ClearCase Comprehensive

Module Five

Braching and Merging

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood,
and I—I took the one less traveled by,
And that has made all the difference.

Robert Frost

5.1 Branching and Merging in ClearCase

One of the criticisms that is often leveled at ClearCase is that it does not enforce any specific branching model or protocol, which means that anyone who knows the appropriate ClearCase commands can execute branches and merges. Often with chaotic results.

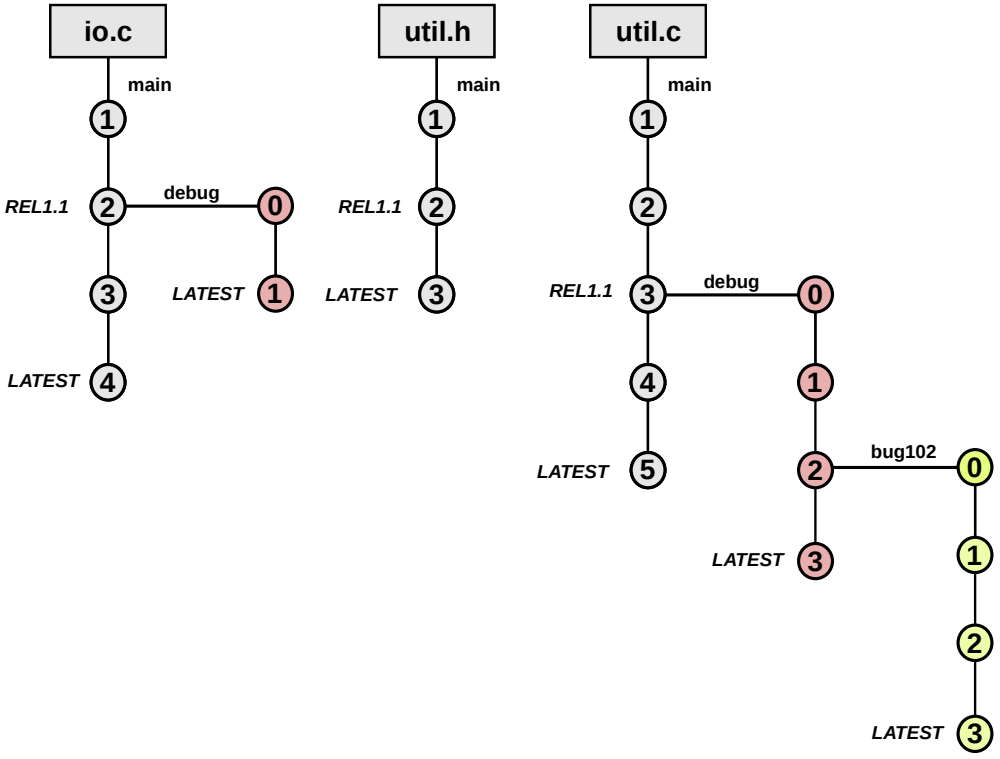
However, the design philosophy of ClearCase is that branching and merging is a planned and managed activity – the models and protocols used are the responsibility of the project administration, not the product. In one sense, it is analogous to supplying a knife to a cook, it is the cook who is responsible for using the right knife, using it correctly and safely, it's not the knife's responsibility.

One of the primary strengths of the ClearCase approach is that it's very flexible and can accommodate most methodologies so that the product is process agnostic. However that does shift the responsibility of making branching and merging work effectively to the project administrator since this is not in any way a responsibility of the enterprise ClearCase administrator.

In the previous module, we emphasized that the branching operation created a copy of a file, which is the best wy to think of it even if it is not quite accurate at an implementation level. When we say we make a branch, it sounds misleading because to be strictly correct, we need to say that we “branch a file.” Branching is the process of creating a copy of a file and then attaching a label to it.

5.1.1 Source Trees

Expanding in the example from a previous module, consider the there elements with their associated version trees:



Each view will show three elments in the “source tree” but which versions will depend on the config specs

```
element * CHECKEDOUT
element * /main/LATEST
```

Will produce:

io.c@@/main/4

util.h@@/main/3

util.c@@/main/5

```
element * CHECKEDOUT
element * .../debug/LATEST
element * /main/LATEST
```

Will produce:

io.c@@/main/debug/1

util.h@@/main/3

util.c@@/main/debug/3/

```
element * CHECKEDOUT
element * .../big102/LATEST
element * /main/LATEST
```

Will produce:

io.c@@/main/4

util.h@@/main/3

util.c@@/main/debug/bug102/3

```
element * CHECKEDOUT
element * .../bug102/LATEST
element * .../debug/LATEST
element * /main/LATEST
```

Will produce:

io.c@@/main/debug/1

util.h@@/main/3

util.c@@/main/debug/bug102/3

```
element * CHECKEDOUT
element * REL1.1
element * /main/LATEST
```

Will produce:

io.c@@/main/2

util.h@@/main/2

util.c@@/main/debug/3

```
element * CHECKEDOUT
element * .../debug/LATEST
element * REL1.1
element * /main/LATEST
```

Will produce:

io.c@@/main/debug/1

util.h@@/main/2

util.c@@/main/3

Based on the two commands that create a branch label and branch an element, couple with the selection capabilities of a view, allow us a great deal of power and flexibility.

Later in this module we will look at some best practices around branching and merging from the project administrator's point of view.

It was mentioned earlier that not accounting for the misconception that a branch is a complete new copy of a source tree can leave open the potential for misuse and disaster. One one particular project, one of my developers created a branch similar to 'altdev" and started altering a couple of files.

After several days, he realized that this particular development was not feasible, so in his altdev view he meticulously went through and deleted all the files from the source tree from version control. Unfortunately, most of the files he removed were in the main branch, but he thought they were only copies in his "branch".

After the repository had been restored from a recent backup, the project lead decided that he would institute some polices and procedures about making branches and removing files from version control.

5.2 Merging Concepts in ClearCase

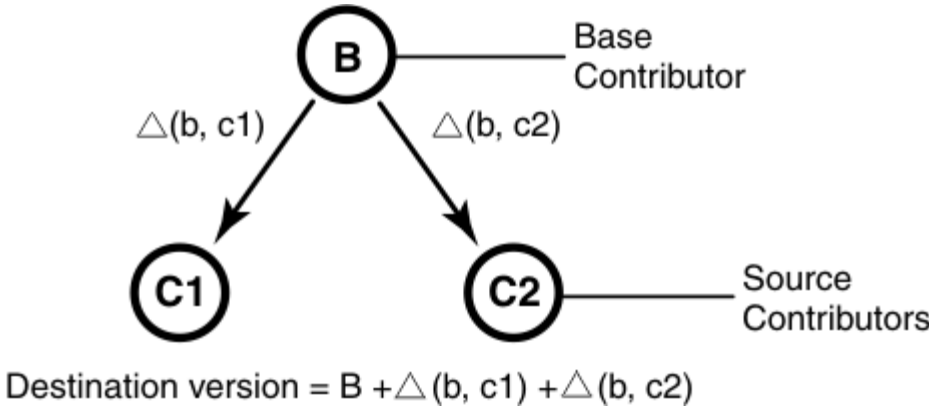
Merging, occurs in two situation in ClearCase:

1. When collisions occur during the check-in of an element from two different users.
2. When merging branches of an element or consolidating versions within a branch.

Irrespective of the situation, the strategy used by ClearCase is the same. Like most of ClearCase, the product does not attempt to figure out what a “correct” merge should be, but rather works generally under the assumption that dangerous mergers (i.e. ones that corrupt the elements under version control) are avoided through efficient and effective policies and procedures.

