script as shown in *Merging Work to Another Project* on page 92.

Part 2: Working in Base ClearCase

The following chapters describe how to use base ClearCase features to set up and manage a customized development environment for your project team.

8

Managing Projects in Base ClearCase

As a project manager, you are responsible for planning, staffing, and managing the technical aspects of a software development project. You decide what will be worked on, assign work to the project's team members, 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.

Base ClearCase offers many features 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 13, *Using ClearCase Throughout the Development Cycle*, follows a project throughout the development cycle to show how you can use ClearCase.

Before reading this part of the manual, read *Developing Software with ClearCase* to become familiar with the concepts of VOBs, views, and config specs.

8.1 Setting Up the Project

This section describes the planning and setup work you need to do before development begins.

Creating and Populating VOBs

If your project is migrating to ClearCase 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 ClearCase administrator, he or she may be responsible for creating and maintaining VOBs, but not for importing data into them.

Administering ClearCase includes detailed information on these topics.

Planning a Branching Strategy

ClearCase uses branches 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. ClearCase allows you to create complex branch hierarchies. See Figure 1 for an illustration of a multilevel branching hierarchy. 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 9, *Defining Project Views*, 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 ClearCase. But there are also similarities that reflect common adherence to best practices. Some of the more common branch types and uses are presented here.

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 don't 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's 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 ClearCase MultiSite

Branches are particularly important when your team works in VOBs that have been replicated to other sites with the 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 *Policy: Certain Branches Are Shared Among MultiSite Sites* on page 138.

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 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 is not recommended because doing so can degrade system performance.

➤ Use the ClearCase View Profiles mechanism to configure views that the project team will use. The View Profile tools promote a specific model for the effective use of ClearCase. Project teams that adhere to this model can take advantage of several areas of automated support, significantly simplifying their ability to exploit some of the more advanced features of ClearCase. For more information on View Profiles, see the online help.

➤ To ensure that all team members configure their views the same way, you can create files that contain standard config specs. For example:

  ➢ **\\vulcan\c\public\c_specs\abc** contains the ABC team's config spec

  ➢ **\\vulcan\c\public\c_specs\xyz** contains the XYZ team's 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 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 machine hosting the view (**platinum_V4.0_int**).

8.2 Implementing Development Policies

To enforce development policies, you can create 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 that ClearCase captures 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 lifecycle. 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 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 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 ClearCase event records
- Initiating a build and/or source code analysis whenever particular objects are modified

For more information on these mechanisms, see Chapter 10, *Implementing Project Development Policies*.

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 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 manual *Administering ClearCase* has more information about creating and using global types.

Generating Reports

ClearCase creates and stores an event record each time an element is modified or merged. Many 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.

Chapter 10, *Implementing Project Development Policies*, provides more detail on using event records and metadata to implement project policies. Event records and other metadata can also be useful if you need to generate reports on activities managed by ClearCase (for example, the complete history of changes to an element). ClearCase provides a variety of report-generation tools. For more information on this topic, see the **fmt_ccase** reference page in the *ClearCase Reference Manual*.

8.3 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 and/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.

There are other scenarios as well. Chapter 11, *Integrating Changes*, describes merging in detail.

9 Defining Project Views

This chapter explains how config specs work and provides sample config specs useful for project development work, for nondevelopment tasks such as monitoring progress and doing research, and for running project builds. It also explains how to share config specs between Windows and UNIX systems.

9.1 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 ClearCase uses 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. ClearCase searches the version tree of each element for the first version that matches the first rule in the config spec. If no versions match the first rule, ClearCase searches for a version that matches the second rule. 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.

9.2 Default Config Spec

This config spec defines a dynamic configuration, which selects changes made on the **main** branch of every element throughout the entire source tree, by any developer:

```
(1)  element * CHECKEDOUT
(2)  element * \main\LATEST
```

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 contents of file **default_config_spec** (located in *ccase-home-dir*) become the new view's config spec.

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. As soon as 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. See *Developing Software with ClearCase* for details on creating snapshot views.

The Standard Configuration Rules

The two configuration rules in the default config spec appear in many of this chapter's 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 this warning:

```
Z:\vob_pr3\src> 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 cannot "see" 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.

9.3 Config Spec Include Files

ClearCase supports an include file facility that 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 **\\main_svr\public\c_specs\major.csp**. Each developer then needs a one-line config spec:

(1)

```
include \\main_svr\public\c_specs\major.csp
```

NOTE: If you are sharing config specs between UNIX and Windows NT computers where the VOB-tags are different, you must have two sources, or you must store the config spec in a UNIX directory that is accessible from both machines.

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. You can use this command to reconfigure your view with the modified config spec:

`Z:\>` **cleartool setcs –current**

9.4 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 **\monet**, which has this structure:

\monet	*(VOB-tag)*
src	*(C language source files)*
include	*(C language header files)*
lib	*(project's libraries)*

For the purposes of this chapter, suppose that the **lib** directory has this substructure:

lib

libcalc.lib	*(checked-in staged version of library)*
libcmd.lib	*(checked-in staged version of library)*
libparse.lib	*(checked-in staged version of library)*
libpub.lib	*(checked-in staged version of library)*
libaux1.lib	*(checked-in staged version of library)*
libaux2.lib	*(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 **\monet\lib**. 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 **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

NOTE: Config specs allow *absolute VOB pathnames*—absolute pathnames that begin with a *VOB-tag* but do not include drive letter or view-tag prefixes. This form of pathname is required to specify VOB elements without regard for current drive assignments or active views. For example:

`\vob_gopher\lib\*`	*(absolute VOB pathname, where \vob_gopher is the VOB-tag)*
`\monet\src\*`	*(absolute VOB pathname, where \monet is the VOB-tag)*
`Z:\monet\src\*`	*(drive-specific pathname; not recommended)*
`M:\myview\vob_gopher\lib\*`	*(view-extended pathname; not recommended)*

9.5 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).

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

Sometimes, other developers 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:

➤ 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.

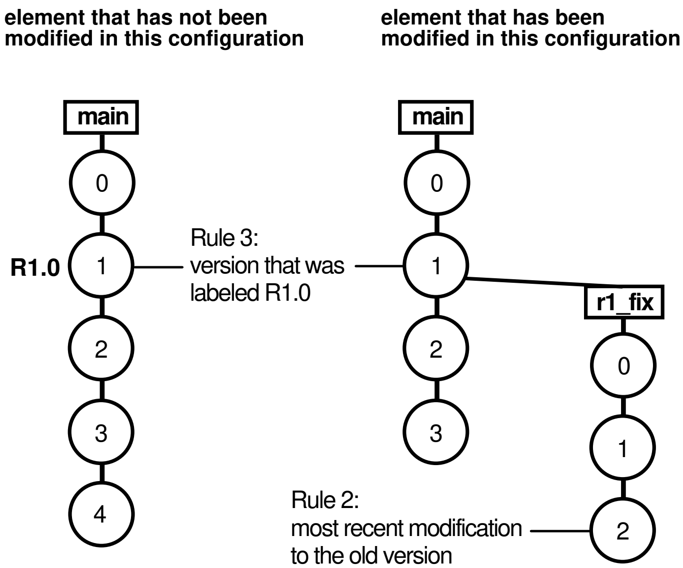
This config spec is efficient: two rules (Rules 2 and 3) configure the appropriate versions of all elements:

➤ 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 34 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 any element's version tree, and at different locations in the version trees of different elements.

Figure 34 Making a Change to an Old Version



Omitting the \main\LATEST Rule

The config spec in *View to Modify an Old Configuration* on page 110 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.

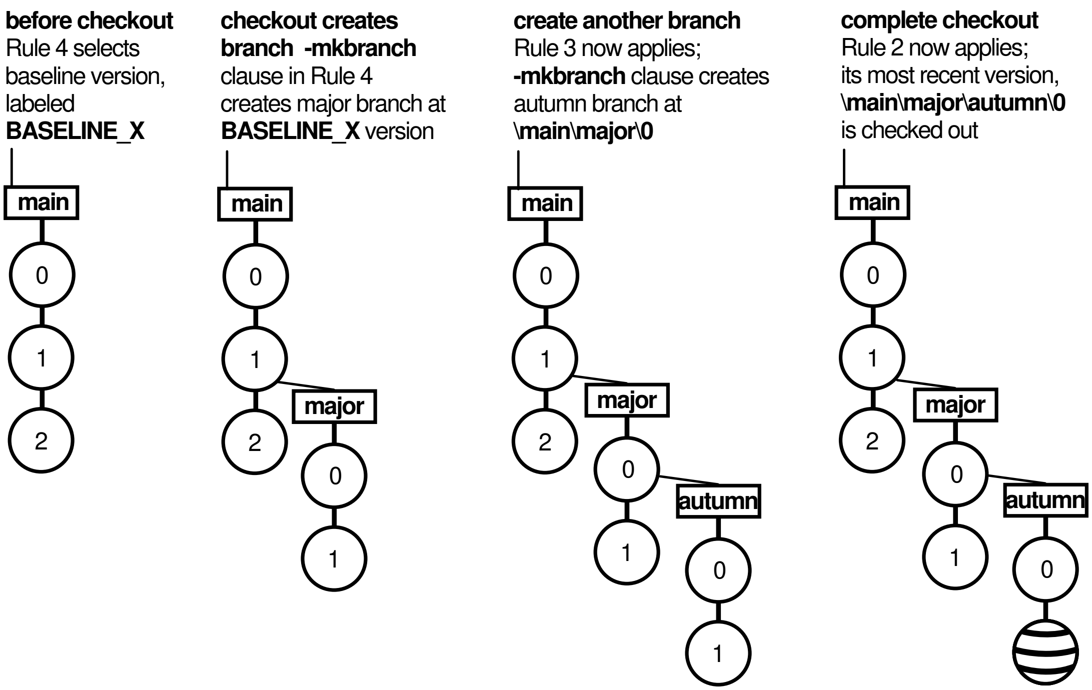
```
(1)   element * CHECKEDOUT
(2)   element * ...\major\autumn\LATEST
(3)   element * ...\major\LATEST -mkbranch autumn
(4)   element * BASELINE_X -mkbranch major
(5)   element * \main\LATEST -mkbranch major
```

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 35 shows what happens in such a view when you check out an element that has not been modified since the baseline.

Figure 35 Multiple-Level Auto-Make-Branch



For more on 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, **\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 **\monet\src*** restricts matching to only one VOB.

This config spec can be extended easily with additional rules that allow additional areas of the source tree to be modified.

9.6 Views to Monitor Project Status

The config specs presented here 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\* ...\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 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).

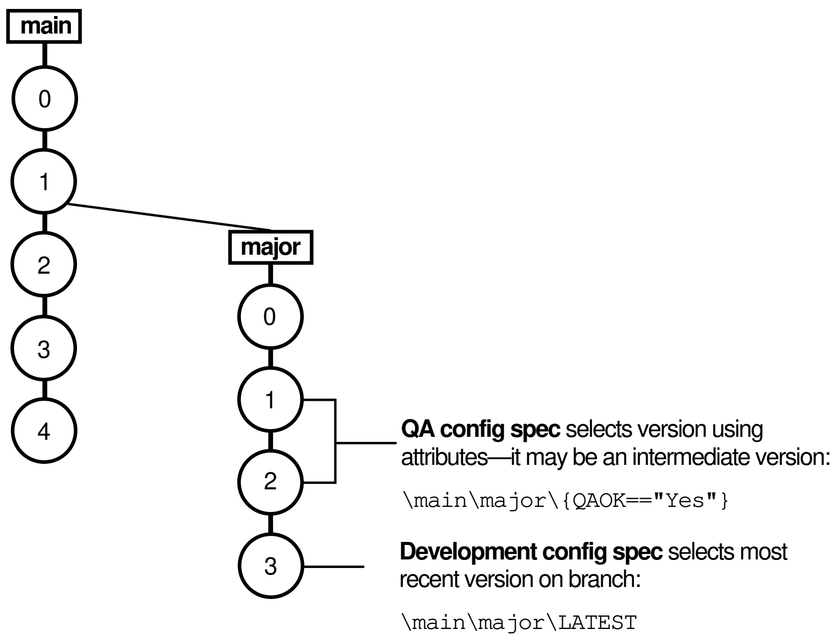
NOTE: Rule 1 on this config spec does not provide a match if an element has a **major** branch, but no version on the branch has a **QAOK** attribute. This command can locate the branches that do not have this attribute:

cleartool find . –branch "{brtype(major) && ! attype_sub(QAOK)}" –print

The **attype_sub** primitive searches for attributes on an element's versions and branches, as well as on the element itself.

This scheme allows the QA team to monitor the progress of the rest of the group. The development config spec always selects the most recent version on the **major** branch, but the QA config spec may select an intermediate version (Figure 36).

Figure 36 Development Config Spec vs. QA Config Spec



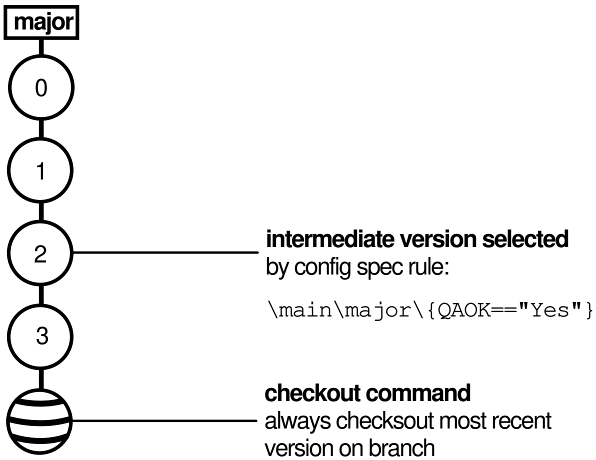
Pitfalls of Using This Configuration for Development

You may be tempted to add a **CHECKEDOUT** rule to the above config spec, turning the QA configuration into a 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 37.)

Figure 37 Checking Out a Branch of an Element



Performing a checkout in this view checks out version **\main\major\3**, not version **\main\major\2**:

```
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 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 this problem by modifying the config spec and creating another branching level at the version that the attribute selects. This is the new 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
```

View That Shows Changes of One Developer

This config spec makes it easy to examine all changes 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
```

NOTE: 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

This 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.

