exgnosis

ClearCase Comprehensive

Student Manual

Author: Rod Davison
Document ID: Custom
Revision: 1.0
Course Ref: Custom
Date: 2014-02-01

Module One

Software Configuration Management

I should probably explain what I mean in distinguishing between software configuration management (SCM) people and people who build software systems. The stereotype is that configuration management people are concerned with tools and control. They are conservative, and they prefer slow, predictable progress. They are also "the few" as compared with "the many" developers in an organization. Software engineers (so the stereotype goes) are reckless. They want to build things fast, and they are confident that they can code their way out of any situation. These are extreme stereotypes, and in my experience, the good software engineers and the good release/quality assurance/configuration management people have a common goal:They are focused on delivering quality systems with the least amount of wasted effort

Brad Appleton

Don't it always seem to go
That you don't know what you got until it's gone.

Joni Mitchell

1.1 What ClearCase Does for Us

ClearCase is a tool that is used to automate source code configuration and artifact management. It is important to understand that ClearCase does not “do” configuration management, it only automates existing configuration management practices used by an organization or development team.

As an analogy, there are many people in this world who are sure they can write because they use a word processor, but what they tend to produce are documents that look beautiful in terms of their formatting which are unreadable at the content level. Writing is a skill that a person possesses irrespective of any tool used – whether writing is good or bad is independent of the authors skill with the word processor. But because a word processor automates many of the low-level, tedious and clerical tasks encountered in the writing process, a good writer will be more productive and efficient at their craft using a tool like a word processor because they can spend more of their time on the actual creative part of the the writing process.

The field of Configuration Management (CM) is not immune to this same mistaken belief, often articulated as “all we need to do for code management is just to make everyone use ClearCase.” Suppose an organization is experiencing CM problems and decides to look for a solution. Inevitably, the vendors of the various CM tools are quite convincing in their claims of how their tool will solve all of the organizations CM problems. The organization then deploys some CM tool under the belief that they are “doing configuration management” but nothing seems to really change except that their people now spend what should be productive work time fiddling around with the CM tool.

This sort of error is quite common. An expensive car does not make you into a good driver, a top of the line database server does not mean that you manage data effectively – and using ClearCase (or CVS or Subversion) does not mean that your are doing CM.

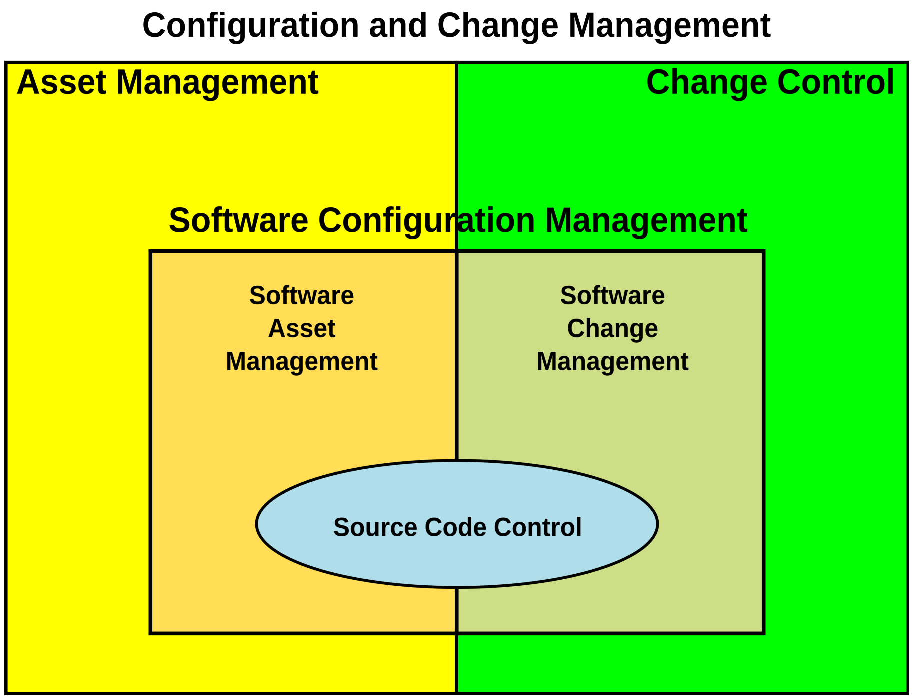
For the rest of this module, we will introduce some CM terminology and concepts to provide some context for understanding how ClearCase is organized and functions.

1.1.1 Software Configuration Management

According to the IEEE:

> *Configuration management (CM) is a discipline that oversees the entire life cycle of a software product or family of related products. Specifically, CM requires identification of the components to be controlled (configuration items) and the structure of the product, control over changes to the items (including documentation), accurate and complete record keeping, and a mechanism to audit or verify any actions.*
>
> *Software configuration management is a discipline whose goal is to control changes to large software system families, through the functions of component identification, change tracking, version selection and base-lining, software manufacture, and managing simultaneous updates (teamwork).*

In the diagram below, we can see that what is generally call CM could be more accurately called configuration and change management.


Configuration and Change Management
Asset Management
Change Control
Software Configuration Management
Software Asset Management
Software Change Management
Source Code Control

We tend to use the term CM to collectively refer to two distinct but related functions: asset management and change control. However, these functions are not just confined to managing software but include the management of hardware, equipment and other physical assets as well.

Within these two functions, there are specialized ways of dealing with classes of assets; for example IT equipment and physical real estate each need to be inventoried and identified in different ways. We can move a computer, replace parts, take it apart, upgrade it and change in a variety of other ways while a building normally doesn't get moved or subjected to as much change as a computer is.

The specialized ways we have for performing these two functions for software is a subfield of CM called software configuration management, often abbreviated as SCM. Software assets need to be managed differently because of their virtual nature; for example we can make copies of software which is something we can't do with computers of buildings.

However, when we talk about SCM, were are not just talking about source code; instead we think in terms of what we could be called the release artifacts, or the finished product which Ivar Jacobson defines this was:

The finished product consists of a body of source code embodied in components that can be compiled and executed, the manuals and the associated deliverables. It includes the requirements, use cases, non-functional specifications and test cases. It includes the architecture and the visual models -- artifacts modeled in UML. It is all the things that enable the stakeholders -- customers, users, analysts, designers, im-

plementers. testers and management -- to specify, design, implement, test and use a system. Moreover, it is these things that enable the stakeholders to use and modify the system from generation to generation.

The term software used in the SCM sense refers to a broad category of intangible assets which can include documentation, manuals, development environments and programming tools, operating systems, project documents, testing plans and the like.

Out of all of these software assets, the ones that are usually the most interest to us – the reason we hire all those developers – are those that are related to the source code under development, which is really the whole purpose for having all those other assets. Managing this specialized asset is referred to as Source Code Control and is what we will be spending most of this course on.

1.1.2 Software Asset Management

There are number of different activities that are part of software asset management. Some of these are::

1 **Asset Identification**: This involves cataloguing and identifying software assets, their properties, location and where and how they are used. This is more complex for software than for physical assets since, unlike a desk, a single source file may be used in multiple projects or may have multiple copies.

2 **Asset Control:** There are the policies for how a particular asset may be used or accessed. For example, in a configuration management system we might identify certain files as CLASSIFIED which would require them to be encrypted and accessible only to certain individuals. We might also declare a particular file "deprecated" or "frozen" so that it won't be modified. Within many projects, the issues of who "owns" or controls a software asset is something that needs to be managed carefully.

3 **Build and Release Management:** Configuration management plans usually have to support a schedule of builds and releases. Generally we don't want to destroy version 1.0 when version 1.1 is released, nor do we want developers making code changes on files while they are in test. Similarly we may want to maintain several variants of the same product that have a certain amount of overlap, but also have significant differences – like a Windows version and a UNIX version of the same product.