5.2.1 The Merge Algorithm

All branches must merge. The way a merge works in ClearCase is in terms of a base contributor and the deltas from the base contributor to the versions to be merged. Suppse that we have a base contributor B which has two branches where C1 and C2 represent the versions to be merged.



The final result of the destination version is created by starting with the base versions and adding all of deltas that created the two versions to be merged. According to the ClearCase documentation:

To merge versions, Rational ClearCase takes the following steps:

1. *It identifies the base contributor.*
2. *Next, it compares each contributor against the base contributor.*
3. *For any line that is unchanged between the base contributor and any other contributor, Rational ClearCase copies the line to the merge output file.*
4. *For any line that has changed between the base contributor and another contributor, Rational ClearCase performs a trivial merge by accepting the change in the contributor.*

5 *For any line that has changed between the base contributor and more than one other contributor, Rational ClearCase requires that you resolve the conflicting difference.*

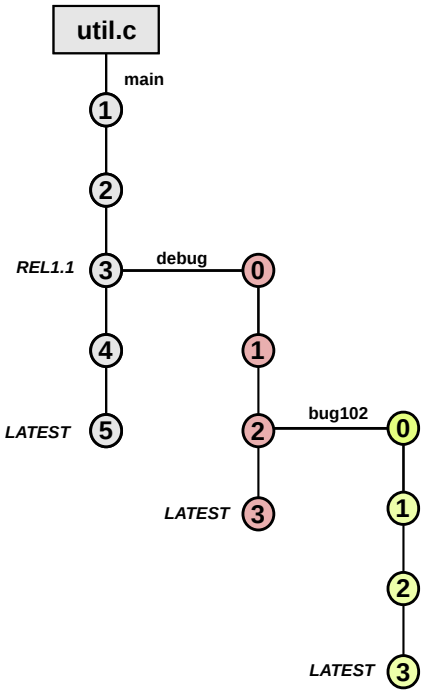
To claify this a bit further. There two different kinds of merges mentioned are defined this way:

Trivial merges. *In a trivial merge operation, the base contributor and the contributor to which you are merging are the same. In the comparison and merge tools, the base contributor is the element against which all other contributors are compared when reporting differences. Rational ClearCase resolves any trivial merges without your intervention.*

Nontrivial merges*. Differences between the base contributor and contributors are such that your manual intervention is required to resolve the differences. When encountering such differences, Rational ClearCase will prompt for your resolution.*

5.2.2 Base and Target Contributors

The base contributor, except in some exceptional cases, is the most recent common ancestor of the versions to be merged. The target contributor is the version that will incorporate the results of the merge (we usually say it is the version we “merge to”).



Merging the lastest version of the bug102 branch to the latest version of the debug branch would make **/main/debug/2** the base contributor and **/main/debug/3** as the target contributor.

Merging the latest version of the debug branch to the latest version of the main branch would make **/main/3** the base contributor and **/main/5** the target contributor.

5.2.3 Planning for Successful Merges

One of the goals of merge planning is that the various deltas (i.e. Δ(B,Cn) sets), should be disjoint as possible so that there are as few collisions between the deltas as possible. There are several ways that this can be accomplished:

1. **Task Allocation:** By planning out the development or modification tasks, each developer should be working on a separate part of the base contributor file so that multiple developers are not trying to modify the same segment of the base contributor file.
2. **Change Planning:** By examining the impact of each change on the base contributor, conflicts are resolved at design time rather than after coding. This often requires more up front analysis but almost inevitably results in smoother merges with fewer collisions.
3. **Smaller Merges:** When a series of changes may result in collisions, dividing up the changes into smaller increments and merging after each increment usually reduces the number of collisions, especially when a sequence of changes are going to impact the same segment of the base contributor.

5.3 Primary Branching Patterns and Strategies

This section is not ClearCase specific but rather looks at the commonly used branching strategies. Remember that there is not “correct” branching strategy, but most development projects do find that they are a good match for these basic patterns.

In this section we will use the term “branch” in a generic sense rather than referring to ClearCase branches.

In general, a branch is a line of development that exists independently of another line of development but shares a common history at some point in the version tree. Branches always begin as copies of something.

Note: In the material that follows, we will be using the term “branch” in to mean a collection of element versions that share a common branch label. While it does sound like we are treating branches as “things,” something we warned about earlier in this module, we don't mean to imply that, we are just following common usage.

There are two very common branch models in use. These are:

1. **Release Branching:** All development is done on the main branch.
2. **Feature Branching:** All development work is done in non-main branches.

We will now look at each of these in turn.

5.3.1 Release Branching

Release branching is typically done when regular releases of an application are produced. After each release, there are two independent lines of development that have to be supported:

1. **Next Release Development:** Starting from the last release, new features and enhancements are incorporated for the next release.
2. **Current Release Support:** While the current release is in production, on-going bug fixes have to be made to the production version.

Typically, the work flow looks like this:

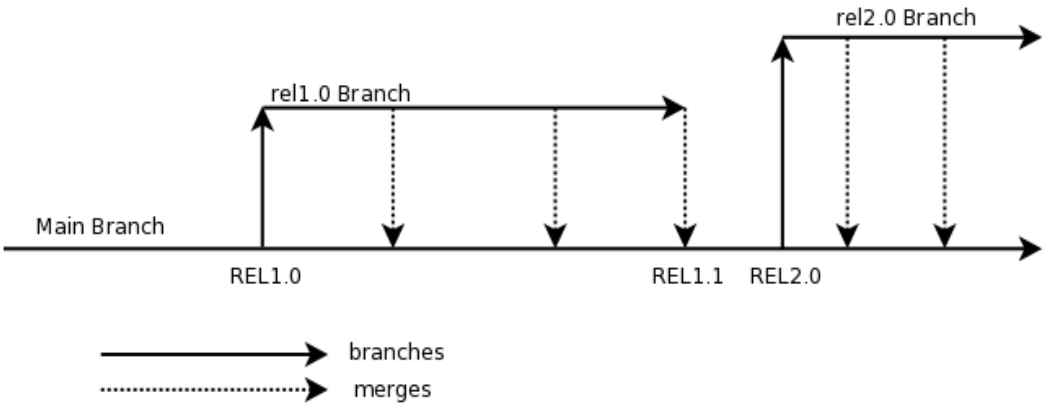
1. Developers commit all new work to the main branch whether they are new features or bug fixes.
2. When application is “released,” all current files are labeled with a release tag, suppose it's REL1.0
3. After labeling, all files are branched to a release branch which we will suppose is labeled rel1.0
4. Parallel development begins. The development team begins working on the main branch towards release 2.0. The support team begins applying bug fixes to the rel1.0 branch.

5 At a planned milestone, the current versions of the rel1.0 files are labeled REL1.1 and a minor release is made. The bugfix work continues of the rel.1 branch.

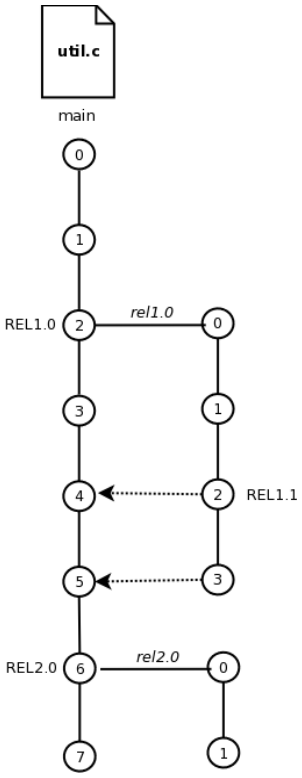
6 If it hasn't already happened, the rel1.0 branch in merged back into the main branch at when release 1.1 is made. Generally though, a number of smaller merges have been done to incorporate the bug fixes back into the production code.