NOTE: 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

This config spec defines a frozen configuration in a slightly different way than the previous one:

(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 elements from being checked out or created.

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.

9.7 Views for Project Builds

The config specs in this section are useful for running the various types of builds 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, they can also provide up-to-date versions of lower-level software (libraries, utility programs, and so on).

Suppose that every night, a script does the following:

- Builds libraries in various subdirectories of **\monet\lib**
- Checks them in as DO versions in the library staging area, **\monet\lib**
- Labels the versions **LAST_NIGHT**

You can use this config spec if you want to use the libraries produced by the nightly builds:

```
(1)  element * CHECKEDOUT
(2)  element lib\*.lib LAST_NIGHT
(3)  element lib\*.lib 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 night's 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 described above uses version labels to select particular versions of libraries. For more flexibility, the **LAST_NIGHT** version of some libraries may be selected, the **R2_BL2** version of others, and the most recent version of still others:

```
(1)   element * CHECKEDOUT
(2a)  element lib\libcmd.lib LAST_NIGHT
(2b)  element lib\libparse.lib LAST_NIGHT
(3a)  element lib\libcalc.lib R2_BL2
(3b)  element lib\*.lib \main\LATEST
(4)   element * \main\LATEST
```

(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. For example, **lib_selector** attributes can take values such as `experimental`, `stable`, and `released`. A config spec can mix and match library versions like this:

```
(1)   element * CHECKEDOUT
(2)   element lib\libcmd.lib {lib_selector=="experimental"}
(3)   element lib\libcalc.lib {lib_selector=="experimental"}
(4)   element lib\libparse.lib {lib_selector=="stable"}
(5)   element lib\*.lib {lib_selector=="released"}
(6)   element * \main\LATEST
```

View That Selects Versions of Application Subsystems

This config spec selects specific versions of the application's subsystems:

```
(1)   element * CHECKEDOUT
(2)   element \monet\lib\... R2_BL1
(3)   element \monet\include\... R2_BL2
(4)   element \monet\src\... \main\LATEST
(5)   element * \main\LATEST
```

In this situation, a developer is making changes to the application's 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

This config spec defines a view that selects only enough files required to rebuild a particular program or examine its sources:

(1)
```
element * -config \monet\src\monet
```

All elements that were not involved in the build of **monet** appear in the output of 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 pathname that includes a *DO-ID*:

(1)
```
element * -config \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's config record does not list the version of the makefile that was used to build it. Instead, the CR includes a copy of the build script itself. (Why? When a new version of the makefile is created with a revision to one target's 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 **omake**), 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's dependency list:

```
monet.exe: $(MAKEFILE) monet.obj ...
     link -out:monet.exe monet.obj ...
```

clearmake always records the versions of explicit dependencies in the CR.

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 117 or *View to Modify an Old Configuration* on page 110 to configure a view for building. You can also use the special target **.DEPENDENCY_IGNORED_FOR_REUSE**; for more information, see *Including a Makefile Version in a Configuration Record* in *Building Software with ClearCase*.

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-up branch type is **r1_fix,** this modified config spec reconfigures the view for performing the fix:

```
element * CHECKEDOUT
element * ...\r1_fix\LATEST
element * -config \monet\src\monet -mkbranch r1_fix
element * \main\LATEST -mkbranch r1_fix
```

Selecting Versions That Built a Set of Programs

It is easy to expand the config spec that selects only enough files required to rebuild a particular program to configure a view with the sources used to build a set of programs, rather than a single program:

```
element * -config \monet\src\monet
element * -config \monet\src\xmonet
element * -config \monet\src\monet_conf
```

There can be version conflicts in such configurations, however. 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.

You can modify this config spec as described in *Fixing Bugs in the Program on page 122*, to change the build configuration to a development configuration.

9.8 Sharing Config Specs Between UNIX and Windows

You can, in principle, share config specs between UNIX and Windows systems. That is, users on both systems, using views whose storage directories reside on either platform, can set and edit the same set of config specs.

We recommend that you 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 (**/**), not backslashes (****) in pathnames.
- Use relative, not full, pathnames whenever possible, and do not use VOB-tags in pathnames. You can ignore this restriction if your UNIX and Windows VOB-tags both use single, identical pathname components that differ only in their leading slash characters—**\src** and **/src**, for example.
- Always edit and set config specs on UNIX.

The following sections describe these requirements in detail.

Pathname Separators

When writing config specs to be shared by Windows and UNIX computers, you must use slash (**/**), not backslash (****), as the pathname separator. ClearCase on UNIX recognizes slashes only. (Note that **cleartool** recognizes both slashes and backslashes in pathnames; **clearmake** is less flexible. See *clearmake Makefiles and BOS Files* in *Building Software with ClearCase* for more information.)

Pathnames in Config Spec Element Rules

Windows and UNIX network regions often use different VOB-tags to register the same VOBs. Only single-component VOB-tag names, such as **\proj1**, are permitted on Windows computers; multiple-component VOB-tags, such as **/vobs/src**, are common on UNIX.

When VOB-tags differ between regions, any config spec element rules that use full pathnames (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 pathnames in element rules make sense. That is, a config spec with full pathnames 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 *Example* on page 124):

- The config spec's element rules use only relative pathnames, which do not include VOB-tags.
- Shared VOBs are registered with identical, single-component VOB-tags in both Windows and UNIX network regions. (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's *compiled* form is portable. The restriction is that full VOB pathnames 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 ClearCase 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.

Example

This 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** on a UNIX network region, only Windows computers can compile the config spec.

To address the problem, do one of the following:

➤ Use relative pathnames that do not include VOB-tags, for example:

```
element ...\abc_proj_src\*      \main\rel2\LATEST
```

➤ On UNIX, change the VOB-tag so that it has a single component, **/vob_p2**.

10 Implementing Project Development Policies

This chapter presents brief scenarios that show how you can implement and enforce common development policies with ClearCase. The scenarios use various combinations of these functions and metadata:

- Attributes
- Labels
- Branches
- Triggers
- Config specs
- Locks
- Hyperlinks

Sharing Triggers Between UNIX and Windows on page 139 describes how to define triggers for use on UNIX and Windows computers.

10.1 Policy: Good Documentation of Changes Is Required

Each 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. The trigger action script analyzes the user's comment (passed in an environment variable), disallowing unacceptable ones (null or empty comments, for example).

NOTE: When ClearCase fires a trigger, it proceeds based on the success or failure of the trigger operation, as determined by the trigger script's exit code. A **.bat** file returns the exit code of its last command. Preoperation triggers are the only kind of trigger that cause the ClearCase execution to fail.

Trigger Definition:

cleartool mktrtype –element –all –preop checkin ^
–c "must enter descriptive comment" –exec \\neon\scripts\comm_pol.bat CommentPolicy

Trigger Action Script:

```
@echo off
rem comm_pol.bat
rem
rem Check for null comment
rem
if "%CLEARCASE_COMMENT%"=="" copy > NUL:
```

10.2 Policy: All Source Files Require a Progress Indicator

You may want to 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:

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 conscientious 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:

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);
}
```

10.3 Policy: 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 label the versions selected by that view:

`Z:\vob_hw>` **cleartool mklbtype –c "Release 2.0 sources" REL2**

`Z:\vob_hw>` **cleartool mklabel –recurse REL2** *top-level-directory*

Alternatively, you can use the configuration records of the release's derived objects to control the labeling process:

`Z:\vob_hw>` **clearmake vega**

... sometime later, after QA approves the build:
`Z:\vob_hw>` **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:

`Z:\vob_hw>` **cleartool lock –nusers vobadm lbtype:REL2**

The object is locked to all users except those specified with the **–nusers** option, in this case, **vobadm**.

10.4 Policy: Isolate Work on Release Bugs to a Branch

You may want to fix bugs found in the released system on a named bugfix branch, and to begin this work with the exact configuration of versions from that release.

This policy reflects the ClearCase baseline-plus-changes model. First, a label (**REL2,** for example) 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 -mkbranch 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.

10.5 Policy: 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. Some 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 last night's 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"}
```

➤ You can also 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 C:\cspecs\rules_r2
```

10.6 Policy: 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:\my_vob**

10.7 Policy: 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 GUI, an e-mail message to the doc group ensures that these changes are documented.

To enforce this policy, create a trigger type that sends mail, and then attach it to the relevant elements.

Trigger Definition:

cleartool mktrtype –nc -element –postop checkin ^
–exec "ccperl \\neon\scripts\informwriters.pl" InformWriters

```
Created trigger type "InformWriters".
```

Trigger Action Script:

```
use Net::SMTP;

my $smtp = new Net::SMTP 'neon.purpledoc.com';

$smtp->mail('ClearCase Admin');
$smtp->to('ClearCase Admin');
$smtp->to('docgrp');

$smtp->data();
$smtp->datasend("From: ClearCase Admin\n");
$smtp->datasend("To: docgrp\n");
$smtp->datasend("Subject: checkin\n");
$smtp->datasend("\n");

# create variables for pathname/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 \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 \gui_src –type f ^
   –exec "cleartool mktrigger InformWriters %CLEARCASE_PN%"

10.8 Policy: 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 may want to 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:

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, it is monitored.) If a developer attempts to check in a **c_source** file that fails the **appl_met.bat** test, the checkin fails.

NOTE: 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 batch file can be invoked as a trigger action, and developers can also use it manually.

10.9 Policy: 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 "G:\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
```

10.10 Policy: 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 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@\dev DesignDoc@\design

```
Created hyperlink type "DesignDoc".
Created hyperlink type "DesignDoc".
```

The *hyperlink inheritance* feature makes the implementation of requirements tracing easy:

➤ When the source module, **hello.c**, and the design document, **hello_dsn.doc**, are updated, the project manager creates a new hyperlink connecting the two updated versions:

cleartool mkhlink -c "source doc" DesignDoc hello.c \design\hello_dsn.doc
```
Created hyperlink "DesignDoc@90@\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:

version that inherits hyperlink->

version to which -> hyperlink is explicitly attached

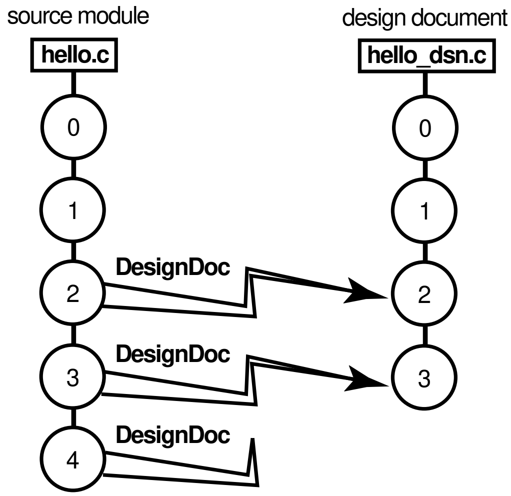
cleartool describe –ihlink DesignDoc hello.c@@\main\2
```
hello.c@@\main\2
  Inherited hyperlinks: DesignDoc@90@\dev
    \dev\hello.c@@\main\1 ->
    \doc\hello_dsn.doc@@\main\1
```

➤ When either the source module or the design document incorporates a significant update, which renders the connection invalid, the project manager creates a *null-ended* hyperlink to sever the connection:

cleartool mkhlink -c "sever connection to design doc" DesignDoc hello.c
```
Created hyperlink "DesignDoc@94@\dev".
```

Figure 38 illustrates the hyperlinks that connect the source file to the design doc.

Figure 38 Requirements Tracing



10.11 Policy: Prevent Use of Certain Commands

To control which users can execute certain commands on ClearCase objects, you can create a pair of trigger types. One of the types controls use of the command on element-related objects, and the other controls 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.