4 **Artifact Managemen**t. A release is not just an executable but is a collection of artifacts that collectively make up the product as we saw in the Ivar Jacobson quote. Artifact management is about ensuring that all of the artifacts "sync up" from version to version and release to release. modify the system from generation to generation.

1.1.3 Change Management

As the last line in point 4 above underscores the fact that no matter how well we identify our assets, things will change, and controlling that change so that it doesn't create havoc among our assets is the responsibility of the second major function of SCM: change management. This normally covers areas like:

1. **Change Request Management:** A change request management process is a way of evaluating requested changes to the assets to determine whether or not the change should be done, and if so, then how that request would affect the assets under configuration management. The related tool IBM Rational ClearQuest is used to automate this process.
2. **Change Control**: Usually when a change is to be made, we want to confine the scope of changes so that we don't have a “ripple effect” and unintended consequences of the a change in one place creating effects that show up in other unexpected places. Related to that is the idea that we don't want multiple changes to start interacting with each other in negative ways.
3. **Change Tracking:** If we encounter problems as a result of changes, we often want to be able to go back and look at what changes were made, when they were made, by who and why they were made.
4. **Change Rollbacks:** Even if we make changes in a controlled and disciplined manner the need may arise to “undo” the changes to revert back to a previous point in our development.
5. **Asset Synchronization:** A code change implies a series of collateral changes that have to be made as well, for example to test cases and documentation.
6. **Legacy Release Management.** While we may be on release 2.0, we may still be supporting release 1.2. Some bug fixes in release 1.2 may be incorporated into release 2.0, but generally we don't want new code in release 2.0 to be contaminating our maintenance release 1.2.1

One of the most important points of this module is that ClearCase does not “do” these functions, but rather it is a tool for automating how you do these functions.

1.2 The Library Metaphor

Before we look a source code, we are going to see if we can learn something from another group of CM experts – librarians. As a profession, they have been around for a lot longer than programmers, and they have had hundreds of years of experience in organizing things and people checking things in and out.

The reason for doing this is to establish a metaphor, the software project as a library. Of course there are significant differences, but it does help us by giving us suggestions as to what we should be doing with our SCM processes.

1.2.1 Library Asset Identification and Location

Most of us have our own libraries at home, but very few of us ever feel the need to use the Library of Congress or Dewey Decimal classification system to identify our books, nor do many of us actually set up card catalogues for our books.

After all, either there are few enough books that we can remember where each of them is, or there are only so many places a particular book could be so looking for it is trivial.

We often see the same thing happening with programmers. A project is small and they are the only person working on it, so the few files are easy to find and it's relatively simple to find what we are looking for.

However, after a while the number of books starts to accumulate to the point where (as seen on the next below) the “I'll just remember where everything is” system starts to break down for a couple of reasons.

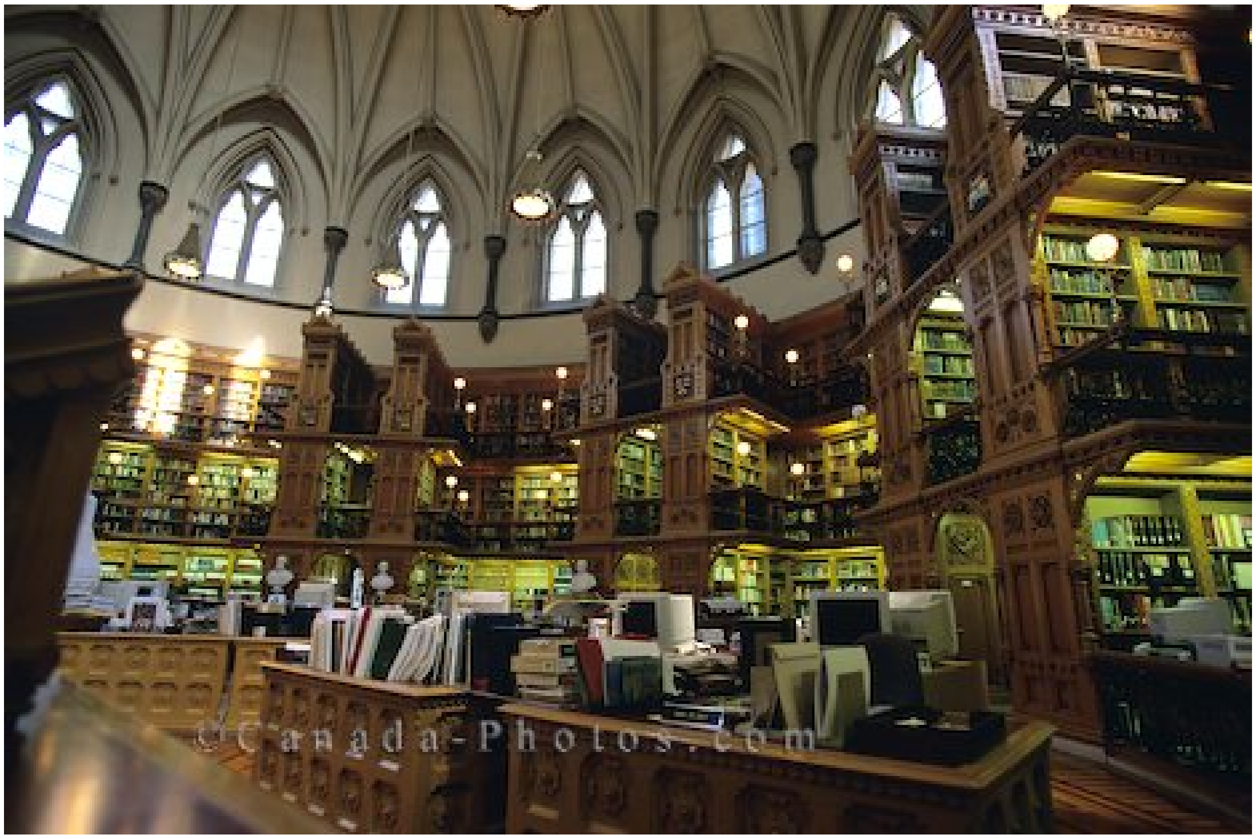
1 There are too many books to remember where they are. Now we start to waste time looking for things which affects our productivity.

2 We might start loaning books to other people but forget so we wind up looking for things that aren't even there.

3 We are not sure where we've put the books because we ran out of room in our original library and are using multiple locations to keep them.

Of course this has an analog in programming too; we spend all of our time looking for code, trying to figure out if we are working with the right copy and whether we are in fact even looking in the right place. And of course once we get more than one person using the same code, the problems escalate.

How does the library solve this problem? By using and asset identification system which then can scale almost without limit. The picture below is of the Library of the Parliament of Canada in Ottawa.



Librarians have learned three very important things about running libraries.

1 Libraries need a way of uniquely identifying what books they have which means they need to use an asset identification system. Libraries use a couple of different systems that are familiar to anyone who has used a library; the LC or Library of Congress classification system and the older Dewey decimal system.
2 In addition to knowing what books the library has in its collections, they also have to have a way of finding those books. Libraries use a catalogue and locator system to enable to them to locate a specific book in the stacks. Notice that the locator system is built on top of the identification system.
3 Changes mess up the whole system when books are added, taken out, put back in the wrong place; in fact using the library without change controls renders the whole asset management system useless in a very short time.

The reason we need a locator system is that asset identification system is not enough: even if we assigned every book a unique id but then just placed them randomly on the shelves, we would have chaos. The locator system ensures that we can actually find our assets.

Finding a book in the library is a multi-step process.

1. We start with some identifying information about the book we want to find; the title or author perhaps.
2. We use the card catalogue to find out if the book is actually in the library rather than wondering randomly about in the hopes we might come across it. We are confirming that the book is an asset under the control of the library. The result of looking through the catalogue is finding the id of the book in our system – something like QC457A1481982 for the book 1981 Commercial Composite Infrared Index, grating numerical index
3. The number QC457A1481982 is a call (or Cutter) number that points us to the actual location of the book in the stacks. Armed with this and a map of the library, we can find the book.