7 When release 2.0 is ready, a final merge from the big fix branch is made and the process begins again.

The basic strategy looks like this:

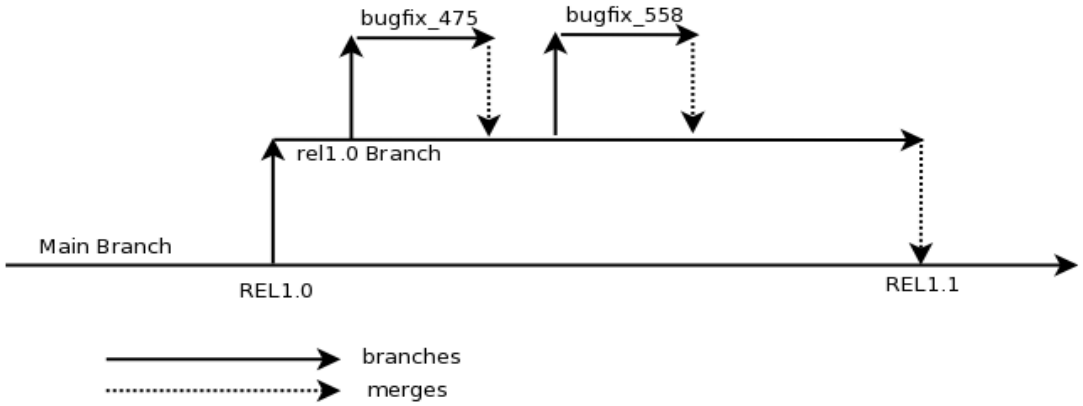


Remember that this is the strategy. For a particular element, the version tree looks something like this (assuming the element has versions in all the branches).

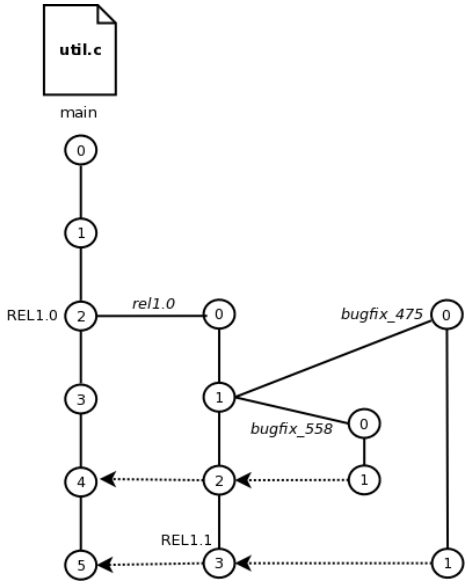


Notice that the tags allow us to build release 1.0, 1.1 or 2.0. The first merge back to the main branch is because of the REL1.1 milestone. The second merge is because of the REL2.0 milestone.

Of course this pattern can be repeated recursively as follows.



Where each bug fix is done on a separate branch then merged back into the production support branch. In this case, we may see an element version tree like this:



In this case we probably have allocated bugfix_475 to one developer and bugfix_558 to another developer who are working in parallel.

5.3.2 Feature Branching

Feature branching is the converse of release branching. The main branch is only changed via merges from feature branches and by bug fixes. A feature branch is a temporary branch created to work on a complex change without corrupting or changing the main branch code until the feature is finished, tested and ready to be incorporated.

In release branching, each release branch remains active since it is a snapshot of a particular release. Feature branches however, are always merged back into the main branch and can be considered as expendable once they have been merged.

The basic work flow looks like this:

1 The main branch is initiated.

2 An enhancement (call it feat87) is planned and a branch created.

3 While code for feat87 is being developed, bug fixes are being made to the main branch.

4 At various points, the main branch is merged to feature branches to keep the feature branches synced with the main branch.

5 When feat87 is completed, it is built and tested.

6 If it passes the test, it is then merged back into the main branch.

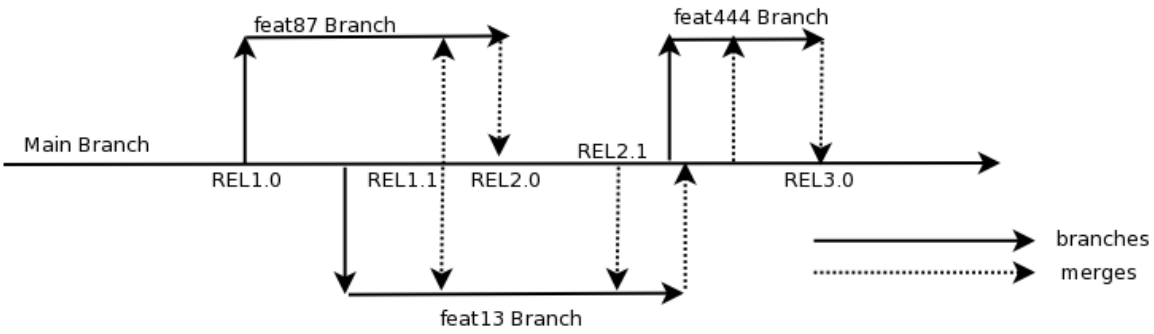
7 The new merged main branch is then regression tested.

8 If the regression test passes, the feat87 branch is marked as terminated. And the new build is possibly assigned a new release value.

Feature branching is always a bit trickier that release branching because of two factors.

1. If bug fixes are not merged and tested into the feature branch before the feature branch in merged back into the main branch, the feature branch may overwrite or undo some of the bug fixes.
2. Merging several feature branches carelessly, or planning the feature development carelessly, may cause collisions when the multiple feature branches are merged back into the main branch. This can often result in wasted time as the feature branches are reworked for compatibility.

A typical feature branching strategy looks like this:



This diagram depicts the following project plan.

1. The project is currently at release 1.0, and all current versions of all elements are tagged as REL1.0
2. Three new features are going to incorporated:

   feat87: planned for release 2.0

   feat13 and *feat444*: planned for release 3.0
3. *feat13* and *feat87* are both created to develop the new features.
4. Release 1.1 fixes a number of bugs in release 1.0, All modified latest versions are tagged REL1.1 in the main branch.
5. *feat87* and *feat13* both merge the main branch into their branches to ensure the feature branches have the bug fixes.
6. Any collisions between the bug fix code and feature code is resolved.
7. The updated feature branches are regression tested.
8. *feat87* is finished and merged back into the main branch.
9. The main branch is tested and regression tested. It passes and the current modified (or all) latest versions in main branch are all tagged REL2.0
10. Bug fix release 2.1 is official
11. *feat13* branch merges the bug fixes from release 2.1

12 *feat444* branch is started.

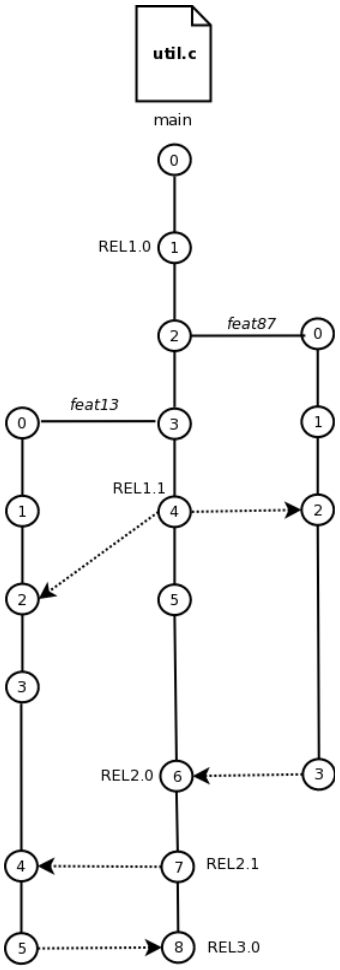
13 *feat13* is finished and merged back into the main branch The main branch is tested and regression tested. It passes.