NOTE: You cannot use triggers to prevent a command 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@\dev **
\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 "\dev".
```

10.12 Policy: Certain Branches Are Shared Among MultiSite Sites

If your company uses ClearCase MultiSite to support development at different sites, you must tailor your branching strategy to the needs of the different sites. The standard MultiSite development model is that a replica of the VOB is 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. (See *Introduction to MultiSite* in *ClearCase MultiSite Manual* for more information on MultiSite mastership.)

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). 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.

MultiSite provides two 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 *ClearCase MultiSite Manual*.

➤ The pull model, in which the developer who needs to work on the branch uses the **reqmaster** command to request mastership of the branch.

This model requires the 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 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)

Implementing Requests for Branch Mastership in *ClearCase MultiSite Manual* describes the process of enabling the pull model and a scenario in which developers use the pull model. *Working On a Team* in *Developing Software with ClearCase* describes the procedure developers use to request mastership.

10.13 Sharing Triggers Between UNIX and Windows

You can define triggers that fire correctly on both UNIX and Windows computers. The following sections describe two techniques. With one, you use different pathnames or different scripts; with the other, you use the same script for both platforms.

Using Different Pathnames or Different Scripts

To define a trigger that fires on UNIX, Windows, or both, and that uses different path names to point to the trigger scripts, use the **–execunix** and **–execwin** options with the **mktrtype** command. These options behave the same as **–exec** when fired on the appropriate platform (UNIX or Windows, respectively). On the other platform, they do nothing. This technique allows a single trigger type to use different paths for the same script or to use completely different scripts on UNIX and Windows computers. For example:

cleartool mktrtype –element –all –nc –preop checkin ^
–execunix /public/scripts/precheckin.sh –execwin \\neon\scripts\precheckin.bat ^
pre_ci_trig

On UNIX, only the script **precheckin.sh** runs. On Windows, only **precheckin.bat** runs.

To prevent users on a new platform from bypassing the trigger process, triggers that specify only **–execunix** always fail on Windows. Likewise, triggers that specify only **–execwin** fail on UNIX.

Using the Same Script

To use the same trigger script on both Windows and UNIX platforms, you must use a batch command interpreter that runs on both operating systems. For this purpose, ClearCase includes the **ccperl** program. On Windows, **ccperl** is a version of the Perl program available on UNIX.

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

Notes

➤ To conditionalize script execution based on operating system, use environment variables in Perl scripts.

➤ To collect or display information interactively, you can use the **clearprompt** command.

➤ For more information on using the **–execunix** and **–execwin** options, see the **mktrtype** reference page.

11 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 into any other branch. This chapter describes techniques and scenarios for merging versions of elements and branches. ClearCase includes automated merge facilities for handling almost any scenario.

11.1 How Merging Works

A merge combines the contents of two or more files or directories into a single new file/directory. The ClearCase merge algorithm uses the following files during a merge (see Figure 39):

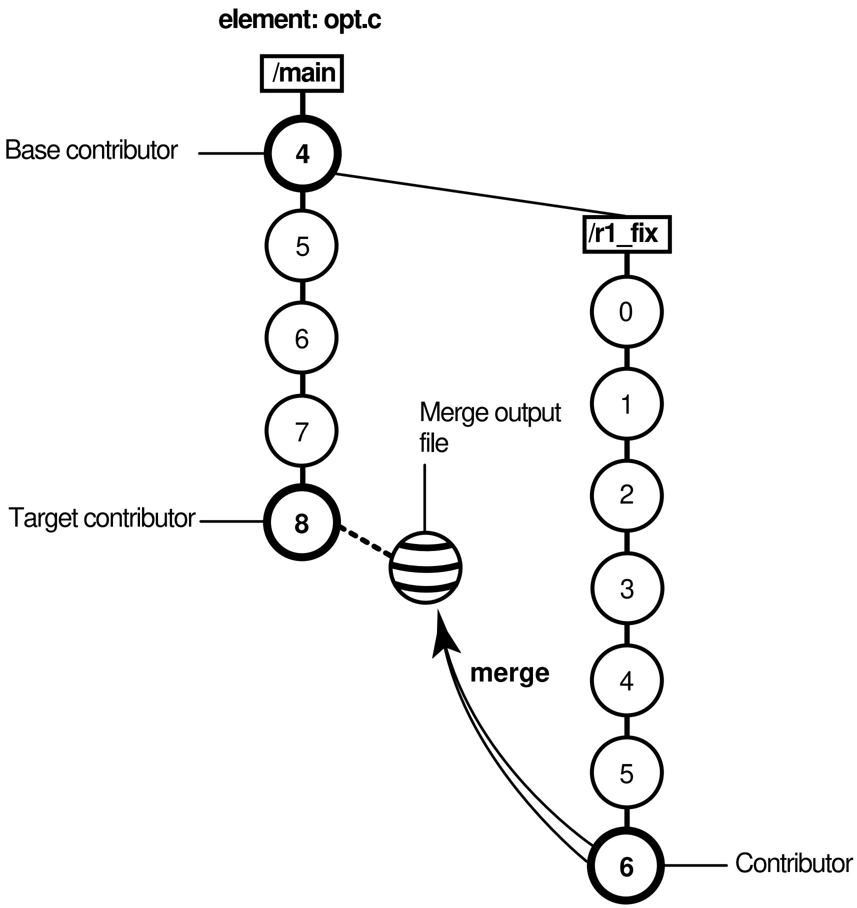
➤ Contributors, which are typically one version from each branch you are merging. (You can merge up to 15 contributors.) You specify which versions are contributors.

➤ 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.) ClearCase 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 39 Versions Involved in a Typical Merge



To merge files and directories, ClearCase takes the following steps:

1. It identifies the base contributor.

2. It compares each contributor against the base contributor. (See Figure 40.)

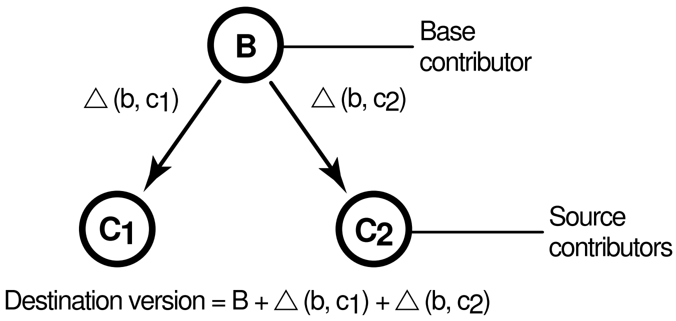
3. For any line that is unchanged between the base contributor and any other contributor, it copies the line to the merge output file.

4. For any line that has changed between the base contributor and one other contributor, it accepts the change in the contributor; depending on how you started the merge operation, ClearCase may copy the change 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, ClearCase requires that you resolve the conflicting difference.

Figure 40 ClearCase Merge Algorithm



To merge versions, you can use the GUI tools, described briefly in the next section, or the command-line interface, described in *Using the Command Line to Merge Elements* on page 144.

Using the GUI to Merge Elements

ClearCase provides three graphical tools to help you merge elements:

- Merge Manager
- Diff Merge
- Version Tree Browser

The Merge Manager manages the process of merging one or more 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

You can start the Merge Manager in several ways:

- Click **Start>Programs>ClearCase>Merge Manager**.
- In ClearCase Explorer, click **Base ClearCase**, and then click **Merge Manager**.

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.

You can start Diff Merge in several ways:

- On the shortcut menu in Windows Explorer, click **Compare**.
- In the Merge Manager, click **Compare**.

The Version Tree Browser displays the version tree for an element. The version tree is useful when merging to do the following::

- 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**.

You can start the Version Tree Browser in several ways:

- Click **Start>Programs>ClearCase>Version Tree Browser**

- On the shortcut menu in Windows Explorer, click **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 *ClearCase Reference Manual*.

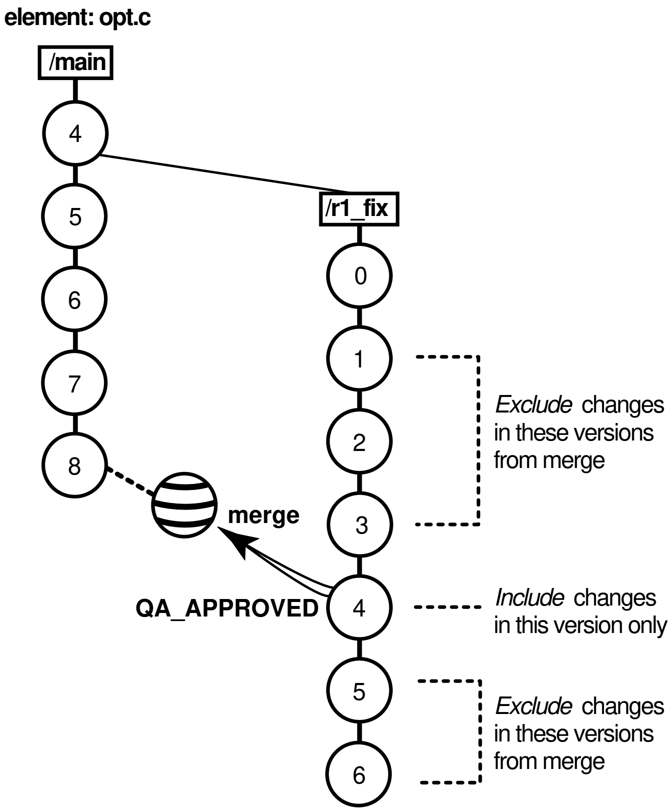
11.2 Common Merge Scenarios

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.

Scenario: Selective Merge from a Subbranch

In this scenario, 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 41.

Figure 41 Selective Merge from a Subbranch



In a view configured with the default config spec, enter these commands to perform the selective merge:

```
Z:\avob> cleartool checkout opt.c
Z:\avob> 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**:

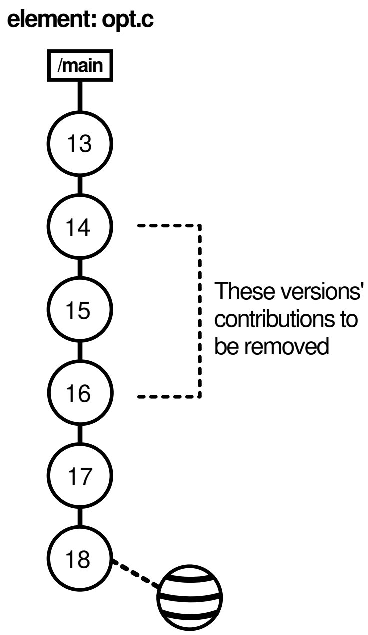
```
Z:\avob> cleartool merge –to opt.c –insert –version \main\r1_fix\2 \main\r1_fix\4
```

No merge arrow is created for a selective merge.

Scenario: Removing the Contributions of Some Versions

A new feature, implemented in versions 14 through 16 on the **main** branch, will not be included in the product. You must remove the changes made in those versions. See Figure 42.

Figure 42 Removing the Contributions of Some Versions



Enter these commands to perform this subtractive merge:

```
Z:\avob> cleartool checkout opt.c
Z:\avob> cleartool merge –to opt.c –delete –version \main\14 \main\16
```

No merge arrow is created for a subtractive merge.

Scenario: 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. It can accommodate the following schemes for isolating the project's work.

All Project Work Is 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:

```
Z:\avob> cd root-of-source-tree
Z:\avob> 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's merge requires user interaction. If the project has made changes in several VOBs, you can perform all the merges at once by specifying several pathnames, 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's work has been done in a view whose view-tag is **goph_vu**. These commands perform the merge:

```
Z:\avob> cd root-of-source-tree
Z:\avob> cleartool findmerge . –ftag goph_vu –merge –graphical
```

NOTE: Working in a single shared view is not recommended because doing so can degrade system performance.

Scenario: Merging a New Release of an Entire Source Tree

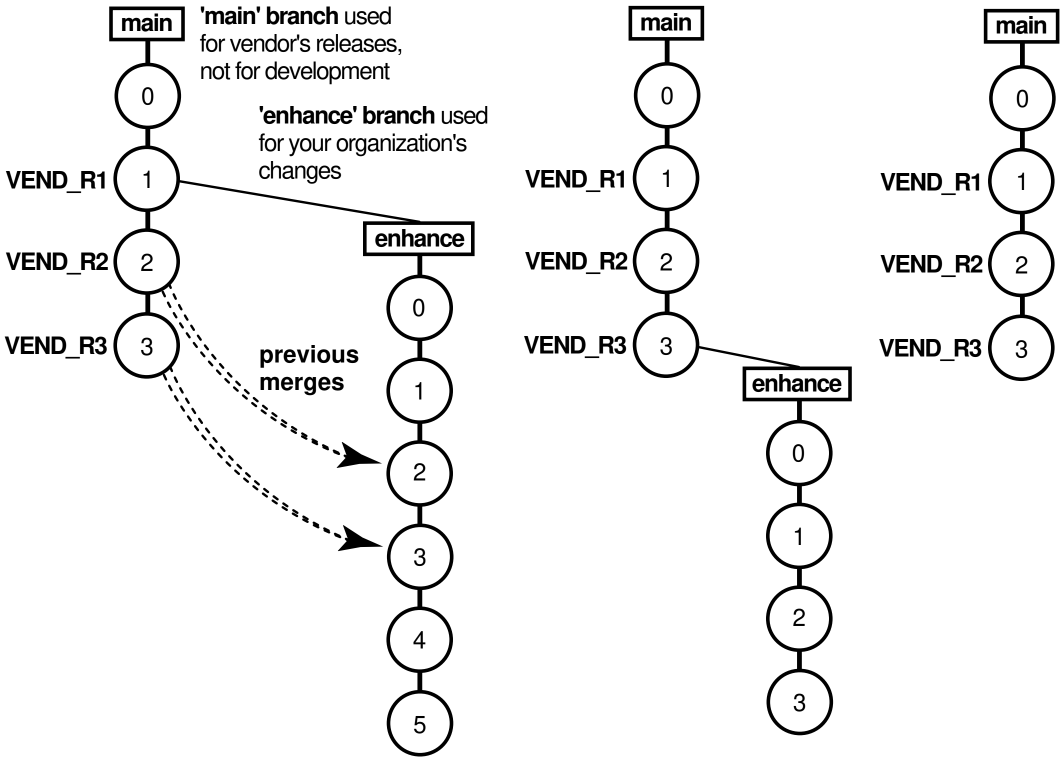
Your team has been using an externally supplied source-code product, maintaining the sources in a VOB. The successive versions supplied by the vendor are checked in to the **main** branch and labeled **VEND_R1**, **VEND_R2**, and **VEND_R3.** Your team's fixes and enhancements are created on subbranch **enhance.** The views that your team works in are configured 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 43 show 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 43 Merging a New Release of an Entire Source Tree



Release 4 has arrived, 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 integration:

1. Load the vendor's Release 4 media into a standard directory tree:

C:\> **cd %TMP%**
C:\> **xcopy** *drive\path options*

The directory tree created is **mathlib_4.0**.

2. As the VOB owner, run **clearexport_ffile**, to create a datafile containing descriptions of the new versions.

```
C:\> cd \mathlib_4.0
C:\mathlib_4.0> clearexport_ffile
  . (lots of output)
  .
```

3. In a view configured with the default config spec, start **clearimport** on the file **clearexport_ffile** created. This creates Release 4 versions on the **main** branches of elements (and creates new elements as needed).

```
C:\mathlib_4.0> net use z: \\view\mainline
The command completed successfully.
C:\mathlib_4.0> z:
Z:\> cd \vob_proj\mathlib
Z:\vob_proj\mathlib> clearimport %TMP%\mathlib_4.0\cvt_data
```

4. Label the new versions:

```
Z:\vob_proj> cleartool mklbtype –c "Release 4 of MathLib sources" VEND_R4
Created label type "VEND_R4".
Z:\vob_proj> cleartool mklabel –recurse VEND_R4  \vob_proj\mathlib
  . (lots of output)
  .
```

5. Set to a view that is configured with your team's config spec and selects the versions on the **enhance** branch:

```
Z:\vob_proj> net use y: \\view\enh_vu
The command completed successfully.
Z:\vob_proj> y:
Y:\vob_proj>
```

6. Merge from the **VEND_R4** configuration to your view:

```
Y:\vob_proj> cleartool findmerge -nback \vob_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.

7. 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 as follows:

```
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.

Scenario: Merging Directory Versions

One of the most powerful features of ClearCase is versioning of directories. Each version of a directory element catalogs a set of file elements and directory elements. In a development project, directories change are 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 from the previous section. Suppose you find that the vendor has made several changes in directory **M:\view1\vob_proj\mathlib\src**:

- File elements **Makefile**, **getcwd.c,** and **fork3.c** have been revised.
- File elements **readln.c** and **get.c** have been deleted.
- A new file element, **newpaths.c**, has been 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: M:\view1\vob_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 then 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 needed. Afterward, you can cancel the directory merges by using the **uncheckout** command on the directories.

11.3 Using Your Own 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:

```
Z:\avob> cleartool checkout nextwhat.c
Checkout comments for "nextwhat.c":
merge enhance branch
.
Checked out "nextwhat.c" from version "\main\1".