As with a library, we need the same with code: an asset identification system and location algorithm so that we can do the same sort of thing with our code, test cases and documentation. Once we have such a system, we can automate it with ClearCase in the same way that we would automate the library system by moving from a manual card catalogue to a database.

1.2.2 Library Change Management

Of course nothing ever stays the same. Users of the library remove books from the shelves and then either check them out, possibly re-shelve them in the wrong place or just leave them lying around. In other words, after enough use, the location algorithm breaks down and the value of the asset identification system is lost. In addition there are new books being added, old books being removed, perhaps some just temporarily for repair, multiple copies of the same book, and different versions of the same book as well.

All of this change has the effect of corrupting the library's configuration management system. In response, the library develops a set of policies and procedures to manage changes and their impact on the library.

1.2.2.1 Policies

Generically policies describe what can and cannot be done and who can do what under what circumstances. For example the library might establish the following policies:

1. Only members in good standing can check out books.
2. No one may have more than six books checked out at one time.
3. Reference materials may not be checked out.

4 Materials my be placed on reserve and then checked out only the member that requested the hold.

5 Books may only be borrowed for two weeks.

6 Members must return books to the library staff and not re-shelve them themselves.

The policies delimit the sorts of changes that can occur and how those changes are to be managed in terms of their impact on the asset identification system.

1.2.2.2 Procedures

Having policies is often not enough because while they tell us what we are supposed to do, they often do not tell us how to do it. For example the library may have a policy about placing items on reserve or a policy about checking books out, but the users and staff may not know how to follow the policy or they may have different ideas about how to follow the policy. The end result is chaos.

In addition to policies, we often need to define procedures which can be thought of step by step instructions on how to follow a particular policy or set of policies. “We do it this way because then we can make sure the borrower is authorized to check books out.”

For example, at the library, we may have a “borrow” process that tells us how to check out a book.

1 A user looks up the call number of the book in the book catalogue.

2 The user then gets the book from the shelf and takes it to the circulation desk.

3 At the circulation desk, the user presents their membership card and the book to the librarian at the desk.

4 The librarian checks the user's card to ensure that they are a member in good standing with less than six books checked out and no overdue items or fines outstanding.

5 The librarian checks to see that each item is not on reserve or has some other reason to prevent it from being checked out.

6 If the user is permitted to check out the item, the information is updated in the current circulation list and the user's membership file.

7 If the user is not permitted to check out the item, then the librarian will take the item and place it in the “to be re-shelved” bin.

Notice that the process is generic in the above description and could also be described from the point of view of what the library patron does or what the librarian does.

Also notice that the process is not compulsory at every step. For example, if we already know the call number of the book, we can skip the first step, but we don't have the option of skipping step 4 where we are vetted by the librarian.

Very often we find policies expressed as following a procedure. “It is our policy that you do things according to this exact procedure.”

1.2.3 What Does the Library Teach Us?

The library example has an important lesson for us which we can summarize in the following points.

1. We need to define our asset identification and location systems before we start populating our repositories with items. In ClearCase, this means fully defining our source trees, directories and their layout within projects before creating them in version control. We do not do version control on the fly.
2. We need a set of policies to define what we can and cannot do with the items under version control in order to keep the integrity of the configuration management assets.
3. We need polices to manage changes to the assets.

We may need to define procedures to ensure that the policies are followed correctly.

1.2.4 Applying the Library Metaphor to SCM

We spoke generally about configuration management so far, but before we get into the product itself, we will look at how these relate to specific questions and issues in software development. The basic challenges of SCM, and more specifically source code control are:

1. Identifying the basic units that make up our system under development. We want to identify the individual elements that we are working with. For example they could be files, documents, manuals, chapters, test cases or test suites and so on.
2. Establishing a way to refer uniquely and specifically to each of those units.
3. Identifying who owns or is responsible for each unit, and who is allowed to use or modify each unit and in what way.
4. How the various unit are combined to produce carious components like modules sub-systems and systems for example.
5. How making changes should be planned to avoid unintended effects in other parts of the system.
6. How to execute the changes to avoid corrupting the SCM assets.

And these, of course, arise from the basic development and support processes. In this course, by the support function we mean the process of making changes to software that is released and in production.

Our SCM plan and system should support the following basic activities:

1 Producing code which involves designing, writing, reviewing, unit testing and debugging.

2 Creating builds into components for integration or build testing.

3 Support the allocation or work tasks and project development plans.

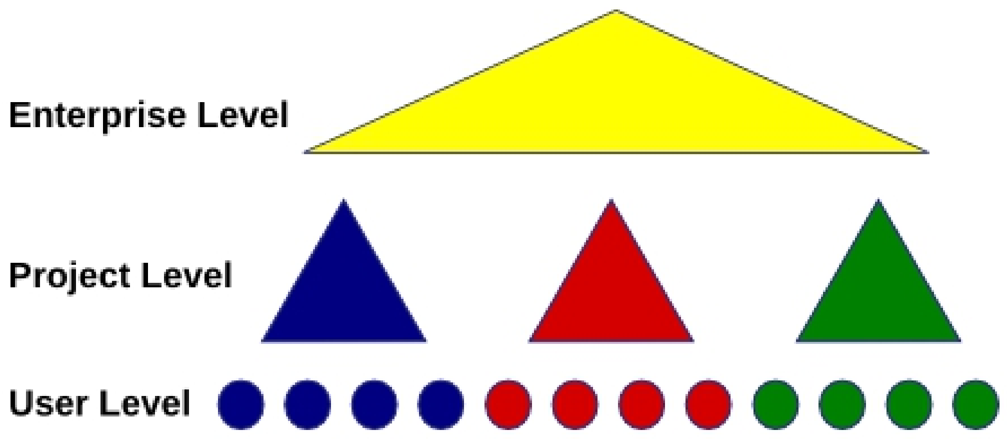
4 Manage the artifacts that are associated with the project including specifications, design documents, test cases, test plans and requirements documents.

Point three above is important because this is where SCM overlaps project management and software development process definition. For example, we should not be testing code that is about to undergo a modification or writing code before the module design is stable.

One of the main concerns we have is that the SCM policies and procedures support the overall project management and development processes, which is something that ClearCase was designed to do.

1.3 Different Levels of Interacting with ClearCase

ClearCase,is designed to be used within an organization at three different levels.



Each of the levels represents a perspective or view of ClearCase within an organization.

1 **User Level.** This is the level that most people in the organization will be interacting with ClearCase. They will be checking files in and out and generally not actually thinking much about ClearCase except in an indirect manner as they go about their daily tasks. This is the level of interaction that this course is focused on.

2 **Project Level.** At the project level, the specific policies and procedures are defined and implemented that allow those at the user level to interact with ClearCase in a smooth and efficient manner. The types of tasks typically done at this level are developing VOB layouts, planning branching strategies and merge schedules, release planning, and integrating ClearCase policies with project and work policies.

3 **Enterprise Level.** The enterprise level is the back end support level where the main concerns are issues like VOB storage, ClearCase network support, server configuration and licensing – the issues that are necessary to maintain a ClearCase infrastructure throughout the organization.

Very often we find that when there are problems in using ClearCase in an organization, it is often because one of these levels is missing or deficient and the responsibilities of that level are performed in an ad hoc manner by other levels.

For example, in many organizations the project level is often missing completely resulting in the users trying to perform many of the functions that should be done at the project level for themselves – usually in a haphazard and unplanned manner. The usual result is chaos and increasing frustration on the part of the users with all of the ClearCase “stuff” they have to do, which usually culminates is them avoiding using ClearCase so they can concentrate on their work.