14 The main branch is merged to *feat444* branch to keep the two branches synced.

15 *feat444* is finished and merged back into the main branch. The main branch is tested and regression tested. It passes and the current modified (or all) latest versions in main branch are all tagged REL3.0

On the next page is a diagram of the version history of **`util.c`** which is used in both feature 87 and feature 13 branches as well as getting several bug fixes

Notice that it is very important to keep the branches synced or else development can start to collapse in on itself. When using feature branching, process discipline is essential to that all branches and merges are performed on schedule and correctly.



5.4 Other Branching Patterns

The two primary branching patterns we looked at in the previous section are ***functional*** branching patterns. Here are four other main branching patterns.

These are:

1. **Physical:** Branching is done on the basis of the system's physical configuration so that different branches are created for specific components and subsystems. This is especially useful when there is sharing or reuse of components or subsystems
2. **Environmental:** Different branches are created to accommodate different target platform, third party libraries, operating systems, etc.
3. **Organizational:** Different branches are created based on task or project organization, such as branches for specific sub-projects, roles, and groups. One place that we see organizational branching is when some portion of the project is outsourced.
4. **Procedural:** Branching is based on organization or team policies. Branches are often created to support different policies (e.g. some branches may require a security classification by the developer), processes (e.g a fast track urgent bug fix), and states (different branches may reflect different business rules in different countries the software will be used it)

Generally these branching patterns are part of a release management strategy that also requires a careful planning of source directories, code partitioning and appropriate build or make scripts.

For example, assume we are developing an application that will run under Windows and Unix. The application has two main code sections, a core that is common to both versions, and a set of GUI files that provide the code for Windows and the Unix X-windows environment respectively.

There are at least three possible strategies for building the different targets.

1. Within each file we can have conditional sections if possible, one for Windows and another for Unix. This is the worst possible solution.
2. We can have three directories, one common, one for Windows specific code and one for Unix specific code. The makefile we run chooses the appropriate directory.
3. We create two branches of a GUI directory, one for Unix and one for Windows. We also create a Unix view and a Windows view that select the appropriate branches. Then the same makefile will create a Windows or Unix release depending on the view in which it is run.

Either 2 or 3 or a hybrid of the two would be an acceptable solution – which one would be better would depend on the other factors as both the project level and the CM level.

5.5 Merging Scenarios

Merging is an activity that should never be taken on the spur of the moment. Some best practices for merges are:

1. Merges should be scheduled, they should never be spontaneous.
2. Each team member should know exactly what they need to do to get ready for a merge, for example, close off any current work being done.
3. Before the merge, a full backup should be run.
4. Before the merge, each team member should sign off that any assets they are responsible for are in a state that is ready for the merger.
5. The merge logic should be scripted and the script reviewed, inspected and debugged before the merge.
6. All work is suspended during the merge.
7. After the merge, all team members should validate the state of their assets.

In other words, merging is serious business and a major event. Work should not resume until everyone has verified that the merge worked exactly as it should have.

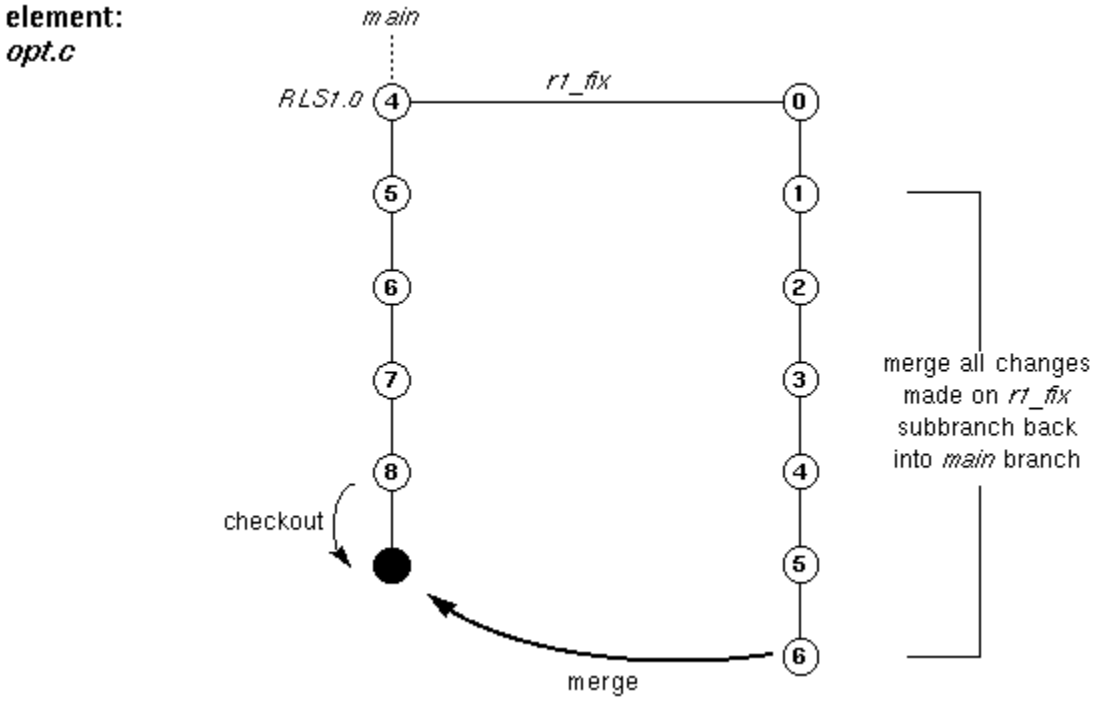
ClearCase has a rich set of merging commands that allow for a number of merging scenarios. Because this set of commands is so rich and powerful, it is easy to create very effective merges that essentially merge anything to anything else, but it is also easy to totally scramble a version tree with a poorly phrased merge command.

Merge commands should be thoroughly designed as apposed to being written quickly.

The following are some examples from the SGI ClearCase admin guide for different kinds of merges.

5.5.1 Scenario: Merging All the Changes Made on a Subbranch

Bugfixes for an element named opt.c are being made on branch r1_fix, which was created at the baselevel version RLS1.0 (/main/4). Now, all the changes made on the subbranch are to be incorporated back into main, where a few new versions have been created in the meantime.