Z:\avob> <invoke your own tools to merge data into checked-out version>

Z:\avob> 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've made this annotation, your merge is indistinguishable from one performed with ClearCase tools.

12 Using Element Types to Customize Processing of File Elements

Most projects involve many different file types. For example, in a typical software release, developers may work on C-language source files, C-language header files, document files in binary format, and library files.

Every file that is stored in a ClearCase VOB is associated with an element type. ClearCase provides predefined element types for various kinds of file types, 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.

This chapter describes how ClearCase uses element types and type managers to classify and manage files. It also describes how you can customize file classification and management.

12.1 File Types in a Typical Project

Table 4 lists the files used in a typical development project.

Table 4 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`
Derived Files	
library, shared library	**.lib**, **.dll** file-name extension
compiled executable	**.exe** file-name extension

12.2 How ClearCase Assigns Element Types

In various contexts, ClearCase determines one or more *file types* 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, ClearCase determines the file type for the element.

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 and **stat** data.

For example, the following magic file specifies several file types for each kind of file listed in Table 4.

Sample Magic File

```
c_src src_file text_file file:       -name "*.c";
hdr_file text_file file:             -name "*.h" ;
frm_doc binary_delta_file doc file:  -magic 0, "<MakerFile" ;
library derived_file file:           -name "*.lib";
program compressed_file:             -name "*.exe" ;
```

12.3 Element Types and Type Managers

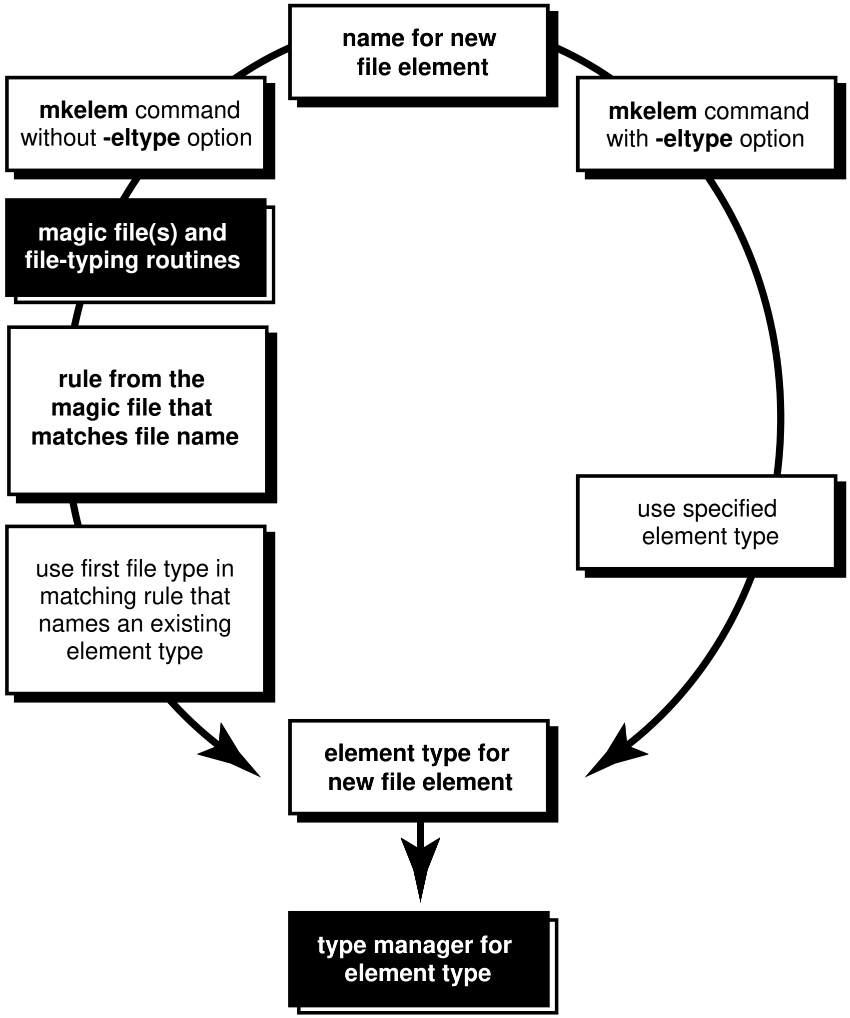
ClearCase can handle different classes of files differently because it uses *element types* to categorize elements. Each file element in a VOB must have an element type. An element gets its type when it is created; you can change an element's type subsequently, with the **chtype** command. (An element 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 a file element's data is handled depends on its element type.

NOTE: 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.

Figure 44 shows how an element type is assigned to a newly created element.

Figure 44 Data Handling: File Type, Element Type, Type Manager



For example, a new element named **monet.h** is assigned an element type as follows:

1. A developer creates an element:

   `Z:\myvob>` **cleartool mkelem monet.h**

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.

   NOTE: ClearCase supports a search path facility, using the environment variable **MAGIC_PATH**. See the **cc.magic** reference page for details.

   Suppose that the magic file shown in *Sample Magic File* on page 156 is the first (or only) one to be used. In this case, rule (2) is the first to match the name **monet.h**, yielding this list of file types:

   ```
   hdr_file text_file file
   ```

3. This list is compared with the set of element types defined for the new element's VOB. Suppose that **text_file** is the first file type that names an existing element type; in this case, **monet.h** is created as an element of type **text_file**.

4. Data storage and retrieval for versions of element **monet.h** are handled by the type manager associated with the **text_file** element type; its name is **text_file_delta**:

   ```
   Z:\myvob> 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 ClearCase administrator can use global types.) In this case, a new element, **monet.h**, 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 **hdr_file** element.

Other Applications of Element Types

Element types allow differential and customized handling of files beyond the selection of type managers. Following are some examples.

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 has reorganized 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 reconfigured 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** is to be executed on each C-language source file. If all such files have element type **c_src**, a single **cleartool** command runs the program:

```
cleartool> find –avobs –visible –element 'eltype(c_src)' ^
     –exec 'check_var_names %CLEARCASE_PN%'
```

12.4 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; 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 is associated with one of the type managers provided with 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 159. For differential data handling, use the **–manager** option to create an element type that uses a different type manager from its supertype.

13 Using ClearCase Throughout the Development Cycle

The previous chapters describe various aspects of managing a project with ClearCase. This chapter presents one way in which you can use ClearCase to organize the work throughout a development project. During this cycle, developers create a new release and maintain the previous release.

This chapter describes concepts and methods to address typical organizational needs. Instead of using command-line tools that are described here, consider using GUI tools such as the Merge Manager to accomplish similar goals.

13.1 Project Overview

Release 2.0 development of the **monet** project includes the following kinds of work:

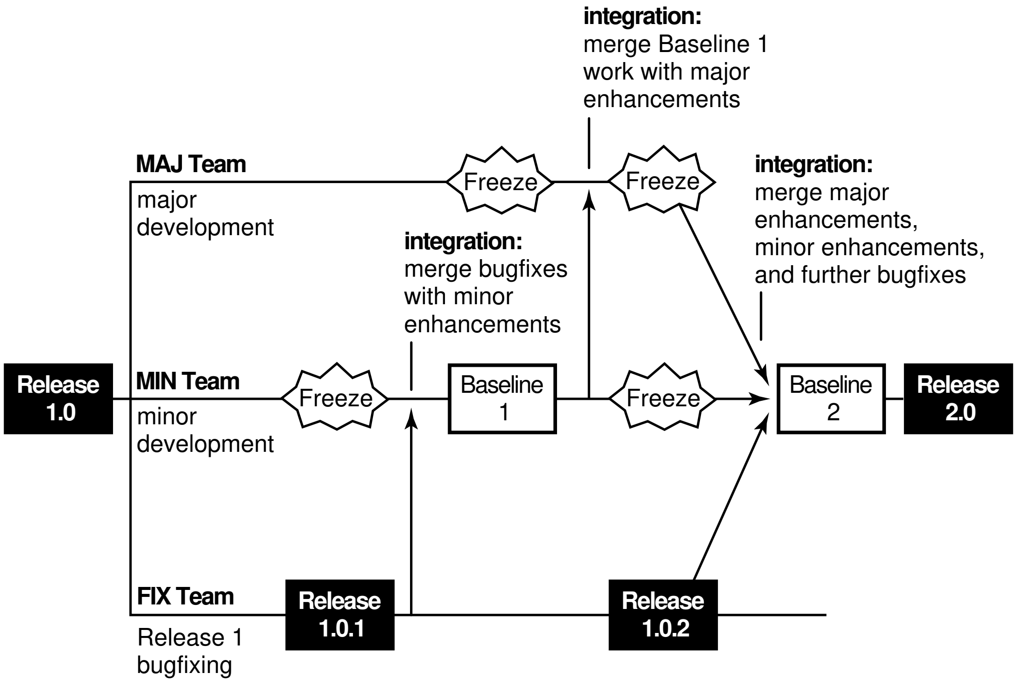
- Patches. Several high-priority bug fixes to Release 1.0 are needed.
- Minor enhancements. Some commands need new options; some option names need to be shortened (**–recursive** becomes **–r**); some algorithms need performance work.
- Major new features. A graphical user interface is required, as are many new commands and internationalization support.

These three development efforts can proceed largely in parallel (Figure 45), 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 45 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. ClearCase makes it easy to 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).

NOTE: Some developers may belong to multiple teams. These developers work in multiple views, each configured for the respective team's tasks.

The development area for the **monet** project is shown here. At the beginning of Release 2.0 development, the most recent versions on the **main** branch are labeled **R1.0**.

`\monet`	*(project top-level directory)*
`    src\`	*(sources)*
`    include\`	*(include files)*
`    lib\`	*(shared libraries)*

13.2 Development Strategy

This section describes the ClearCase issues to be resolved before development begins.

Project Manager and 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 is 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.

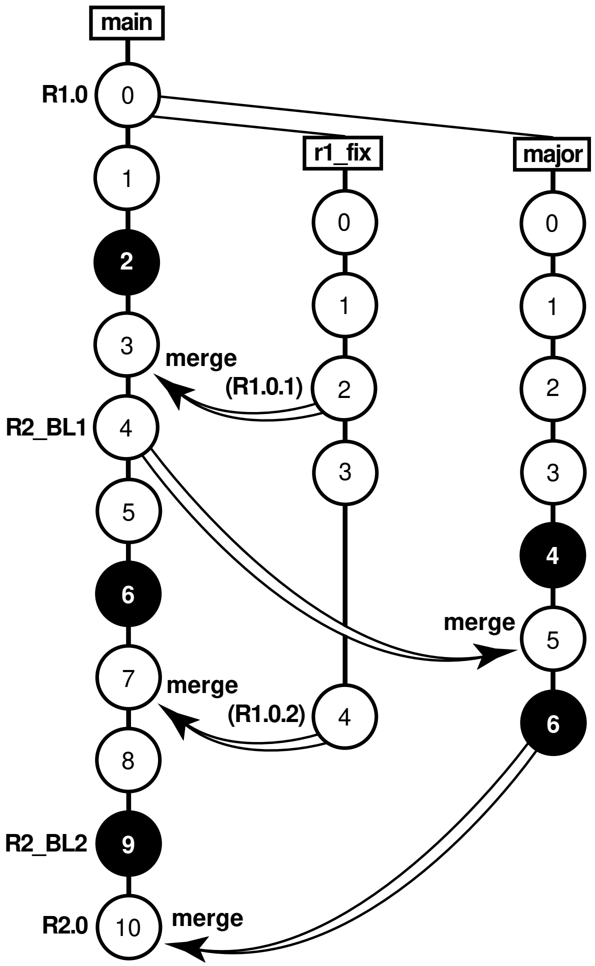
Because the MIN team will produce the first baseline release on its own, the project manager gives the **main** branch to this team. The MAJ team will develop new features on a subbranch, and will not be ready to integrate for a while; the FIX team will fix Release 1.0 bugs on another subbranch and can 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.

Figure 46 shows the evolution of a typical element during Release 2.0 development, and indicates correspondences to the overall project plan (Figure 45).

Figure 46 Development Milestones: Evolution of a Typical Element



1. (All branches) Start minor and major enhancements, along with **R1.0** bug fixing
2. (**main**) Freeze minor enhancements work
3. (**main**) Merge bug fixes from Release 1.0.1 into minor enhancements
4. (**main**) Baseline 1 release
5. (**major**) Freeze major enhancements work
6. (**major**) Merge Baseline 1 changes into major enhancements
7. (**main**) Freeze minor enhancements work
8. (**main**) Merge additional bugfixes into minor enhancements
9. (**major**) Freeze major enhancements work
10. (**main**) Merge major enhancements work with minor enhancements work
11. (**main**) Baseline 2 release
12. (**main**) Final testing period
13. (**main**) Release 2.0

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) and Rule (4) 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**.

13.3 Creating Branch Types

The project manager creates the **major** and **r1_fix** branch types required for the config specs in *Creating Project Views* on page 166:

cleartool mkbrtype –c "monet R2 major enhancements" major@\libpub major@\monet

```
Created branch type "major".
Created branch type "major".
```

cleartool mkbrtype –c "monet R1 bugfixes" r1_fix@\libpub r1_fix@\monet

```
Created branch type "r1_fix".
Created branch type "r1_fix".
```

NOTE: 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.

13.4 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:

- **\\vulcan\c_specs\major** contains the MAJ team's config spec.
- **\\vulcan\c_specs\fix** contains the FIX team's config spec.

These config spec files are stored in a standard directory outside a VOB, to ensure that all developers get the same version.

13.5 Creating, Configuring, and Registering Views

Each developer creates a view on his or her own machine. For example, developer **arb** enters these commands on her local host **quark**:

```
C:\> mkdir c:\users\arb\vw_store
c:\> net share users c:\users
C:\> cleartool mkview –tag arb_major \\quark\users\arb\vw_store\arb_major.vws
Created view.
Host-local path: quark:C:\users\arb\vw_store\arb_major.vws
Global path:     \\quark\users\arb\vw_store\arb_major.vws
```

NOTE: The **net share** command makes arb's view publicly accessible.

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. **arb** edits her config spec with the **cleartool edcs** command, deletes the existing lines, and adds the following line:

```
\\vulcan\c_specs\major
```

If the project manager changes the standard file, **arb** must enter the command **cleartool setcs –current** to pick up the changes.

13.6 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:

```
C:\> mn                                    (mn is a common convention; it stands for "main")
c:\> z:
Z:\> cd \monet\src
Z:\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) in the MAJ team's config spec in *Creating Project Views* on page 166). If a developer on the MIN team enters this command, the elements are checked out on the **main** branch, with no conflict.