1.3.1 The Enterprise Level Tasks

At the enterprise level, the main concerns are to ensure that ClearCase is available, that it's running correctly, that the VOB and View storage and registries are manged properly with respect to storage, backup and security, and that ClearCase network is running as it should.

While we do not normally concern ourselves with these tasks at the user level, we do need to take into account what is being done at the enterprise so that we can do can things like find our VOBs and views and also be able to have a basic feel for what to do when things do not seem to be going as they should. In other words, to know when to call ClearCase support.

1.3.2 Project Level Tasks

This course is not about working at the project level. However since users always have to work within the structure set up at the project level, we will be looking at how we interact with what is done at the project level, which gives a sense of explaining why we do things a certain way as ClearCase users.

What the project level actually is concerned with is making ClearCase usable for a particular set of users by creating an infrastructure that allows the users to get on with their work and not have to think about ClearCase. The ideal situation is that the project planning adapts ClearCase into their work flows so that it becomes ubiquitous and unobtrusive.

The best sign that we have done the project level ClearCase management effectively is when users never think about using ClearCase because it fits seamlessly into their project tasks.

The actual tasks that are done at the project level are going to vary from organization to organization depending on a number of factors such as IT infrastructure organization, process methodologies and other variables, however there is going to be a commonly recurring set of tasks that define the project level in generic terms.

Configuration Asset Definition. One of the factors that makes for good SCM is that the initial set of assets under version control is well defined, adequate and well structured. There has to be a master view of the items under version control in the same way that in a database, there needs to be a data model that is independent of any particular user's view of the data. However just as important is that the SCM plan allows the assets to evolve and change as the project moves through various stages so that the SCM does not “hold back” project progress. Failing to have a robust and flexible set of assets usually results in a brittle CM repository that starts to collapse in on itself as the various users attempt to “work around” the structure.

Policy and Procedure Establishment. Policies and procedures are the agreements and rules by which the assets under version control are accessed and changed. Polices are the rules that specify what may or may not be done, and procedures are descriptions of how to interact with the repository in a way that respects the rules. Policies and procedures are defined at the project level although they may be derived from organizational policies and procedures.

Life Cycle Process Support. ClearCase out of the box is process agnostic, which means that it works with every process flavor from traditional highly structured waterfall processes to iterative process like RUP and even to highly adaptive processes like Agile methodologies. However, in order to support the specific type of process that is going to be used by team, the appropriate ClearCase artifact and strategies need to be in place. For example. ClearCase allows branching and merging, but the strategy for branching and merging is something that needs to be defined depending on the particular project process. Agile projects will use a radically different branching and merging plan than those developing a set of releases for an application in production in a large organization.

It's not just the development phase of a life cycle that has to be supported by the ClearCase polices and procedures, but also the kinds of tasks that take place during the support and retirement phases of an application's life cycle.

The policies and procedures of life cycle support are not necessarily those of the developers, which means that we may need to also have polices in place to support the requirements of the organization on issues like auditing and security.

Infrastructure and Performance Support. ClearCase is an older product and can have performance issues, especially when running in a network environment with layers of security and network applications that it has to interact with. Very often, we can get significant improvements in the performance of ClearCase by establishing certain kinds of procedures to avoid the kind of network congestion that interferes with the user experience of ClearCase.

Release Management Support. Release management includes the branching and emerging models and strategies that are part of a project, but it also includes the planning, testing and supporting of releases and versions. Since release management is a cross disciplinary activity involving developers, testers and others, it is most effectively supported in ClearCase when it is part of the project level activities.

1.3.3 User Level Tasks

In a well designed ClearCase environment there are no user level tasks except to follow the defined policies and procedures. One of the deadly mistakes we want to avoid is having users performing project level tasks in an ad hoc manner, something that never ends well.

So how do you know if you are using ClearCase as it was intended to be used? The primary measure is that you don't really notice it. Instead you are able to concentrate on your work and not think about ClearCase of configuration management except for maybe a few points here and there. If you start to spend a lot of time fiddling with ClearCase and elements under version control, or it seems that ClearCase is not worth the effort, then is suggests something is wrong at the project or administrative levels.

1.3.3.1 Brad Appleton's Checklist.

The following are attributed to Brad Appleton, one of the SCM gurus out there. SCM is not being done well when...

1 Bugs that have been fixed suddenly reappear.

2 Previous releases of a product cannot be built.

3 Previous releases of a product cannot be found.

4 Files cannot be found.

5 Multiple copies of the same file are found with "little" differences.

6 Files or code mysteriously "mutates" on its own.

7 Similar code exists in multiple places in the same and/or different projects.

8 Programmers make changes that are "overwritten" by other programmers.

9 Developers have deja vu a lot – they code they are writing seems to be code they have written before, but there is no trace of it.

10 Documentation is, at best, a theoretical concept.

11 "I know I wrote it, but I don't know where I put it."

12 "This used to work, but now it points to code that isn't there anymore."

13 New fixes made it worse, and there's no "undo" button.

14 Dropped a document with no page numbers, or dropped two documents, no titles on pages, which was which?

15 Bug reported by customer, but don't know what version they have, don't know what fix to give them.

ClearCase Comprehensive

exergis

Module Two

ClearCase Architecture and Concepts

I never make stupid mistakes.
Only very, very clever ones.
John Peel

Now that we use ClearCase to organize our code,
we nave come to the brutal realization that our code is not worth organizing.
Anonymous

2.1 2.1 The Wonderful World of ClearCase

Many users find ClearCase confusing to use because it behaved differently than most other SCM products – especially with the dual VOB-View architecture. This module is a high level look at ClearCase to introduce the basic concepts and terminology that, once understood, make using ClearCase a lot more intuitive and easier.

2.1.1 The ClearCase Product

ClearCase for Unix was originally first released in 1992 by Atria Software. The product was later released on Windows. Atria Software, after several mergers, was acquired by Rational Software as part of its “Rational Solution.” which was an attempt to automate the full development life cycle within the Rational Unified Process. Rational was purchased by IBM in 2003.

ClearCase underwent a major revision with the release of Version 7.1 in December 2008 including enhancements for CCRC clients. IBM released Version 8.0 in November 2011.

2.1.1.1 Integration

One of the on-going goals of IBM-Rational has been to integrate ClearCase with other Rational products such as ClearQuest, RequisitePro, Rational Rose and the various testing tools. Because of the SCM function of ClearCase, it integrates with a variety of developer platforms like Eclipse, NetBeans and Visual Studio.

Because of the difficulty in integrating products like ClearCase, ClearQuest and RequisitePro, primarily because they all originated with different vendors, IBM-Rational introduced UCM or Unified Change Management, as a methodology that provides unified approach to managing project assets that works across products.

UCM does this by introducing a set of high level artifacts like “project” and “baseline” which are then converted out of sight of the user into the product specific items, like views and tags in ClearCase. As a result, ClearCase comes in two versions, the original or “base” ClearCase and the UCM version of ClearCase.

In this course, we will only be dealing with base ClearCase.

2.1.1.2 Issues with ClearCase

ClearCase is mature product, which is a nice way of saying that it's now 20 years old. On one hand the fact that ClearCase is still a leading choice of SCM software is a testament to the robustness of the design and how well ClearCase has been engineered.

On the other hand, the age of the product often shows up in issues mostly dealing with the speed of the product, especially with dynamic views, and the inconsistencies between the Unix and Windows versions (such as the difficulty of using Windows Drive letters). Most of the practical problems deal with issues that center around using ClearCase in highly secure environments where calls to the VOB may wind up propagating across a number of applications and hosts.

However, as we shall see in this course, all of these issues can be worked around by planning out the SCM policies and processes in advance to avoid the specific problem areas.