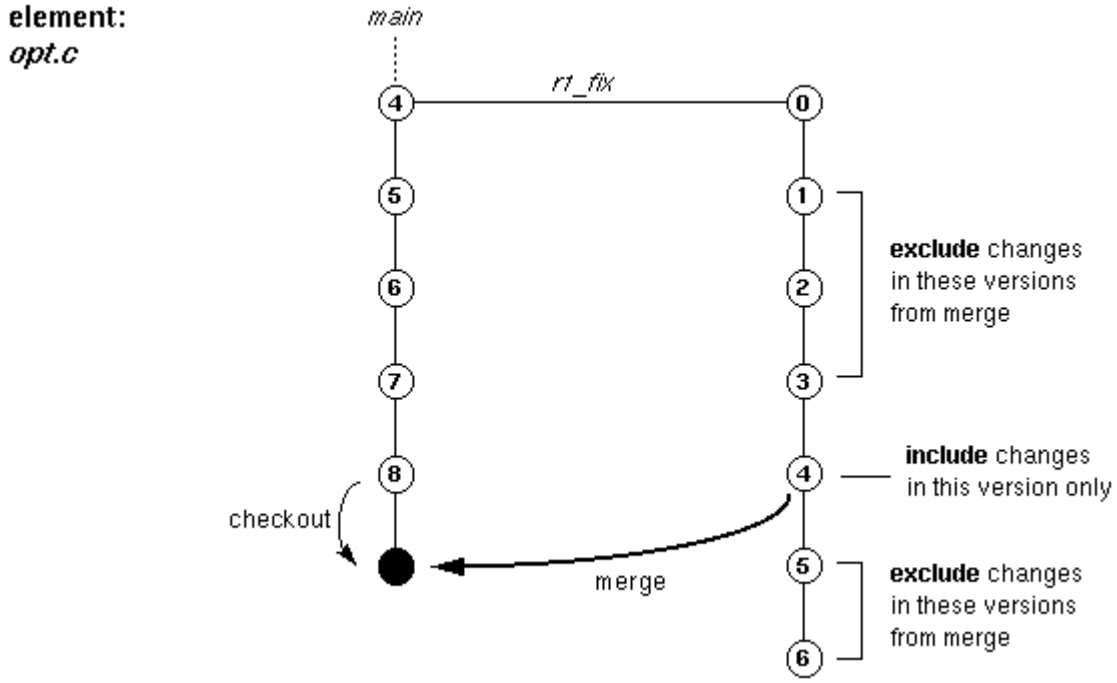


Set a view configured with the default config, go to the source directory and enter this command to perform the merge:

```
% cleartool findmerge opt.c -fversion .../r1_fix/LATEST
```

5.5.2 Scenario: Selective Merge from a Subbranch

This a variant of the preceding merge scenario. The project leader wants the changes in version /main/r1_fix/4 (and only that version — it's a particularly critical bugfix) to be incorporated into new development. In performing the merge, you specify which version(s) on the r1_fix branch to be included.



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

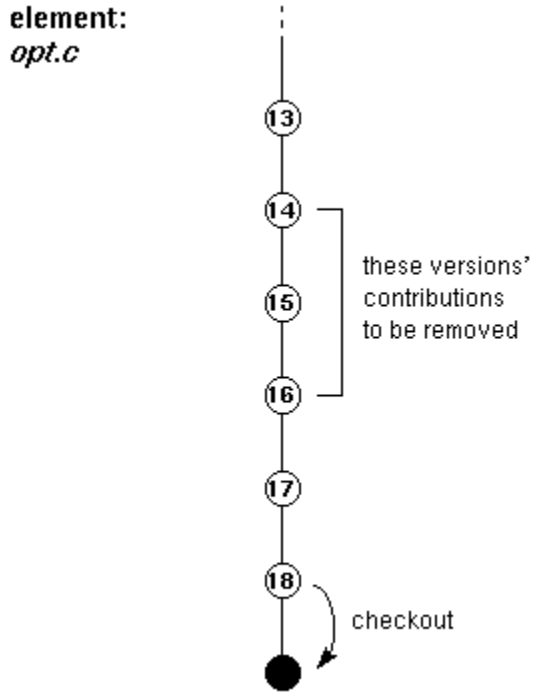
```
% cleartool checkout opt.c
% cleartool merge -to opt.c -insert -version /main/r1_fix/4
```

You can also specify a range of consecutive versions to be merged. For example, this command merges selects only the changes in versions /main/r1_fix/2 through /main/r1_fix/4:

```
% cleartool merge -to opt.c -insert -version /main/r1_fix/2
/main/r1_fix/4
```

5.5.3 Scenario: Removing the Contributions of Some Versions

The project leader has decided that a new feature, implemented in versions 14 through 16 on the main branch in Figure 6-5, will not be included in the product. You must perform a “subtractive merge” to remove the changes made in those versions.



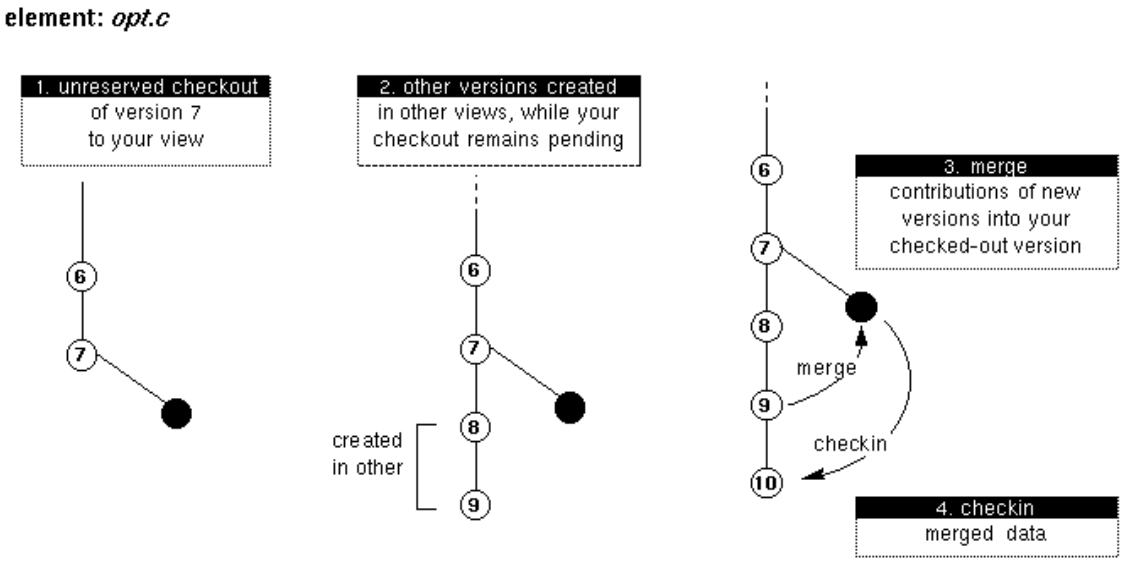
Enter these commands to perform the subtractive merge:

```
% cleartool checkout opt.c
% cleartool merge -to opt.c -delete -version /main/14 /main/16
```

5.5.4 Scenario: Merging an Unreserved Checkout

ClearCase allows the same version to have several checkouts at the same, each in a different view. At most one of the checkouts is reserved — all the others (or perhaps every one of them) is unreserved. This mechanism allows several users to work on the same file at the same time, without having to use separate branches.

To prevent confusion and loss of data in such situations, ClearCase imposes a constraint: if the version from which you performed an unreserved checkout (version 7 below) is not the most recent version on the branch, then you cannot simply check-in your work — this would obliterate the contributions of the versions created in the interim (versions 8 and 9 i below 6). Instead, you must first merge the most recent version into your checked-out version, then perform the check-in.



Enter these commands to merge your unreserved checkout:

```
% cleartool merge -to opt.c -version /main/9
% cleartool checkin opt.c
```

5.5.5 Scenario: Merging All of a Project's Work

In the preceding scenarios, a merge was performed on a single element; now for a more realistic scenario. Suppose a team of developers has been working in isolation on a project for an extended period (weeks or months). Now, your job is to merge all the changes back into the main branch.

The findmerge command has the tools to handle most common cases easily. It can accommodate the following schemes for isolating the project's work.

5.5.5.1 All of Project's Work Isolated "On a Branch"

In one standard ClearCase approach to parallel development, all of a project's work takes place "on the same branch". More precisely, new versions of source files are all 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 "mainline" view, configured with the default config spec:

```
% cd root-of-source-tree
% cleartool findmerge . -fversion .../gopher/LATEST -merge -xmerge
```

The -merge -xmerge syntax causes the merge to take place automatically whenever possible, and to bring up 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.