ClearCase is fully compatible with standard development tools and practices. Thus, developers use the editing, compilation, and debugging tools they prefer 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 intrateam 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 reconfigure the view to select the versions at a point before those changes were made.

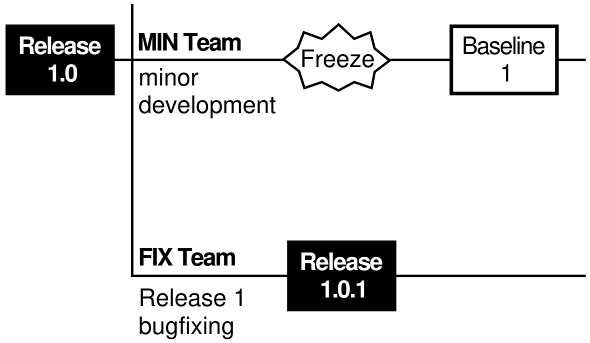
➤ Private subbranches. A developer can create a private subbranch in one or more elements (for example, **\main\major\anne_wk**). The config spec must be changed to select versions on the **\main\major\anne_wk** branch instead of versions on the **\main\major** branch.

➤ Viewing only their own revisions. Developers can use a ClearCase query to configure a view that sees only their own revisions to the source tree.

13.7 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 47).

Figure 47 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 \libpub \monet\src ^**
–fversion ...\r1_fix\LATEST –merge –graphical

.
. <*lots of output*>
.

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 shared directory that is accessible from all developer hosts:

```
C:\> mkdir c:\vw_store
C:\> net share c c:\
C:\> cleartool mkview –tag base1_vu \\nt_svr\c\vw_store\base1_vu.vws
Created view.
Host-local path: nt_svr:C:\vw_store\base1_vu.vws
Global path:     \\nt_svr\c\vw_store\base1_vu.vws
```

To make the view accessible only to the MIN team, the project manager must use File Manager to set permissions on the **vw_store** directory so that only the members of the MIN team can access it.

Any member of the **mon** group can use the following command to begin working in this view:

```
C:\> net use z: \\view\base1_vu
C:\> z:
Z:\>
```

Because all integration work takes place on the **main** branch, there is no need to change the configuration of the new view from the ClearCase default. MIN developers use this view (**net use** *DRIVE*: **\\view\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 created to fix inconsistencies (and other bugs) go onto the **main** branch.

Labeling Sources

The **monet** application's 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:

```
Z:\> cleartool mklbtype –c "Release 2, Baseline 1" R2_BL1@\monet R2_BL1@\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:

```
Z:\> cleartool lock –nusers meister lbtype:R2_BL1@vob:\monet lbtype:R2_BL1@\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 permitted):

```
Z:\> cleartool lscheckout –all \monet
```

```
Z:\> cleartool lscheckout –all \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:

```
Z:\> cleartool mklabel –recurse R2_BL1 \monet \libpub
Created label "R2_BL1" on "\monet" version "\main\1".
Created label "R2_BL1" on "\monet\src" version "\main\3".
 <many more label messages>
```

Removing the Integration View

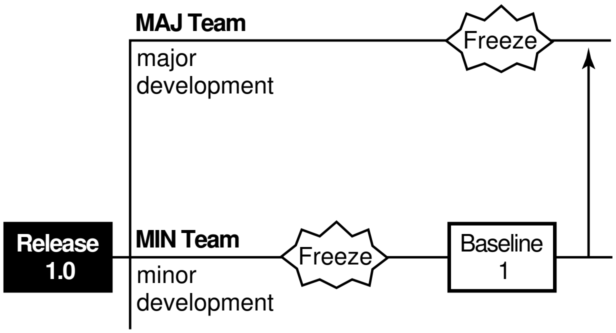
The view registered as **base1_vu** is no longer needed, so the project manager removes it:

```
C:\> cleartool rmview –force –tag base1_vu
```

13.8 Merging Ongoing Development Work

After Baseline 1 is created, the MAJ team merges the Baseline 1 changes into its work (Figure 48). 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 48 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.

NOTE: 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:

   `Z:\>` **cleartool lscheckout –all \monet \libpub**

   NOTE: 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:

```
Z:\> net use y: \\view\major_vu                    (use any MAJ team view)
Z:\> y:
Y:\> cleartool findmerge \monet  \libpub –type d –fversion \main\LATEST –merge
Needs merge \monet\src [automatic to \main\major\3 from \main\LATEST]
   .
   . <lots of output>
   .
Log has been written to "findmerge.log.04-Feb-99.09:58:25".
```

3. After checking in the files, the project manager determines which elements need to be merged:

```
Y:\> cleartool findmerge \monet  \libpub –fversion \main\LATEST –print
   .
   . <lots of output>
   .
A 'findmerge' log has been written to
"findmerge.log.04-Feb-99.10.01.23"
```

This last **findmerge** log file is in the form of a batch file: it contains a series of **cleartool findmerge** commands, each of which performs the required merge for one element:

```
Y:\> type findmerge.log.04-Feb-99.10.01.23
cleartool findmerge \monet\src\opt.c@@\main\major\1 -fver \main\LATEST -merge
cleartool findmerge \monet\src\prs.c@@\main\major\3 -fver \main\LATEST -merge
   .
   .
cleartool findmerge \libpub\src\dcanon.c@@\main\major\3 -fver \main\LATEST -merge
cleartool findmerge \libpub\src\getcwd.c@@\main\major\2 -fver \main\LATEST -merge
cleartool findmerge \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@\monet brtype:major@\libpub
Locked branch type "major".
Locked branch type "major".
```

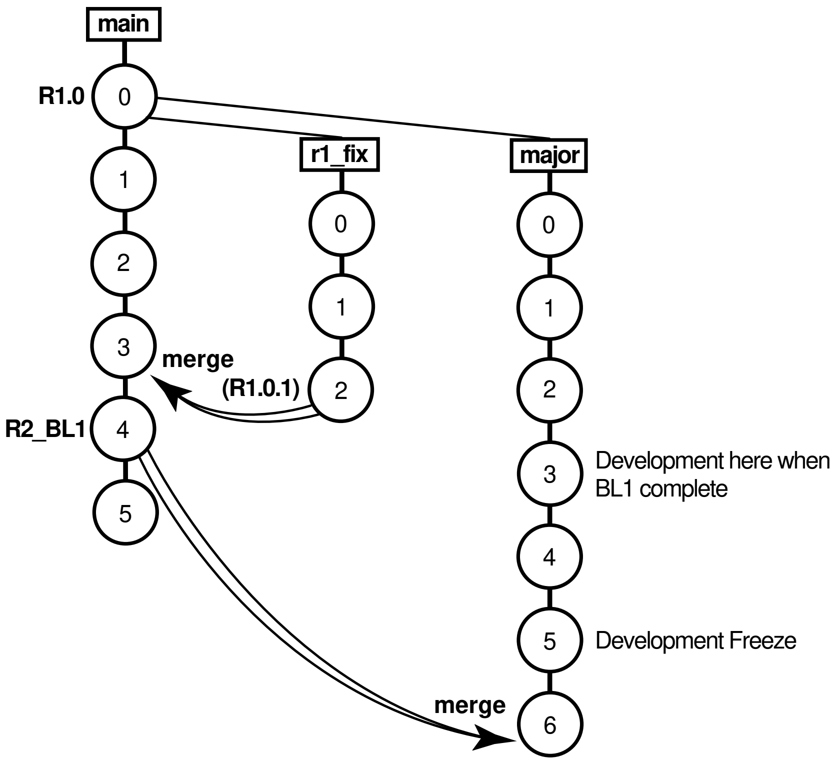
Merging Work

Because the MAJ team is not contributing to a baseline soon, it is not essential 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 (Figure 49).

Figure 49 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:

```
Y:\> cleartool findmerge \monet \libpub –fversion \main\LATEST –whynot –print
   .
   .
No merge "\monet\src" [\main\major\4 already merged from \main\3]
No merge "\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:

```
Y:\> cleartool unlock brtype:major@\monet brtype:major@\libpub
Unlocked branch type "major".
Unlocked branch type "major".
```

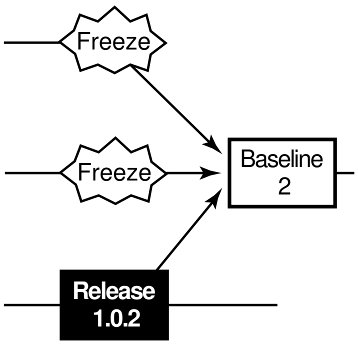
13.9 Creating Baseline 2

The MIN team is ready to freeze for Baseline 2, and the MAJ team will be soon (Figure 50). Baseline 2 will integrate all three development efforts, thus requiring 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 171.)

Figure 50 Baseline 2



ClearCase supports merges 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 169.

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.

NOTE: 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:

```
Y:\> cleartool lscheckout –brtype main –recurse \monet \libpub
Y:\> cleartool lscheckout –brtype major –recurse \monet \libpub
Y:\>
```

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:

```
Y:\> net use x: \\view\minor_vu                    (use any MIN team view)
Y:\> x:
X:\> cleartool findmerge \monet \libpub –fversion ...\major\LATEST –print
   .
   . <lots of output>
   .
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 ClearCase to record the merges with a hyperlink of type **Merge**:

```
X:\> cleartool lock –nusers arb,david brtype:major@\monet brtype:major@\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:

```
X:\> cleartool lock –nusers meister,arb,david,sakai ^
brtype:main@\monet  brtype:main@\libpub
Locked branch type "main".
Locked branch type "main".
```

(To lock the branch, you must be the branch creator, element owner, VOB owner, or a member of the *ClearCase group*. See the **lock** reference page.)

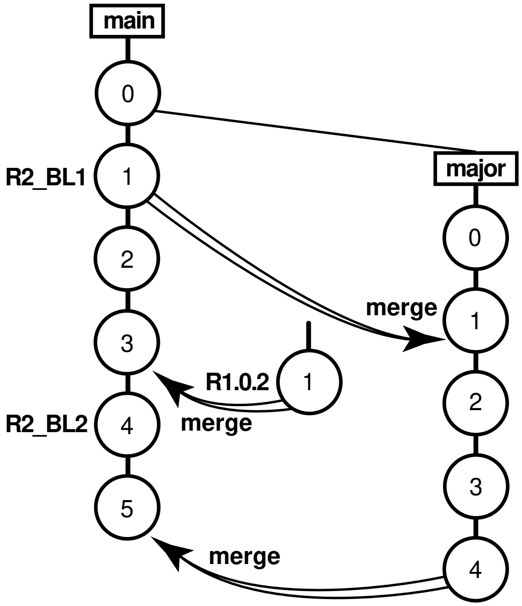
Merging from the major Branch

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 172. 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:

```
X:\> cleartool findmerge \monet \libpub –fversion \main\major\LATEST ^
–merge –graphical
   .
   . <lots of output>
   .
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 51.

Figure 51 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:

`X:\>` **cleartool lock –replace –obsolete brtype:major@\monet brtype:major@\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 170). 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**.)

13.10 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 (Figure 52).

Figure 52 Final Test and Release

Before it is shipped to customers, the **monet** application 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:

  a. The project manager authorizes a particular developer to fix the bug, by granting her permission to create new versions (on the **main** branch).

  b. The developer's checkin activity is tracked by a ClearCase trigger.

  c. 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@\monet R2.0@\libpub
```
Created label type "R2.0".
Created label type "R2.0".
```

cleartool lock –nusers meister lbtype:R2.0@\monet lbtype:R2.0@\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 \monet \libpub
<*many label messages*>

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 177). Now, the project manager asks **vobadm** to close down the **main** branch to everyone except himself, **meister**:

`Z:\>` **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 181.)

Setting Up the Test View

The project manager creates a new shared view, **r2_vu**, that is configured with a one-rule config spec:

`Z:\>` **cleartool mkview –tag r2_vu \\nt_svr\public\integ_r2.vws**
```
Created view.
Host-local path:   nt_srv:c:\public\integ_r2.vws
Global path:           \\nt_srv\public\integ_r2.vws
```
`Z:\>` **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 Bugfixing

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 *Policy: Notify Team Members of Relevant Changes* on page 132).

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 "ccperl \\neon\scripts\notify_manager.pl" r2_checkin@\monet r2_checkin@\libpub
Created trigger type "r2_checkin".
Created trigger type "r2_checkin".
```

Only the VOB owner or a member of the ClearCase group 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.

```
Z:\> cleartool lock –replace –nusers arb,meister brtype:main
Locked branch type "main".
```

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 \monet\src\opt.c by arb
\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**:

`Z:\>` **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:

`Z:\>` **cleartool mklabel –replace R2.0 z:\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** 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**, the project manager asks the ClearCase administrator to clean up and prepare for the next release:

- The ClearCase administrator removes the checkin triggers from all elements by deleting the all-element trigger type:

  cleartool rmtype trtype:r2_checkin@\monet trtype:r2_checkin@\libpub
  ```
  Removed trigger type "r2_checkin".
  Removed trigger type "r2_checkin".
  ```

- The ClearCase administrator reopens the **main** branch:

  cleartool unlock brtype:main
  ```
  Unlocked branch type "main".
  ```

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.

A.1 View Profiles and UCM

Base ClearCase includes a set of features called *view profiles*, which you can use to automate much of the work required to set up and maintain your team's shared 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

This section compares the features of view profiles and UCM.

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's shared integration branch, just as a view profile's 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. 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 5 summarizes the key differences between view profiles and UCM features.

Table 5 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.

Table 5 View Profile Features and Their UCM Counterparts

View profile construct	UCM counterpart
No automated support for updating private branch with work from other developers.	Rebase operaton adds changes from the integration stream to private work area.
Views are configured with information from profiles.	Views are configured with information from streams.