There are aspects of the product that are anachronistic. Because it predates the world wide web and it's associated technologies, it does use techniques and constructs that are similar to what we use today, but ClearCase names them differently or uses them in an unfamiliar way. For the average user, this can create some confusion since it seems that ClearCase does things differently than everyone else.

For example, ClearCase uses constructs called VOB tags which are used to locate VOBs during an action called “mounting a VOB.” However, a VOB tag is what we would call a URL today, and “mounting a VOB” would be called a DNS look-up today. However since the concepts of URLs and DNS lookups didn't really exist in 1992, ClearCase used it's own terminology and models to implement those same ideas.

As we look in this course at how you as a user will be interacting with ClearCase, the design concepts of ClearCase will be explained to show how those interactions are intended to be intuitive and easy to master.

2.1.2 ClearCase Design – User Perspective

The main design goal of ClearCase is that the product should be as invisible as possible to the end user. As long as there are robust work processes in place, then ClearCase should be able to essentially work everywhere in the background without the user ever really having to aware of the product. This means that ClearCase:

1. Makes files under version control look exactly like regular files in the native file system. This means there is no learning curve for working with files in ClearCase, users continue to work with files as they always have.
2. Ensures that make utilities and other tools like awk and grep, and in particular user scripts, run without modification when applied to files under version control. There is no need to “port” the development environments to accommodate files under version control.
3. Automates ClearCase actions by using “triggers” so that it can run out of sight by integrating with development and other tools. For example, opening a file in a programmer IDE is an event that “triggers” ClearCase to check-out that file.
4. Has a Unix like command line set of commands that can be integrated into standard shell scripts for automation.
5. Models its commands after the Unix command set so that anyone who is familiar with Unix commands is able to find the ClearCase command set intuitive.

The last goal may seem a bit odd in 2012, but again we have to remember that in 1992 ClearCase was used by organizations where either Unix or mainframes were the main development platforms for large projects.

2.1.3 ClearCase Design – System Perspective

Along with the design goal to make ClearCase as user friendly as possible, there was a second design goal which was to make ClearCase as efficient as possible and easy to administer,

This means that ClearCase:

1. Decouples how items under version control are viewed by users from how they are actually stored. The actual storage mechanisms used for items are optimized for performance.
2. Decouples the users perspective of where the VOB are from their actual physical location so that the administrators can restructure their infrastructure without the users even being aware of the changes.
3. Allows administrators to implement security, backup and other administrative policies without requiring the user to be aware of them.

In order to accomplish these two main design goals, ClearCase uses a unique VOB-View architecture which is our next topic.

2.2 Architecture

First we should define two terms that we have been using without explaining what they are.

VOB: This stands for *versioned object base.* A VOB is a repository (i.e. a physical location) that ClearCase uses to store elements under version control and all the meta-data associated with those elements.

View: A view is a process that runs on the view server in conjunction with a client process on the user's machine that:

1. Intercepts file system commands and ClearCase commands and then converts those into the appropriate command to the VOB to access specific items under version control.
2. Converts the results of the VOB requests into something that looks the local file system.
3. If the user is not in VOB, then the view process passes on any file system commands to the process that normally processes file system commands.

By a file system command here we mean things like `“dir”` and `“type”` in Windows or `“ls”` and `“cat”` in Unix. We also mean commands to the file system that come from graphical interfaces like Windows Explore or a similar tool like Dolphin or Nautilus in Unix.

In Unix the view makes the VOB look like an ordinary directory in the file system. In Windows, a view is presented as a network drive (snapshot views actually work differently but we will defer a discussion of that until the module on snapshot views).

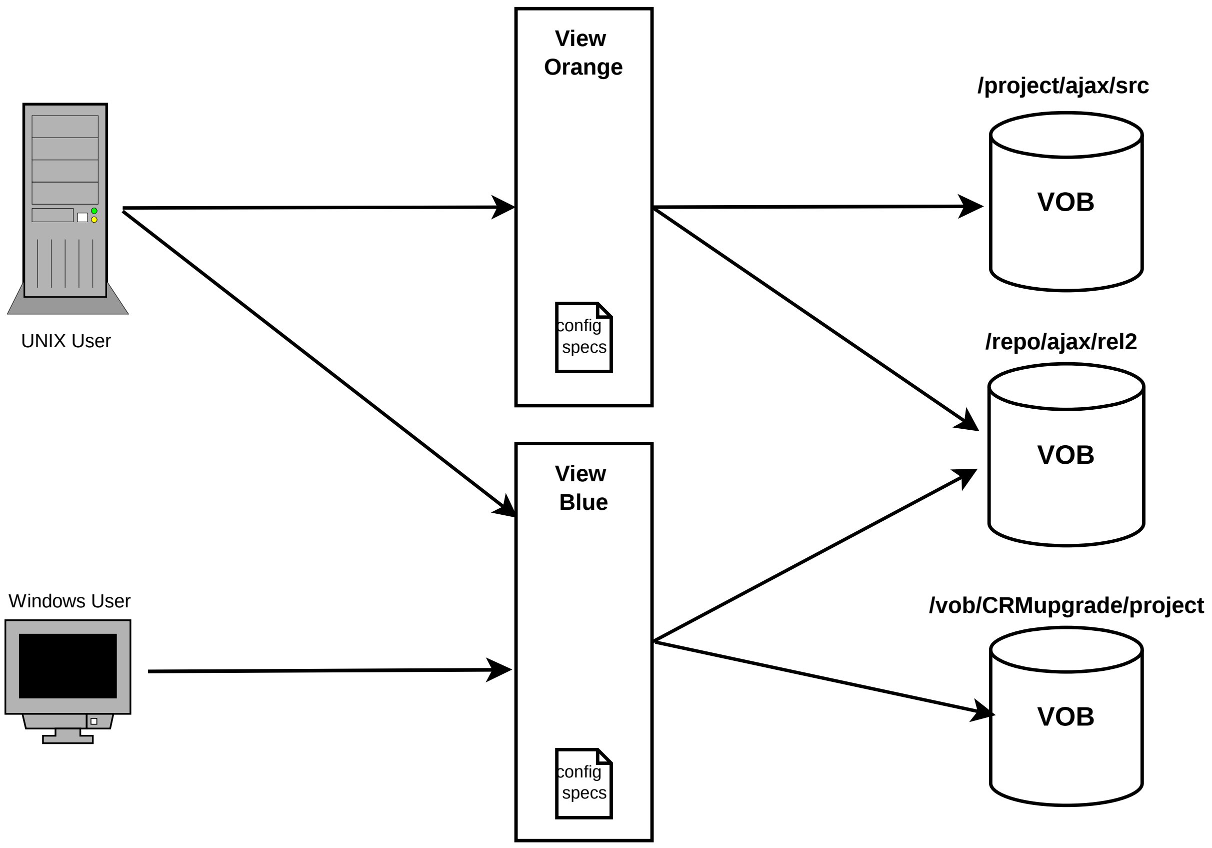
Even though views are processes, each view has associated with it a storage location that contains the configuration specs (the rules the view uses to figure out what to ask the VOB for) and the temporary storage it needs to run across sessions.

In the diagram on the next page there are:

1. Two users running ClearCase clients on their local machines. One user in on a UNIX system and one on a windows system
2. Two views “Orange” and “Blue” each of which has their own set of rules for selecting versions from VOBS.
3. Three VOBs which contain the actual files under version control.

To the UNIX user, the VOB `/project/ajax/src` is invisible until they activate a view using the `setview` command. If the Blue view is selected, then the user sees the VOBs as part of the UNIX file system, just as if they had mounted a removable file system.

To the Windows user, the Views appear as network drives and the VOBs appear as top level folders in that those drives. The names of the VOB are presented to the Windows user with the appropriate pathnames like `\project\ajax\src`.



The following sequence of events occurs.