5.5.5.2 All of Project's Work Isolated "In a View"

Some projects are organized so that all changes are performed in a single view (typically, a shared view). For such projects, use the -ftag option to findmerge. Suppose the project's work has been performed in a view whose view-tag is goph_vu. These commands perform the merge:

```
% cd root-of-source-tree
% cleartool findmerge . -ftag goph_vu -merge -xmerge
```

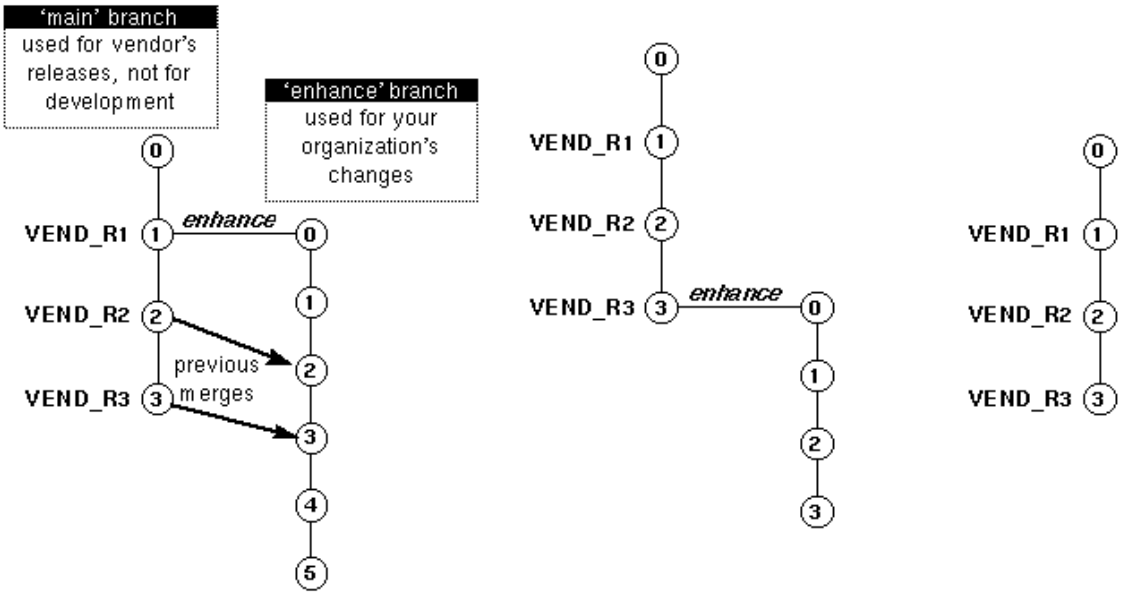
5.5.6 Scenario: Merging a New Release of an Entire Source Tree

Here is another project-level merge scenario as shown below. Your group has been using an externally-supplied source-code product, maintaining the sources in a VOB. The successive versions supplied by the vendor are checked into the main branch and labeled VEND_R1 through VEND_R3. Your group's fixes and enhancements are created on subbranch enhance. The views in which your group works are all configured to branch from the VEND_R3 baselevel:

```
element * CHECKEDOUT
element * .../enhance/LATEST
element * VEND_R3 -mkbranch enhance
element * /main/LATEST -mkbranch enhance
```

The version trees illustrated below shows three different likely cases:

1. an element that your group started changing at Release 1 (enhance branch created at the version labeled VEND_R1)
2. an element that your group started changing at Release 3
3. an element that your group has never changed



Now, a tape containing Release 4 arrives, and you need to integrate the new release with your group's changes. After you add the new release to the main branch and label the versions VEND_R4, it will be easy to specify the merge process: “for all elements, merge from the version labeled VEND_R4 to the most recent version on the enhance branch; if an element has no enhance branch, don't do anything at all”

Here's a complete procedure for accomplishing the integration:

Load the vendor's “Release 4” tape into a standard UNIX directory tree:

```
% cd /usr/tmp
% tar -xv
```

Suppose this creates directory tree mathlib_4.0.

As the VOB owner, run the UNIX-to-ClearCase conversion utility, clearcvt_unix, to create a script that will add new versions to the main branches of elements (and very likely, to create new elements, as well).

```
% cd ./mathlib_4.0
% clearcvt_unix
 . (lots of output)
```

In a view configured with the default config spec, run the conversion script created by clearcvt_unix, creating Release 4 versions on the main branches on elements:

```
% cleartool setview mainline
% cd /vobs/proj/mathlib
% /usr/tmp/mathlib_4.0/cvt_dir/cvt_script
```

Label the new versions:

```
% cleartool mklbtype -c "Release 4 of MathLib sources" VEND_R4
Created label type "VEND_R4".
% cleartool mklabel -recurse VEND_R4 /vobs/proj/mathlib
 . (lots of output)
```

.

Go to a view that is configured with your group's config spec, selecting the versions on the enhance branch:

```
% cleartool setview enh_vu
```

Merge from the VEND_R4 configuration to your view:

```
% cleartool findmerge /vobs/proj/mathlib -fver  VEND_R4 -merge
-xmerge
```

The -merge -xmerge syntax says “merge automatically if possible; but if not possible, bring up the graphical merge tool”.

Verify the merges, and checkin the modified elements.

You have now established Release 4 as the new baselevel. Developers in your group can update their views' configurations as follows:

```
element * CHECKEDOUT
element * .../enhance/LATEST
element * VEND_R4 -mkbranch enhance
element * /main/LATEST -mkbranch enhance
```

ClearCase Comprehensive

Module Six

Project Configuration

If history repeats itself, and the unexpected always happens, how incapable must Man be of learning from experience.

George Bernard Shaw

Those who cannot remember the past are condemned to repeat it.

George Santayana

Good planning is the key to success and half the effort.

6.1 What is Project Configuration?

Up until now, we have been looking at ClearCase from strictly a functional perspective, and now we turn to the project level.

Project configuration might be defined as the set of activities that must be performed during the project planning phase to ensure a robust and transparent ClearCase deployment that both leverages develop efforts and mitigates project risk.

In the material that follows, we will look at both the theory and how these plans are implemented in ClearCase.

One of the ongoing problems that we have at the theoretical level is that we cannot separate the work we do for ClearCase project configuation with the larger organizational CM and other practices in the organization.

For example, the following policies will have an impact

1. Security policies
2. Audit requirements
3. Records management policies and requirements
4. Litigation holds and legal requirements.

The following section will start with the more general concept of an SCM plan and then drill down into how we would implement a number of these policies in a ClereCase environment.

6.2 Planning for ClearCase[1]

ClearCase implements the SCM policies and procedures of the project. In order to develop these, there should be a planning process that includes:

1. Understanding and documenting current software development organization, processes, and procedures.
2. Performing process reengineering to eliminate problems.
3. Establishing clearly defined roles and responsibilities.
4. Establishing policies that govern the ways in which each development environment will function.
5. Identifying how ClearCase and other products will help in the software development life cycle.
6. Creating an SCM plan. Software teams should not attempt to implement the ClearCase architecture and design within the production development environment until an SCM plan, policies, and procedures are in place.

The clear vision that results from this planning process allows organizations to architect and design the following items:

1. Hardware and software requirements
2. ClearCase and ClearQuest integration with other development tools
3. An SCM environment using either Base ClearCase or Unified Change Management (UCM)
4. ClearCase objects (VOBs, projects, promotion levels, and so forth)
5. Individual or shared environment areas
6. A plan for populating the ClearCase SCM environment
7. Client and server patch level updates
8. A production baseline and a baseline naming convention
9. An employee training plan
10. An implementation plan for development tools (WebSphere Studio, Rational XDE™, or Eclipse) that can carry development efforts through to production (these tools allow managers to control when and what code is moved into production).

The next few pages represent a sample configuration management plan.

1 From IBM's SCM for ClearCase – pdf distributed with class.

Software configuration management plan

Before you go into the details, you might want to add a front page and some revision history.