A.2 How to Move View Profile Information to UCM

This section presents some general guidelines on 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. 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.

Members of the project team can now join the project, creating their own development streams and views.

For further information on creating a UCM project, see Chapter 5, *Setting Up the Project*.

B ClearCase-ClearQuest Integrations

B.1 Understanding the Two ClearCase-ClearQuest Integrations

The integration of ClearQuest and ClearCase associates one or more ClearQuest records with one or more ClearCase versions allowing you to use features of each product. ClearCase supports two separate integrations with ClearQuest:

- The base ClearCase-ClearQuest integration
- The UCM-ClearQuest integration

Online help that describes the base ClearCase-ClearQuest integration is available from within the ClearQuest Integration Configuration GUI on Windows. To open the GUI, click **Start>Programs>ClearCase Administration>Integrations>ClearQuest Integration Configuration**. Note that this integration cannot be used with UCM projects.

For further information on the UCM-ClearCase integration, see Part 1 of this book.

In general, we recommend that you use the base ClearCase and UCM integrations separately, and avoid using a common ClearQuest user database. However, it is possible for both integrations to use the same ClearQuest user database. This can be useful if you are moving a project to UCM and have a substantial amount of information in a ClearQuest user database that was created with the base ClearCase-ClearQuest integration. You may want the new work in UCM to be reflected in new ClearQuest records in the same ClearQuest user database.

The remainder of this appendix discusses considerations in managing the coexistence of the base ClearCase-ClearQuest integration and the UCM-ClearQuest integration.

Managing Coexisting Integrations

When a ClearQuest user database that had been integrated with ClearCase previously is configured for integration with UCM, the existing change sets are preserved intact in the ClearQuest user database, but cannot be migrated to the UCM integration.

Change sets of existing records in the ClearQuest user database are preserved, and you can access them from ClearQuest. 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-ClearQuest Integration* on page 36 for related information.

Schema

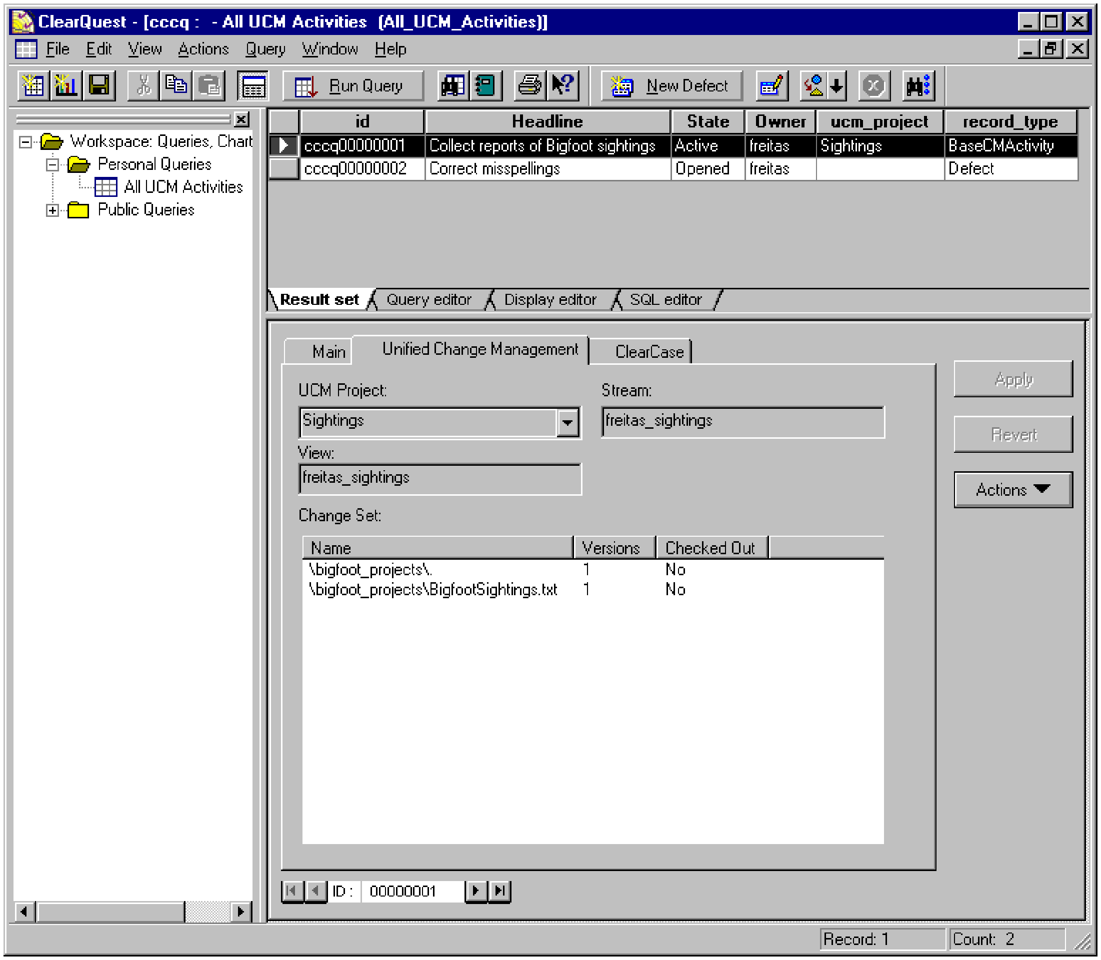
A ClearQuest schema can contain modifications from both the base ClearCase-ClearQuest integration and the UCM-ClearQuest integration. A record type in such a schema would include both the ClearCase package and the Unified Change Management package.

An individual record of that record type can store either 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, as shown in Figure 53. The **Unified Change Management** tab lists the change set for a UCM activity. The **ClearCase** tab shows the change set associated with a ClearQuest record.

Figure 53 Change Sets in ClearQuest GUI



Customizing ClearCase Reports

C

This appendix explains how to customize ClearCase Reports. Specifically, it introduces the ClearCase Reports Programming Interface and gives examples of how you can customize the the report procedures and the user interface.

C.1 How ClearCase Reports Works

ClearCase Reports consists of two parts:

- The report procedures, which you can modify.
- The 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, ClearCase Reports executes as two applications: ClearCase Report Builder and ClearCase Report Viewer. The Report Builder is used to select and define a report's 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 using parameter choosers, 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.

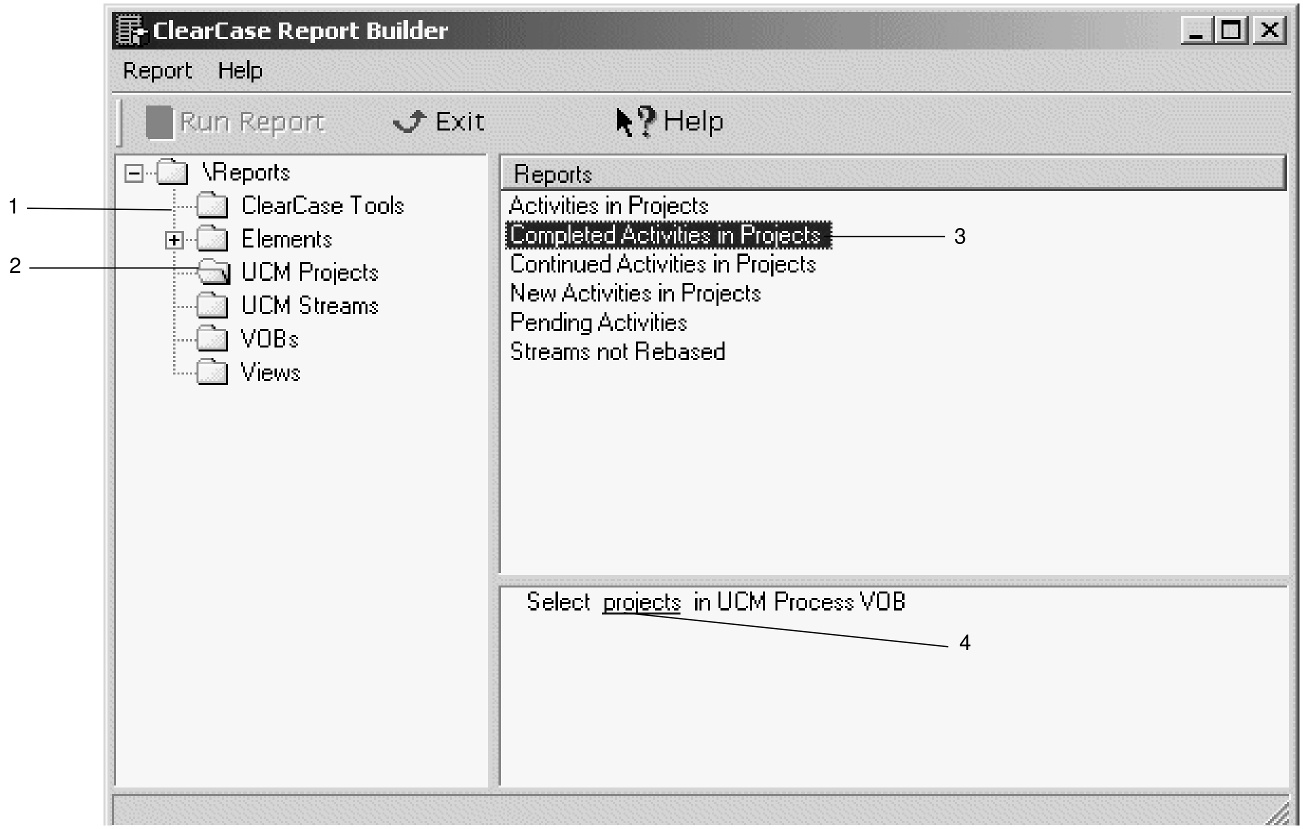
For more information on the processing sequence between the ClearCase Reports applications and the report procedures, see *Run-Time Processing Sequence for Reports Programming Interface* on page 194

C.2 What You Can Customize in ClearCase Reports

The 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 annotated areas (see Figure 54) of the Report Builder:

- Area 1. The name of the folders in the tree pane.
- Area 2. The directory organization displayed in the tree pane.
- Area 3. The report description.
- Area 4. The report parameters

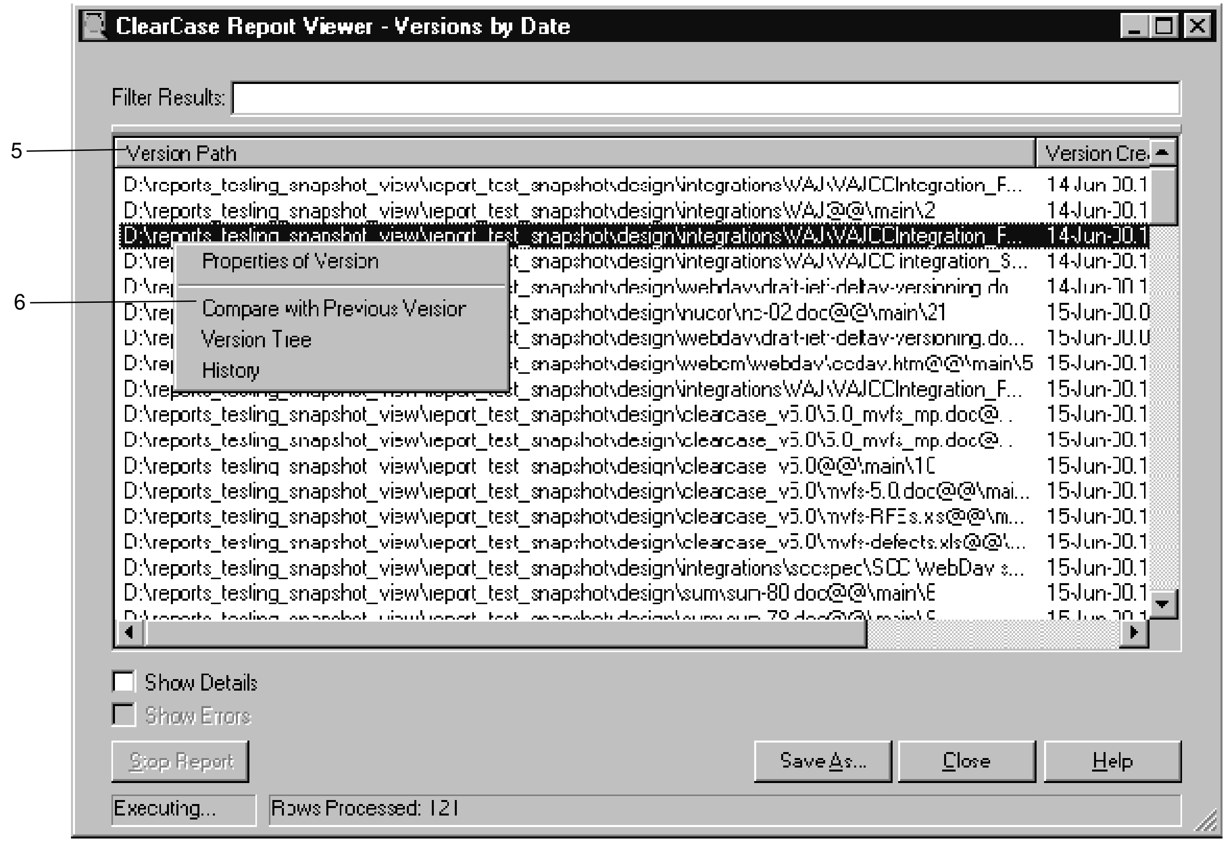
Figure 54 Customizable Areas of Report Builder Interface



You can customize by adding, changing, or removing information for the annotated areas (see Figure 55) of the Report Viewer.

- Area 5. 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.
- Area 6. The commands on the shortcut menu.

Figure 55 Customizable Interface for Report Viewer Window



Refer to *Report Programming Examples* on page 214 for programming examples to demonstrate these areas of report customizations.

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. Processing occurs in three phases.