1 The UNIX user executes the `ls` command using the Blue view and while being located to the `/project/ajax/src` VOB.

2 The ClearCase client process intercepts the `ls` command and passes the request to the Blue view process.

3 The Blue view process looks at the VOBs this user is connected to and consults its config specs for each visible element in the VOB (by visible we mean elements in the VOB that would be listed by the `ls` command if it were a file system) to see what version of the element the view should show to the user.

4 The Blue view makes the request to the appropriate VOB for a specific version of that element.

5 The VOB sees if it has a cached version of that element, and if it does it returns that to the view. If it doesn't then it assembles the requested version from its database and returns that.

6 The collected output from the VOB is then formatted to look like a UNIX file system and presented to the UNIX user.

2.3 Server Organization

To make sense of the relationship between VOBs and views, it is helpful to take a look at how ClearCase is organized so that we can see how the scenario described in the last section is actually supported.

ClearCase is organized as a collection of hosts, each of which performs a specific infrastructure task for ClearCase. In practical terms, a ClearCase host is really a role that is performed so that a single physical machine may play the role of all the hosts mentioned or one role may require multiple physical machines.

Some ClearCase hosts that the CM role will interact with are:

1. **License server hosts**. A ClearCase license server host is responsible for authorizing and restricting ClearCase use according to the ClearCase license. Each host running ClearCase must periodically authenticate itself to a specific license server host.
2. **Registry server host.** One host in the network serves as the ClearCase registry server host. This host is responsible for allowing ClearCase users to be able to get path information as to where VOBs and views are physically located.
3. **Server hosts.** Some hosts function as the physical data repositories for VOB storage and view storage.

There other more specialized hosts but they are not really of interest to us until later in this course.

2.3.1 Registry Servers

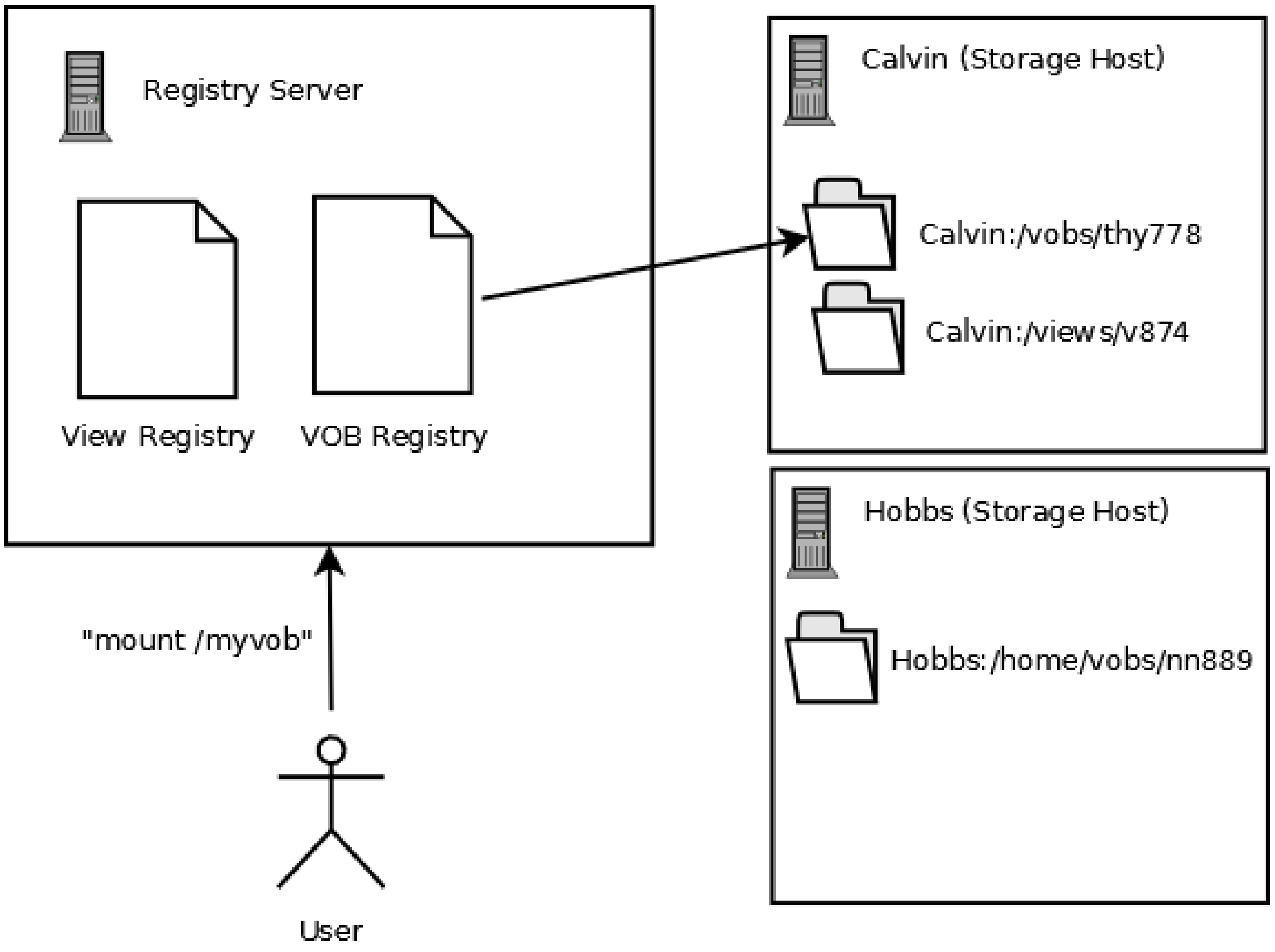
For our purposes, we can think of the registry server as containing two tables that associate VOB names and view names to actual storage locations.

Consider the set up in the diagram on the next page where we have a registry server and two storage servers named Calvin and Hobbs.

There are two VOBs physically located at **Calvin:/vobs/thy778** and **Hobbs:/home/vobs/nn889**. However we don't want the user to refer two these physical locations so we assign two “tags” or what we would call URLs today that the registry server will use as aliases for these vobs.

In the registry server is the following table

VOB tag	VOB location
/myvob	Calvin:/vobs/thy778
/projects/alpha	Hobbs:/home/vobs/nn889



And this one

View Tag	View Location
Alpha_developer	Calvin:/views/v874

As we will see later, if we are in UNIX, we would issue a couple of commands like:

```
% cleartool mount /myvob
% cleartool setview Alpha_developer
```