Remember, the following is an example template for the layout of an SCM plan. Use it as an aid in creating your SCM plan and as you see fit. You decide the level of detail that you want to put into the plan.

Template

1. Introduction

The introduction of the Software Configuration Management Plan provides an overview of the entire document. It includes the purpose, scope, definitions, acronyms, abbreviations, references, and overview of this Software Configuration Management Plan.

1.1 Purpose

Here you specify the purpose of this Software Configuration Management Plan.

1.2 Scope

This subsection provides a brief description of the scope of this Software Configuration Management Plan; what model it is associated with, and anything else that is affected by or influenced by this document.

1.3 Definitions, Acronyms, and Abbreviations

This subsection provides the definitions of all terms, acronyms, and abbreviations required to properly interpret the Software Configuration Management Plan. This information may be provided by reference to the project's Glossary.

1.4 References

This subsection provides a complete list of all documents referenced elsewhere in the Software Configuration Management Plan. Identify each document by title, report number if applicable, date, and publishing organization. Specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.

1.5 Overview

This subsection describes what the rest of the Software Configuration Management Plan contains and explains how the document is organized.

2. The SCM framework

2.1 Organization, Responsibilities, and Interfaces

Describe who is going to be responsible for performing the various Software Configuration Management (SCM) activities described in the SCM Process Workflow.

2.2 Tools, Environment, and Infrastructure

Describe the computing environment and software tools to be used in fulfilling the CM functions throughout the project or product lifecycle.

Describe the tools and procedures required used to version control configuration items generated throughout the project or product lifecycle.

Issues involved in setting up the CM environment include:

- Anticipated size of product data
- Distribution of the product team
- Physical location of servers and client machines.

2.3 Administration and maintenance

Describe the backup and recovery strategies for the SCM implementation.

Describe miscellaneous tasks and procedures to be used for the administration and maintenance of your SCM installation, including license management and installation/upgrade processes.

3. The SCM process

3.1 Configuration Identification

3.1.1 Identification Methods and Naming Convention

Describe how project or product artifacts are to be named, marked, and numbered. The identification scheme needs to cover hardware, system software, Commercial-Off-The-Shelf (COTS) products, and all application development artifacts listed in the product directory structure; for example, plans, models, components, test software, results and data, executables, and so on.

3.1.2 Workspace Management

The workspace is where developers edit source files, build the software components they are working on, and test and debug what they have built. We can say that the workspace is a *development area associated with a change*.

Describe view and/or work area policies, types, naming conventions, and storage locations.

3.1.3 Project Baselines And Branching Strategies

Baselines provide an official standard on which subsequent work is based and to which only authorized changes are made.

Describe at what points during the project or product lifecycle baselines are to be established. The most common baselines would be at the end of each of the Inception, Elaboration, Construction, and Transition phases. Baselines could also be generated at the end of iterations within the various phases or even more frequently.

Describe who authorizes a baseline and what goes into it.

3.1.4 Promotion Model

The promotion model provides guidelines for when development work should be promoted into release or integration branches once development is completed. It also provides guidelines for when and how developers should synchronize their work with the recommended baseline.

3.2 Configuration and Change Control

3.2.1 Change Request Processing and Approval

Describe the process by which problems and changes are submitted, reviewed, and dispositioned. This should include the workflow diagrams for the different workflows used for the different development phases.

3.2.2 Change Control Board (CCB)

Describe the membership and procedures for processing change requests and approvals to be followed by the CCB.

3.3 Configuration Status Accounting

3.3.1 Project Media Storage and Release Process

Describe retention policies, and the back-up, disaster, and recovery plans. Also describe how the media is to be retained—online, offline, media type, and format.

The release process should describe what is in the release, who it is for, and whether there are any known problems and any installation instructions.

3.3.2 Reports and Audits

Describe the content, format, and purpose of the requested reports and configuration audits.

Reports are used to assess the “quality of the product” at any given time of the project or product lifecycle.

Reporting on defects based on change requests may provide some useful quality indicators and, thereby, alert management and developers to particularly critical areas of development. Defects are often classified by criticality (high, medium, and low) and could be reported on the following basis:

Aging (Time-based Reports): How long have defects of the various kinds been open? What is the lag time of when in the lifecycle defects are found versus when they are fixed?

Distribution (Count Based Reports): How many defects are there in the various categories by owner, priority or state of fix?

Trend (Time-related and Count-related Reports): What is the cumulative number of defects found and fixed over time? What is the rate of defect discovery and fix? What is the "quality gap" in terms of open versus closed defects? What is the average defect resolution time?

4. **Milestones**

   Identify the internal and customer milestones related to the project or product SCM effort. This section should include details on when the SCM Plan itself is to be updated.

5. **Training and Resources**

   Describe the software tools, personnel, and training required to implement the specified SCM activities.

6. **Subcontractor and Vendor Software Control**

   Describe how software developed outside of the project environment will be incorporated.

The specific tasks we want to investigate in this module are:

1 The organization and layout of the project VOBs.

2 Development of the branching and release management support.

3 Workspace definition and management (View Management)

4 Retention and retirement.

6.3 Planning VOBs

In this module, we are not going to look at the technical side of the VOBs but rather the planning that will be used in the technical implementation of the VOBs

6.3.1 Deciding on VOB content

In the first module we saw that not just source code can go under version control. For the project, decided what should be put under version control. Generally we want to only put things under version control that meet one or more of several criteria.

1. They will be modified on a regular basis. The resources of ClearCase are wasted by putting objects under version control that rarely change.
2. It is important to be able to synchronize versions of the artifacts with releases or milestones, such a specification documents and test cases.
3. We need to track changes to the artifact for audit or security purposes.
4. The artifact has associated milestones.

For example, progress reports should not be under version control since we would not expect to modify a progress report once it has been authored. On the other hand, test cases are often associated with code changes so they would be good candidate for version control.

The next question to be resolved is where do these artifacts go? Here are couple of typical strategies:

VOB by type: In this approach we would have a source code VOB, a test management VOB and a Spec and Design VOB where each would hold artifacts of a particular type.

VOB by component: In this approach we divide the project into sub projects and components and keep all artifacts related to that component in one VOB. We might have three main directories under the root named SRC, DOC and TEST for example and keep artifacts of each type under the appropriate directory.

The choice of which approach or some hybrid of these two will depend on the project methodology chosen – foe example with all development on the main branch we might choose the VOB by type approach while the development on feature branches may opt for a VOB by component approach depending on geographical location and how independent the development teams are.

6.3.2 Deciding on VOB Size

VOBs to have a performance limit which does depend on the size, number of elements and he amount of metadata in the VOB. It is impossible to give absolute limits because it does depend on hardware and other considerations

Here are the principal tradeoffs:

1. *Splitting data into several small VOBs greatly increases your flexibility: it is easy to move an entire VOB to another host; it is difficult to split a VOB into two parts and move one of them to another host.*
2. *Typically, having multiple small VOBs makes for fewer performance bottlenecks than having one large VOB.*
3. *Having fewer VOBs facilitates data backup.*
4. *Having fewer VOBs facilitates synchronizing label, branch, and other definitions across all the VOBs.*

6.3.3 Element Types

The ability to use element types can be helpful in setting up the VOB, especially when we can effectively employ user defined types.

Each VOB comes with a set of predefined element types. Each type has associated with a type_manager that does the actual versioning of the elements of that type and manages how the objects of that type are managed in the VOB.