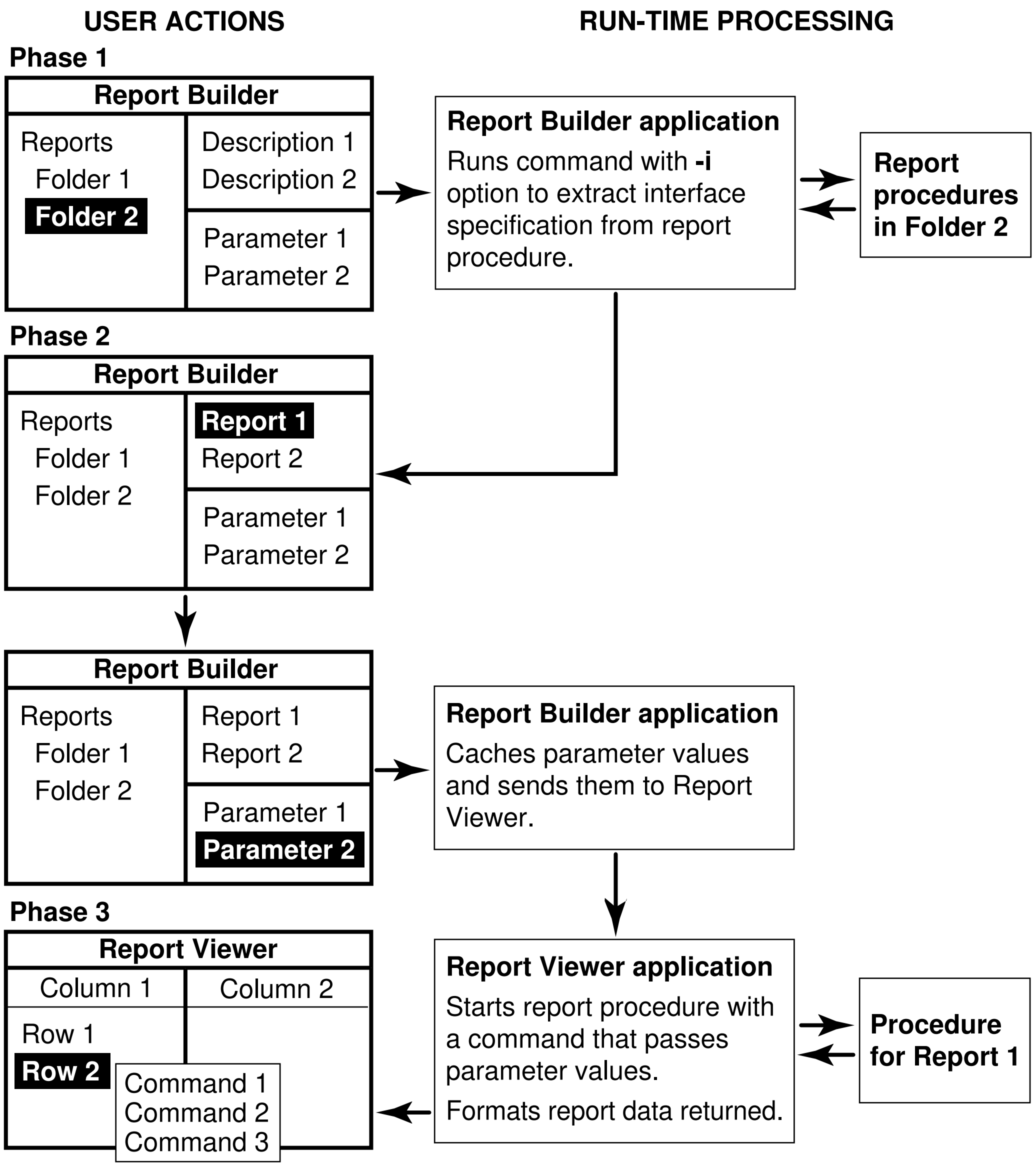
In phase 1, the user opens one of the subfolders in the Reports folder. The Report Builder processes the interface specification of all report procedures associated with the reports in that subfolder and presents the description of each report in the reports pane of the Report Builder. The parameters 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 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 56 illustrates this processing sequence.

Figure 56 Run-Time Processing Sequence



To execute these processing steps 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, which it calls when a user clicks the associated directory folder.
- The report procedure must have a valid interface specification. If the expected format is not present, the report will not run.
- The interface specification in the report procedure must use parameters and choosers supplied by ClearCase Reports. See *Parameters Supplied with ClearCase Reports* on page 205.
- 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 ClearCase is installed on the client, the files for ClearCase Reports are installed 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 ClearCase view to serve as the shared location.

We recommend that you place the copy of the report procedures under source control.

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. We recommend that you do not delete any files that are in **\script_tools** and **\scripts_rightclick;** you may add your own, of course.

The help file used by the reports cannot be modified and is not included in the **\reports** directory. The help file for 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:

   clearexport_ffile -o *name-of-data-file*

3. Copy the data file to an existing VOB.

4. In the **\reports** directory in an existing VOB, enter a command of this form:

   clearimport -verbose -directory \reports*name-of-data-file*

5. Create a dynamic or snapshot view for the ClearCase reports data that is now under source control .

Pointing the Report Builder to the Customized Directory

After you have copied the installed files for 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 dialog box.**

2. In the dialog box, do one of the following:

   ➢ Type the directory path for the customized directory in the text box.

   ➢ Click **...** to open the **Browse for scripts location** dialog box to select a directory location.

   Note: After changing the ClearCase Reports user interface, you must restart Report Builder to activate the changes.

Default Directory Structure for ClearCase Reports

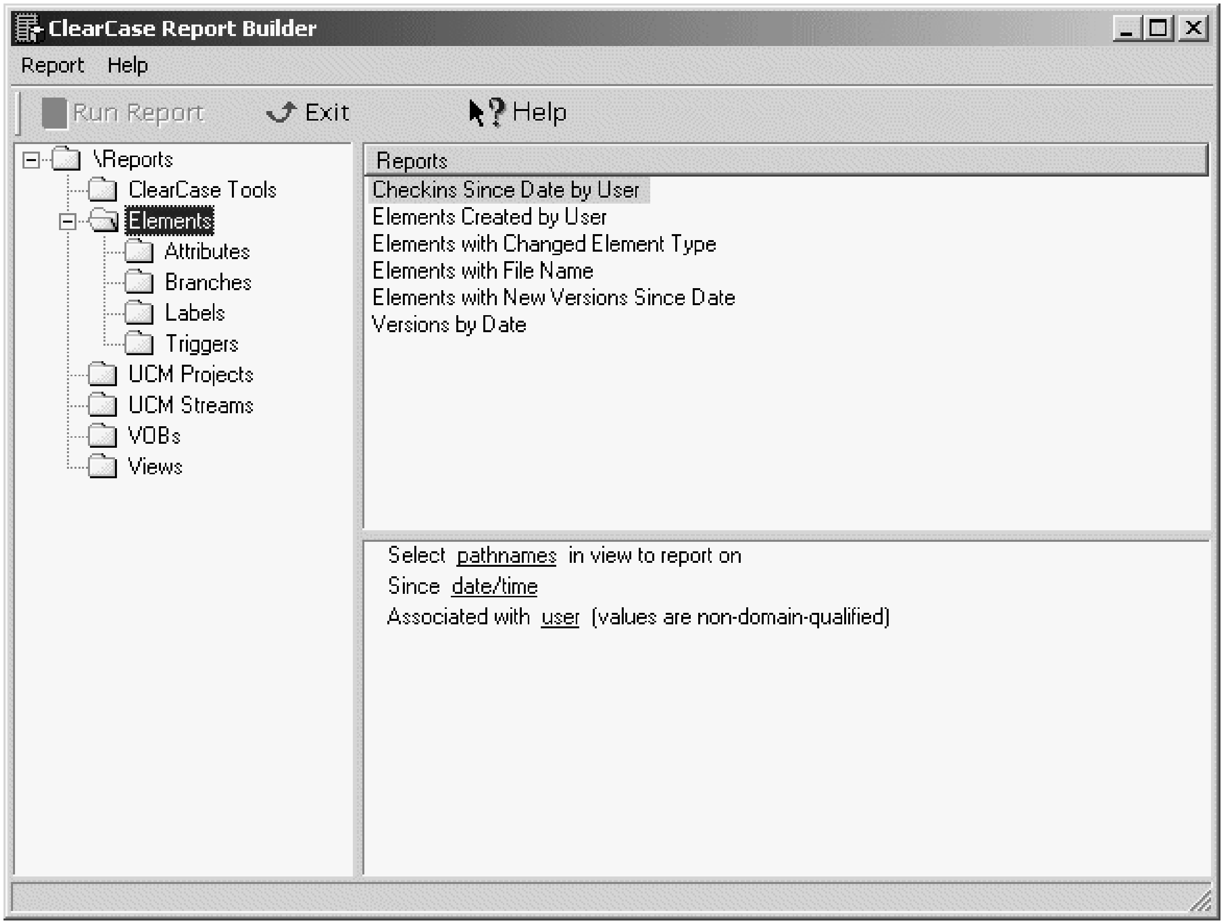
All files for ClearCase Reports are rooted in *ccase-home-dir***reports**. This is the directory structure:

```
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

As seen Figure 57, the Report Builder window contains three panes: the left pane is the tree pane, the top-right pane is the reports pane, and the bottom-right pane is the parameter pane. 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.

Figure 57 Report Builder User Interface



The Report Builder "walks" through 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 ClearCase Automation Library (CAL) to extract data
.pl	Perl, executed under **perl.exe** from user's **PATH** environment variable, for example, ActiveState Perl
.prl	ccperl
.js	JavaScript, run under Windows Scripting Host (cscript.exe)
.vbs	VBScript, run under Windows Scripting Host (cscript.exe)

All other files are ignored. The report procedures supplied with ClearCase Reports end in **.prl**, which is associated with **ccperl.exe** by the Report Builder.

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, for the on-disk directory tree shown here

```
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

```
\Reports
     Admin Reports\
     UCM Reports\
```

C.3 Report Procedure Interface Specifications

As the Report Builder finds report procedures in the customized directory, it queries each report procedure for its interface specification. It does this by starting 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* on page 238.

The examples in the following sections 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 rightclick event in the Report Viewer window calls **Properties _of_View.prl,** which is based on a data stream from the **View Tag** field.

Interface Specification for test_null.prl

The Report Builder uses this command to run **test_null.prl**:

ccperl "D:\Program Files\Rational\Clearcase\Reports\scripts\samples\test_null.prl" -i

This is the interface specification:

```
description : "null interface"
fields : "field1"(string)
```

This report interface attaches the Report Viewer to the field named **field1**.

Interface Specification for test2_null.prl

The Report Builder uses this command to run **test2_null.prl**:

ccperl "D:\Program Files\Rational\Clearcase\Reports\scripts\samples\test2_null.prl" -i

This is the interface specification:

```
description : "null interface"
```

When only **description** is defined, a report procedure can run other graphical user interfaces (for example, **clearprompt)** or otherwise interact with the user. Note, the only required line in the interface specification is **description**; all of the other interface definitions are optional. The reports in the**\ClearCase_Tools** folder define **description only**.

Description Specification

The **description** is the only required part of an interface specification. 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 ID Specification

Help IDs for the **description** and **parameter** fields are supplied for the Report Builder user interface. The **help id** specification is the ID that supports context-sensitive help for the

description or **parameter** fields in the Report Builder user interface. The IDs are integers in the following range:

0-999	Context help for parameters
2000–2999	Context help for report descriptions

The help file for ClearCase Reports (**cc_reports.hlp)** is located in *ccase-home-dir***bin**. The help IDs for the **parameter** and **description** fields **are included in this file**. In this version of ClearCase Reports, you cannot add an ID for your own report.

Parameters Specification

When specifying **parameters**, you can use only those supplied with ClearCase Reports. Each parameter has an associated chooser control, parameter text, and a help ID. (See Table 1.) When you use one of these parameters, naming it is all that is required. For example, this is the **parameter** specification for the **Find elements between two labels** report:

```
Parameters : LOOKIN LABEL LABEL
```

The order of parameters is important. The parameters are displayed in the parameter pane in the order of the specification. (A hotspot for each parameter prompts the user to click to enter a value for the parameter.) At run time, after the parameters are defined with values by the end-user, when the Report Viewer calls the report procedure, the report procedure must handle the parameter values in the same order as they are defined in the specification.

The parameters in Table 6 that are associated with the Type Chooser must also specify the **LOOKIN** parameter in the **interface** specification. The value for the **LOOKIN** parameter must be defined by the user before values for parameters using 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.
```

Table 6 Parameters Supplied with ClearCase Reports (Part 1 of 3)

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

Table 6 Parameters Supplied with ClearCase Reports (Part 2 of 3)

Parameter	Default text displayed in the Parameter pane	Help ID	Chooser	Selection
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 pathnames 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
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 pathname	29	Text	Single

Table 6 Parameters Supplied with ClearCase Reports (Part 3 of 3)

Parameter	Default text displayed in the Parameter pane	Help ID	Chooser	Selection
STRING	With string	30	Text	Single
INTEGER	Enter integer	31	Text	Single
REGULAR_ EXPRESSION	Enter regular expression	32	Text	Single

Rightclick Specification

The **rightclick** specification is a list of commands available on the shortcut menu in the Report Viewer. All rightclick events are supported by a list of scripts in the **\scripts_rightclick** directory. The **rightclick** specification allows you to control the text on the shortcut 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 shortcut 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 7 describes the supported modifiers.

Table 7 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 rightclick 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 not optional):

```
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 8 lists the names for **field_types** and the kind of data represented. We encourage you to use these definitions in your own report procedures wherever possible; however, a report procedure can specify that it is sending back a field with a **field_type** that is not in Table 8. Depending on the column width 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.

Table 8 Field Type Supplied with ClearCase Reports (Part 1 of 2)

Field Name	**Data desciption**	**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
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	atria\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

Table 8 Field Type Supplied with ClearCase Reports (Part 2 of 2)

Field Name	Data desciption	Example
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

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 (see Table 6). . These choosers are supplied with ClearCase Reports:

- Path Chooser
- UCM Targets Chooser
- Types Chooser
- Date/Time Chooser
- Text Chooser

For user information, click **Help** in any chooser dialog box.