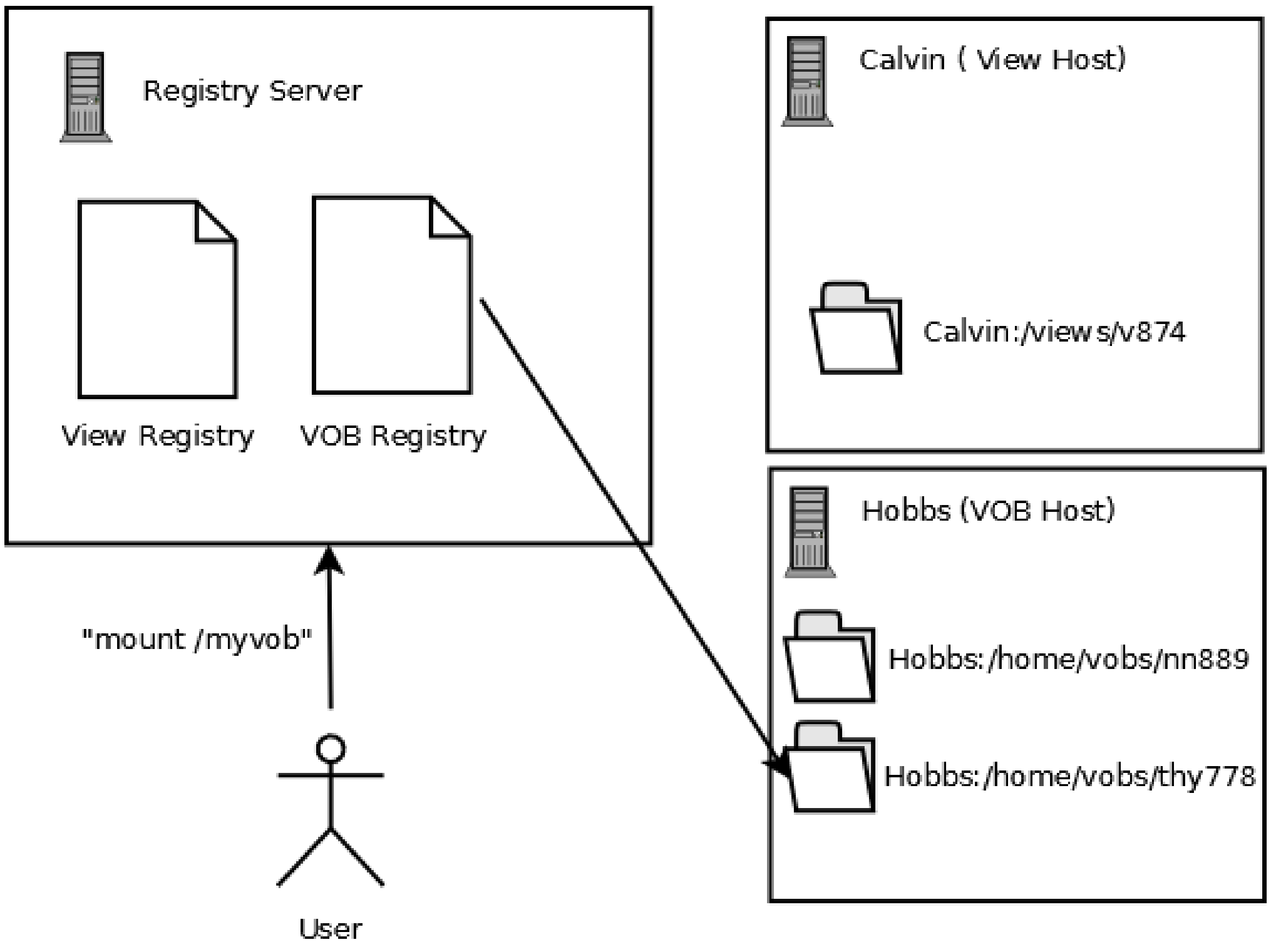
which would have the following effect.

1 It would as the registry server to look up the actual physical location of the VOB represented by the VOB tag **/myvob**. In this case it resolves to the physical location **Clavin:/vobs/thy778**. At this point we are now "located" or "connected" to the actual VOB repository.

2 The setview command causes ClearCase to start a view process. When it starts it looks up the view storage by checking the view tag registry table. As you can see the view tag Alpha_developer resolves to **Calvin:/views/v874**. At this point the view process reads the configuration files in the directory so that it knows how to present what is in the VOB to the user.

Of course these commands would not have to be explicitly issued by the user but could part of the user's login script and run automatically whenever she logged on.

So why both going to all this trouble? Lets suppose that the servers have reorganized so that Calvin will now be a dedicated view storage server and Hobbs will be a dedicated VOB storage server.



The VOB **Calvin:/vobs/thy778** is now moved to **Hobbs:/home/vobs/thy778**. In addition the VOB registry server is updated so that it's VOB table looks like this.

VOB tag	VOB location
/myvob	Hobbs:/home/vobs/thy778
/projects/alpha	Hobbs:/home/vobs/nn889

Now the same set of commands

```
% cleartool mount /myvob
% cleartool setview Alpha_developer
```

Would still locate us to the correct storage location.

2.4 Using ClearTool

ClearCase can be accessed a number of ways. The most basic way that we access ClearCase is through the command line tool. The command line tool has the advantage that it is the same across platforms and that it allows us to access the functionality of ClearCase without the need for any particular client environment or application.

There are also two other ways that we can access ClearCase: through graphical tools and through plug-ins to other tools. For example, developers who use the Eclipse tool generally want to stay within the IDE when working and not have to constantly switch tools: otherwise their natural tendency is to avoid using ClearCase because it "gets in the way." To be able to access the VOB and ClearCase functionality through an add-in to Eclipse is obviously a definite advantage.

The Windows client for ClearCase uses a graphical interface called the ClearCase explorer, modeled after the Windows explorer. However, it is recommended that even if you are going to be using the Windows client, you still should be comfortable using the command line in the Windwos environment.

2.4.1 Connecting to ClearCase

At the command line, we issue commands ClearCase through the ***cleartool*** utility. Once our environment is properly set up, we can see if we can use ClearCase by issuing a couple of test commands. Even though the cleartool utility is available in both the Windows and Unix environments. For simplicity, we will illustrate the cleartool examples using a Unix style shell.

A simple test to see if your ClearCase client is ready to us is to see if it can access a license server.

```
% clearlicense
License server on host "<<whatever>>".
Running since Thursday 08/01/11 21:15:54.
LICENSES:
     Max-Users  Expires       Password [status]
        20         none       aaa.bbb.ccc [Valid]
 Maximum active users allowed: 20
```

Invoking cleartool is done at the command line by typing cleartool at the command line which starts up the cleartool shell. We can tell we are in the cleartool shell since the prompt changes. We can now enter a number of cleartool commands. When we are finished we can exit the cleartool shell by typing quit.

```
% cleartool
 cleartool> (cleartool commands)
     << output from clear tool commands >>
 cleartool> quit
%
```

We can also issue a single cleartool command by entering the command right after cleartool.

```
%
% cleartool (cleartool command)
     << output from clear tool commands
%
```

In Windows, the word cleartool has to be fully typed out, however in Unix, we can abbreviate it to **ct**. We could have typed the above in Unix as

```
%
% ct (cleartool command)
     << output from clear tool commands
%
```

2.4.2 Cleartool Help Tricks

Many of the cleartool commands follow a specific pattern which allows us to often make a good guess at what a command should be or should do.

1 Commands that start with “ls” list things. So **lsvob** lists VOBs, **lsview** lists views, **lscheckout** lists elements currently checked out.

2 Commands that start with “mk” make things. So **mkvob** creates a VOB, **mkview** creates a view.

3 Commands that start with “rm” remove things. So **rmvob** deletes a VOB and **fmview** deletes a view.

ClearCase also has a useful on-line help facility. Typing the word “man” followed by an optional cleartool command will produce either a Unix style man page or pop up an HTML page with a full description of that command. For example, to learn about the **lsvob** command we would type.

```
% cleartool man lsvob
```

2.5 Working with a VOB

As a user you will not normally be creating and removing VOBs but in this class, depending on how things have been set up, you may have to create a VOB to use. Also going through this exercise solidifies a lot of the things we have talked about up to this point.

2.5.1 Creating a VOB

To create a VOB, we need to define a storage location and a tag for the VOB. To do this we use the mkvob command in cleartool. For the purposes of this example, we will assume that we will be using a single physical host “Garden” for the registry server and the storage servers.

2.5.1.1 Unix

Assume we are user **leslie** on host **locke** using Unix and decided to use the directory /home/uservobs as “VOB storage server.” We can pretty much create a VOB wherever we can make a sub-directory.

```
% ct mkvob -tag /myvob -stgloc /home/usrvobs/myvob.vbs
      Created versioned object base.
      Host-local path: locke:/home/uservobs/myvob.vbs
      Global path: /net/locke/home/uservobs/myvob.vbs
      VOB ownership:
      owner leslie
      group dev
      Additional groups:
      group usr
      group adm
```

The -tag option specifies what the tag for the VOB should be, notice that it should look like a Unix path name. The -stgloc option specifies where the VOB is physically created, in this case in the sub-directory myvob.vbs. The suffix .vbs is used to help us remember that this directory contains a ClearCase VOB.