The types that come with ClearCase are:

1. **file:** Versions can contain any kind of data (text, binary, bitmap, and so on). Uses the whole_copy type manager.
2. **compressed_file:** Versions can contain any kind of data. Uses the z_whole_copy type manager.
3. **text_file:** All versions must contain text (multibyte text characters are allowed). Null bytes are not permitted (a byte of all zeros); no line can contain more than 8,000 characters. Uses the text_file_delta type manager.
4. **compressed_text_file:** All versions must contain text; no line can contain more than 8,000 characters. Uses the z_text_file_delta type manager.
5. **binary_delta_file:** Versions can contain any kind of data. Uses the binary_delta type manager.
6. **Html:** Subtype of the binary_delta_file element type. Uses the binary_delta type manager.
7. **ms_word:** Subtype of the file element type. All versions must be Microsoft® Word files. Uses the _ms_word type manager.
8. **Rose:** Subtype of the text_file element type. Uses the _rose type manager.
9. **Rosert:** Subtype of the binary_delta_file element type. Uses the _rosert type manager.
10. **Xde:** Subtype of the text_file element type. Uses the _xde type manager.
11. **Xml:** Subtype of the text_file element type. Uses the _xml type manager.

12 **Directory:** Versions of a directory element catalog (list the names of) elements and VOB symbolic links. Uses the directory type manager, which compares and merges versions of directory elements.

13 **file_system_object:** Generic element type, with no associated type manager.

The type manager program is called when certrain operations are performed on an element. Some typical type manager are:

1 **whole_copy:** Stores any data. Stores a whole copy of each version in a separate data container file.

2 **z_whole_copy:** Stores any data. Stores each version in a separate, compressed data container file using the ccgzip compression program. (ccgzip is a utility that implements IETF RFC 1951 + 1952 compression using zlib source code). Note that compressed files generally take more time to check in (because they must be compressed) and to reconstruct when first accessed (first cleartext fetch).

3 **text_file_delta:** Stores text files only (including those with multibyte text characters). Stores all versions in a single structured data container file. (On UNIX and Linux, similar to an SCCS s. file or an RCS ,v file.) Uses incremental file differences to reconstruct individual versions on the fly.

4 **z_text_file_delta:** Stores text files only. Stores all versions in a single structured data container file, in compressed format using both the ccgzip compression program and deltas.

5 **binary_delta:** Stores any data. Stores each branch's versions in a separate, structured compressed data container file using ccgzip. Uses incremental file differences to reconstruct individual versions on the fly. Version deltas are determined by comparing files on a per-byte basis.

We can add additional type_managers and types however to do this we need to provide a type_manager which can be used by ClearCase to do the versioning.

6.3.4 User Defined Types

An more effective tool from the management perspective is to define subtypes of these types for our own organizational purposes. For example we could create a Java_Source type:

```
mkeltype –supertype text_file –nc Java_Source
```

The Java_Source type will use the text_delta_file type_manager because it is a subtype pf the text_file type.

So why bother? Because it can give us finer grained control in things like config specs.

```
mkeltype –supertype text_file –nc Java_Source
mkeltype –supertype ms_word –nc Design_Doc
```

```
element -eltype Java_Source *.Java REL2.0
element -eltype Design_Doc * /main/LATEST
```

6.3.5 Populating A VOB

Although we can add elements one by one to a VOB, this is not recommended. Instead we use an import utility to move an entire tree of artifacts into a VOB.

We know from experience that, unlike a regular file system, the meta-data, especially the directory met-data, makes the structure of the artifact tree more brittle than a regular file system.

For example consider

1. A file **x.txt** is in a directory **dir@@/main/1.**
2. We delete the file which results in a new directory version **dir@@/main/2**
3. However the file still exists in dir@@/main/1
4. We add a new file and increment to **dir@@/main/3**
5. Now we add a file **x.txt** to the directory to get **dir@@/main/4**.
6. Now we have a problem with x.txt and how the two instances are linked.S

Some basic rules for moving, adding, deleting, copying and renaming elements in a VOB.

1. Don't.

Instead the following procedure is recommended.

1. In an ordinary file system, create a source tree populated by all the files that will appear and be used in the project. These files are usually autogenerated stubs, or as in the case of Word files for example, empty documents generated from the appropriate templates.
2. The source tree is signed off by all of the development team stakeholders
3. The tree is imported into the VOBs using an import utility

6.3.6 Clearfsimport

The clearfsimport command reads the specified file system source objects and places them in the target VOB. This command uses magic files to determine which element type to use for each element created. The source and target directories may not be the same. The utility has a number of options to do things like follow symbolic links and recurse through the directory tree. The -preview option allows a dry run of the import to check for erorrs without actually wiring the data to the VOB.

```
clearfsimport –preview –follow –recurse /usr/src/projectx /vobs/projectx/src
```

6.3.6.1 File Typing

IN order to determine what kind of element a file is, the utility uses a cc_magic file which has typing rules to use to determine the element type.

For example:

```
directory : -stat d ;
c_source source text_file : -printable & -name "*.c" ;
sh_script script text_file : -printable & (-name ".profile" | -name
"*.sh") ;
archive library file: !-printable & -name "*.a" ;
```

6.4 VOB Life Cycles

As we use a VOB , the meta data accumulates. At some point, the meta-data is no longer of use to us beyond historical interest. VOBs should be retired when the meta-data is no lover of value to use. Unfortunately we tend to see VOBs used for project after project with not only meta-data being accumulated but also re-arrangements of the source tree.

The following is a recommended practice.

1. Identify a “re-VOBing” mile stone, perhaps at the release of a specific version for example.
2. The VOB used in release one is archived in some manner so that the met-data will be preserved to be available for audit and other functions. Generally the VOB should be locked to prevent any further modifications
3. An export view is created. The view should show us the VOB as we would want the source tree to look for our next development.
4. The source tree in the VOB is exported using normal file system copy utilities.
5. The exported source tree is reorganized and updated as required for the start of the next phase of the project.
6. The source tree is imported into the new VOB.

These milestones are also points at which we can reorganize our VOBs, for example we can split a large VOB into two smaller VOBs.

6.5 View and Branch Management

As we have seen, views let us see a View is a workspace. In the following section will review some of the Best Practices presented by Brad Appleton of mMotorola.

6.5.1 Project-oriented vs file-oriented branching

File-oriented branching (low-level)

- Branches are organized & viewed in the context of a single file and its version tree (vtree) for that one file
- Focuses primarily on physical modifications to individual files

Project-oriented branching (high-level)

- Branches are organized & viewed in the context of a project (or product) and its version tree (vtree) for the whole system
- Focuses primarily on the flow of logical changes/activities to entire components and (sub)systems

Project-oriented branching is more conceptually powerful

Project-wide vtree depicts the evolution of the whole system

Better conceptual fit for how we try to plan, manage & track projects and workflow

6.5.1.1 Tool support issues for branching & baselining

- Branching support: Meaningful names & hierarchical structure
- Merging support: Merge ancestry & graphical, multi-way merge assistance
- Labeling support: Labeling performance, label renaming
- Logical change-grouping: Change “sets” and “transactions”
- Derived object reuse
- Extensibility (hooks & triggers & UI)

6.5.2 ClearCase branching & labeling “best practices”

- Private development branch
- Shared development branch
- Early vs lazy branching
- Push vs pull integration
- Shared integration branch
- Pull-push integration-task branch
- Push-pull “docking” branch
- Integration lock

- Partial integration lock

6.5.2.1 Private development branch

- Branch-type created per development change/activity (just like a “private branch” with a “view profile” on NT)
- Branch-access restricted exclusively to the developer responsible for implementing the fix or feature
- Checkout/checkin as desired on the “private branch” (automatically grouping together all modifications made)
- Other developers insulated from seeing changes until the branch is “merged back” to integration branch
- Branch “lock down” when change/activity is completed