Path Chooser

The Path Chooser is associated with the **LOOKIN** parameter. It presents a list of view pathnames for users to select, and then sends the selected pathnames to the report procedure. It is also used for the **PVOB** and **PVOBS** parameters to choose the VOB-tag of a UCM Process 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 the **COMPONENT**, **COMPONENTS**, **BASELINE**, **BASELINES**, **USER**, **GROUP**, **ATTRIBUTE_VALUE**, **ELTYPE**, **BRANCHLEVELS**, **FILE_NAME**, **PATH**, **STRING**, **INTEGER**, and **REGULAR_EXPRESSION** parameters.

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 text box reads “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:”.

Viewing the Report

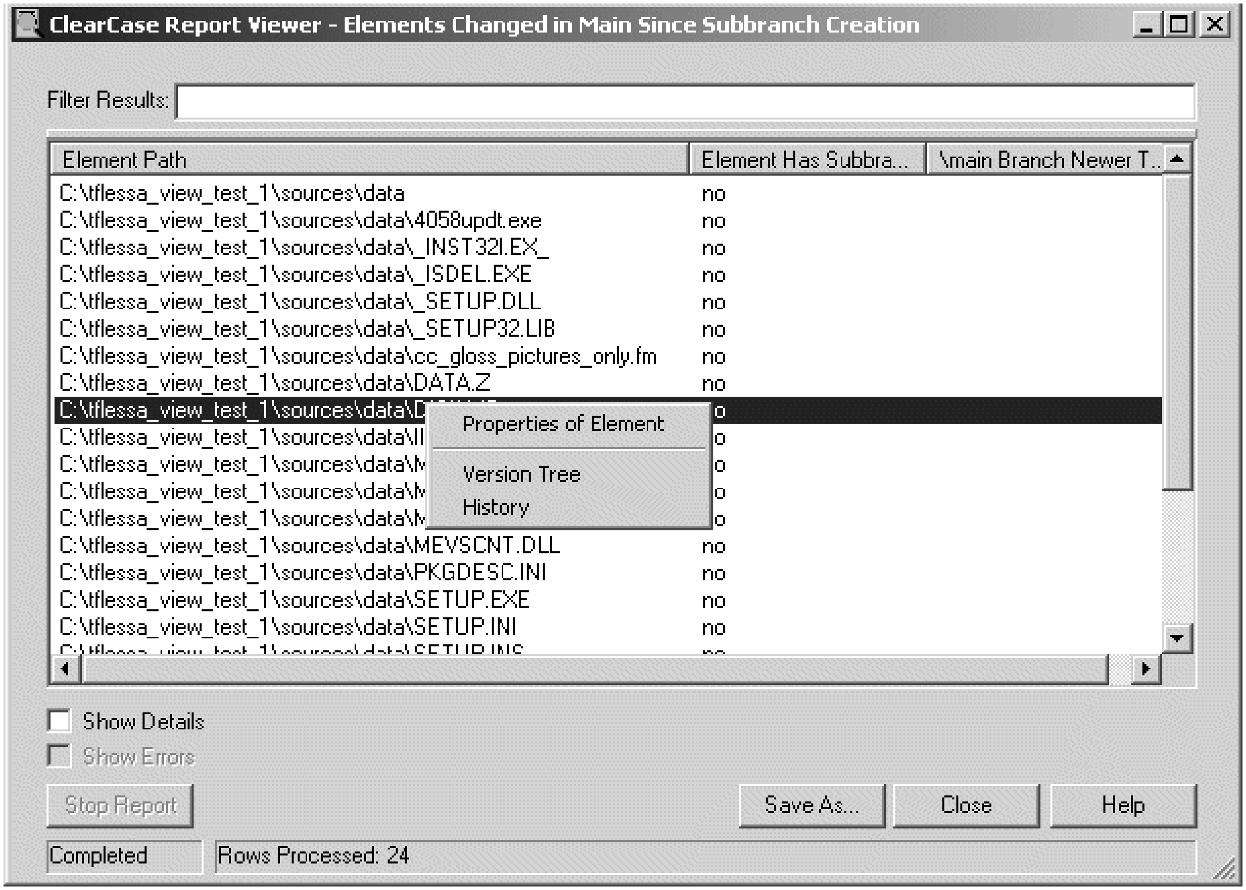
When all required parameters have values, clicking **Run Report** opens the Report Viewer window (see Figure 58). The Report Builder creates a command line to pass the user-defined parameters, in the order defined by interface specification . For example, if a report procedure asks for parameters **LOOKIN 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 for those with a **field_type** of **time_t**, **integer**, and **cctime**. For these three fields only, Report Viewer uses numeric sorting.

Figure 58 Report Viewer Window



Saving Report Data

Clicking **Save As** in the Report Viewer window opens a standard file selection dialog box 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 *pathname* value for the **-out** option**,** where *pathname* is the value that the Report Viewer passes from the Path Chooser.

XML output is supported directly by the Report Viewer. You can reimplement 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. We recommend that you start with the style sheet supplied with ClearCase Reports (**\script_tools\table.xsl)** to create customized XSL files.

C.4 Report Programming Examples

All report procedures supplied with 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 from the ClearCase Customer Web site at **http://clearcase.rational.com/contrib/T0046/T0046.zip**. 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 shortcut menu for **Element_with_Labels.prl.**
- Example 5: Adding a user-defined command to the shortcut 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 user-defined pathname parameter. 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 the following form:

   ```
   LOOKIN = "<path1> [<path2> ...]"
   ```

   This parameter specifies the list of paths with which the **cleartool find** command is to be invoked.

2. The routine, when invoked, extracts the paths from the **LOOKIN** string and passes them to the **check_lookin()** routine (located in **common_script.prl**).

3. **check_lookin()** then puts the paths into the global variable **$ctfind_paths,** enclosing each path in double quotes; it also performs simple validations on the paths received.

4. **Versions_by_Date.prl** calls **cleartool lshistory**, passing **$ctfind_paths** as the paths parameter, and with a **-fmt** parameter to return the necessary information.

5. **Versions_by_Date.prl** 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 **Versions_by_Date.prl** procedure must conform to the 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 ClearCase.

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 available in **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **example1.prl.**

```
$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;"} = "%d;%";
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 Report Directory Organization, Report Description, and Report Output

The Elements with New Versions Since report lists all new versions, for a used-defined pathname, since a user-defined date and time. This report includes the following columns:

- Element Path
- Version Path
- Version Creation Time

The changes to this report 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.prl**

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 in **Elements_with_New_Versions_Since_Date.prl**, this procedure yields two parameters:

   ```
   LOOKIN
   ```

   ```
   CCTIME
   ```

   The mechanics for the **LOOKIN** parameter are described in *Example 1: Adding a Column to Report Output*. When **Elements_with_New_Versions_Since_Date.prl** receives **CCTIME,** it is a string of the following form:

   ```
   CCTIME = "time"
   ```

   This parameter specifies the times that the **cleartool find** command uses.

2. When **Elements_with_New_Versions_Since_Date.prl** 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. **Elements_with_New_Versions_Since_Date.prl** opens a pipe from a **cleartool find -print** command, with the converted **cctime** value passed in as a **created_since(<cctime>)** string. **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** are returned, **Elements_with_New_Versions_Since_Date.prl** 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. **Elements_with_New_Versions_Since_Date.prl** 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, **Elements_with_New_Versions_Since_Date.prl** 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 available at **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **example2.prl.**

```
$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 Report Description, Parameter Types, and Report 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 with 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 executes **Elements_Created_by User.prl** for the **-i** response, this report procedure yields two parameters:

   ```
   LOOKIN
   ```

   ```
   USER
   ```

   The mechanics for the **LOOKIN** parameter are described in *Example 1: Adding a Column to Report Output on page 215***.** When **USER** is received by **Elements_Created_by_User.prl** as a string of the following form:

   ```
   USER= "user-name"
   ```

   This parameter specifies the user name that the **cleartool** subcommand uses.

2. The **USER** string is extracted and stored as **$ccuser** passes it to the **created_by($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. Print the user variable 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 **Elements_Created_by_User.prl** to correspond to required interface changes.

2. Change the logic in **Elements_Created_by_User.prl** to handle data requests for group information; add a **%Gu;** to the **-fmt** parameter in the **cleartool describe** call,to get group information from ClearCase.

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's 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 available at **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **example3.prl.**

```
$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 Shortcut Menu for the Rightclick Handling Mechanism

The Elements with Labels report lists all elements with labels for a user-defined pathname. This report includes one column:

- Element Path

The change to this report adds the **Compare with Previous Version** command to the shortcut menu. Currently, these commands appear on the shortcut 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 shortcut 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 change this behavior for all reports whose primary sort key is **element** or **element_xpn**, modify the rightclick routines supplied with ClearCase Reports in **\script_rightclick** and then edit the associated routine in **\script_tools\common.prl.**

This example changes the rightclick behavior for the script, **Elements_with_Labels.prl.** To make this change , 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 of **Elements_with_Labels.prl.** Then, add a declaration to display the new **rightclick** item.

Modified Report Procedure

Here is the modified version of **Elements_with_Labels.prl**. This report procedure is also available at **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **example4.prl.**

```
$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 the Report Viewer Shortcut Menu

The Elements with Branches report lists all elements associated with the user-defined branch for a user-defined pathname. 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 shortcut menu. This command is not supplied with ClearCase Reports, so the work required to included it is different from that in *Example 4: Changing the Shortcut Menu for the Rightclick Handling Mechanism*.

These commands currently appear on the shortcut 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 routine for the new command functions.

When adding a new **rightclick** script, you must place it 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 **Elements_with_Branches.prl** interface specification. For example, to create a **rightclick** custom script called **my_rc.prl** that 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 also available at **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **example5.prl.**

```
$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 shortcut menu command for starting Merge Manager. This report procedure is also available at **http://clearcase.rational.com/contrib/T0046/T0046.zip** as **my_rc.prl.**

```
# 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";
```

C.5 Troubleshooting

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 **\right_click**.
- 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 **ifaces.prl** testing script. This script checks customized report procedures that have been written in ccperl. It is available at **http://clearcase.rational.com/contrib/ T0046/T0046.zip.**

To start the ifaces.prl testing script, use a command of this form:

ccperl ifaces.prl <*path to script or directory tree*>

We recommend that you test your custom report procedures before checking them in to the shared directory tree that you have configured.

If you do not run **ifaces.prl** before using your custom report in Report Builder and a parsing error occurs in processing the interface specification, the new report does not appear in the list of reports in the reports pane. There is no feedback; you see the report desciption in the reports pane or you see nothing. If you don't 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, referencing a nonexistent rightclick routine, or sending output in the wrong order to STDOUT.

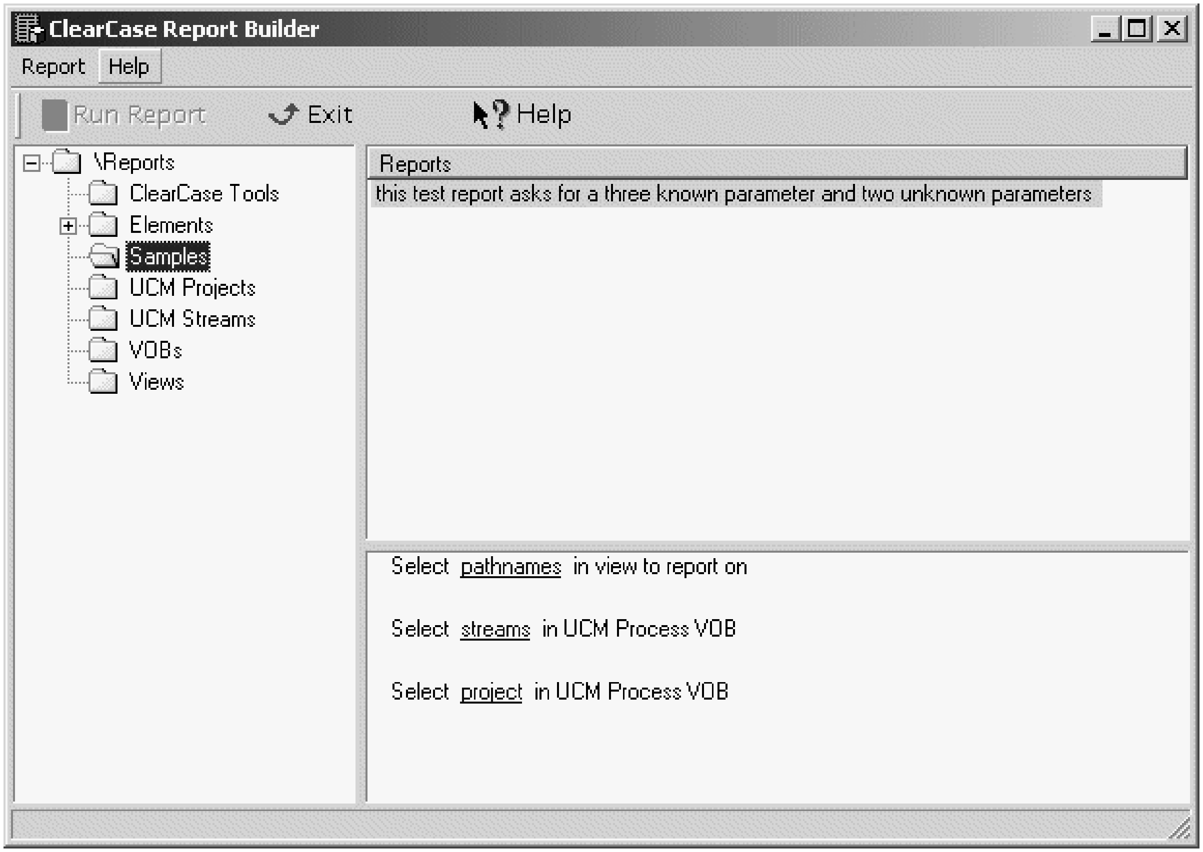
The Report Builder does not check for valid parameters. For example, consider the interface specification for a new report procedure, **my_custom_report.prl,** with the following interface specification:

```
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 (see Figure 59).

Figure 59 Report Builder Window with Invalid Parameters



However, the **ifaces.prl** interface checker detects these errors because these parameter names are not supplied with ClearCase Reports (as described in Table 6):

```
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
```

C.6 Coding High-Level Languages Other Than ccperl

When coding report procedures in languages other than ccperl, such as Visual C++, Java, Javascript or Visual Basic, refer to **http://clearcase.rational.com/contrib//T0046/T0046.zip** for programming samples.

Index

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

V

W