A more common approach is to define a storage location first, then have ClearCase automatically allocate the storage. This is often used to restrict the creation of VOBS to predetermined locations on specific hosts. In this case we would use two commands. The mkstgloc command creates the storage area and then the mkvob command uses the -auto feature to get storage in this default location.

```
% ct mkstgloc –vob vobserver1 /home/uservobs
      Created and advertised Server Storage Location.
      Host-local path: locke:/home/uservobs
```

```
             Global path: /net/locke/home/uservobs
```

The -vob option tells ClearCase that is a storage location for VOBs and the name vob-server1 is the name of this storage location. We can obviously define multiple storage locations

Now we use the mkvob command to create a VOB in this default location.

```
% ct mkvob -tag /myvob -stgloc vobserver1
      Created versioned object base.
      Host-local path: locke:/home/uservobs/myvob.vbs
      Global path: /net/locke/home/uservobs/myvob.vbs
```

2.5.1.2 Windows

The commands for creating a VOB in Windows are exactly the same as for Unix except that there are some syntactic differences is specifying the pathnames and not being allowed to use more than a single level of name.

```
C:\> cleartool mkvob -tag \myvob -stgloc c:\vobs\myvob.vbs
      Created versioned object base.
      Host: locke
      Local path: C:\vobs\myvob.vbs
      Global path: \\locke\vobs\myvob.vbs
      VOB ownership:
      owner leslie
      group dev
```

We could also use a network share name.

```
C:\> cleartool mkvob -tag \myvob -stgloc \\locke\vobs\myvob.vbs
```

2.5.2 Listing VOBs

The **lsvob** command is used to list the contents of the VOB registry.

```
% cleartool lsvob
```

```
        * /myvob       /net/locke/home/uservobs/myvob  private
          /prg/nexus   /net/locke/home/uservobs/nexus  public
        * /expt        /net/hobbs/vobs/thy778   public
         /vobs/pvob   /net/hobbs/vobs/p_vob  private (pvob)
```

When the VOBs are listed, the command tells us a number of things. The asterisk (*) at the beginning of the lines above indicates that the VOB is mounted or actually connected to the phsyical storage location of the VOB. The two pathnames for each VOB pathname of the VOB that you would use to access and the other pathname specifies the VOB's actual location.

The example above has **/myvob a**nd **/expt** mounted for the user who executed the lsvob command.

2.5.3 Mounting and Unmounting VOBs

As we have seen, to mount a VOB means to have the registry server resolve the VOB tag to a physical storage location. Essentially to mount a VOB means to make it accessible to a user and to unmount a VOB is to make it inaccessible to a user.

```
      % cleartool lsvob
        * /myvob       /net/locke/home/uservobs/myvob  private
          /prg/nexus   /net/locke/home/uservobs/nexus  public
        * /expt        /net/hobbs/vobs/thy778   public
         /vobs/pvob   /net/hobbs/vobs/p_vob  private (pvob)
```

In the above example, the VOB /prg/nexus is not mounted. The **mount** command is used to make it accessible. Using the command "**mount -all**" mounts all public VOBs.

```
      % cleartool mount /prg/nexus
      % cleartool lsvob
        * /myvob       /net/locke/home/uservobs/myvob  private
        * /prg/nexus   /net/locke/home/uservobs/nexus  public
        * /expt        /net/hobbs/vobs/thy778   public
         /vobs/pvob   /net/hobbs/vobs/p_vob  private (pvob)
```

And we can also unmount a VOB using the **umount** command (notice that it is NOT unmount but rather umount).

```
      % cleartool umount /myvob
```

```
% cleartool lsvob
    /myvob       /net/locke/home/uservobs/myvob  private
  * /prg/nexus   /net/locke/home/uservobs/nexus  public
  * /expt        /net/hobbs/vobs/thy778   public
   /vobs/pvob   /net/hobbs/vobs/p_vob  private (pvob)
```

2.5.3.1 Public and Private VOBs

The VOBs themselves are neither public nor private, it is the VOB tags that are public or private. This is a way to control access to VOBs. By default VOB tags are private, but the -public option can be used in **mkvob** to create a public VOB.

```
% ct  mkvob -tag /myvob -public -pass zowie -stgloc vobserver1
```

Often a password will be required to create public VOBs so that the administrators can control the creation of public VOBs.

In Unix, all public VOBs are mounted as a group when ClearCase starts. In Windows, all public VOBs are mounted as a group when the command **“cleartool mount –all “** is executed.

The other difference in Unix is that a mount-over directory must exist for a private VOB that is usable the user executing the mount command and is the same name as the VOB tag (note that for public VOBs, this mount over directory is created automatically).

```
%  ct mkvob -tag /home/leslie/myvob -stgloc vobserver1
%  mkdir /home/leslie/myvob
%  ct mount /home/leslie/myvob
```

2.5.4 Removing VOBs

As you would expect, there is a **rmvob** command as well. However, like any rm-anything command, it should be used only with the greatest of care and forethought.

The **rmvob** command deletes a VOB storage directory that was originally allocated using **mkvob**. In addition to removing the physical VOB storage directory, **rmvob** removes all references to the VOB from the VOB registry.

Caution: The **rmvob** does not unmount the VOB and removing the VOB without unmounting it could have disastrous results for some users.

Generally, you have to be root or the owner of the VOB to execute this command.

```
% cleartool umount /myvob
% cleartool rmvob /home/usrvobs/myvob.vbs
 Remove versioned object base "/home/usrvobs/myvob.vbs"? yes
 Removed versioned object base "/home/usrvobs/myvob.vbs".
```

ClearCase Comprehensive

Module Three

Versions and Checkouts

It is not a question of how well each process works,
the question is how well they all work together.
Lloyd Dobens and Clare Crawford-Masonr

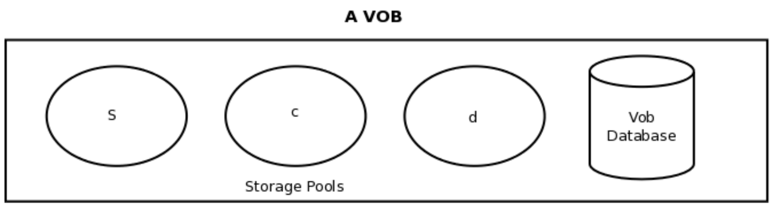
3.1 Versions and Elements

Up until now, we have been talking about VOBs without discussing what's in them. In this module we will look at the actual elements that are under version control.

First of all, it is important to understand that a VOB is not just a directory full of copies of files. In fact if you looked inside a VOB directly as a standard file system object you would see a number of strange looking files and sub-directories.

3.1.1 Inside a VOB

A VOB physically exists as a directory tree on some host somewhere in the ClearCase network. in the file system. These sub-directories that make up a VOB directory tree are:



1 **db** – Holds the VOB database

2 **s** - Holds the source storage pools. All of a file element's versions are stored in a particular source pool in this sub directory

3 **c** – Holds the cleartext storage pools. Recently accessed version of some file elements are cached here for faster retrieval.

4 **d** – Holds the derived object storage pools. These are all of a directory element's derived objects

This is something you never have to worry about but it is important to understand that the internal structure of VOB is optimized for storage and performance.

3.1.2 Element

Elements are objects in the VOB that are versioned or can be represented with a version tree. In ClearCase, we deal with two basic types of elements:

A **file elemen**t is any file that can be stored in a file system. Most of the files that we will want to manage are text files, such as software source code files, design documents, scripts, XML documents and the like.

A **directory elemen**t contains file elements and other directory elements. The use of the term file and directory elements is similar to the way we use the terms when talking about the file system on a computer.

These are the only two types of elements that ClearCase distinguishes since directories by their nature contain information about other objects in a file system which means that ClearCase has to store some information for directories above what is adequate for a file.

This use of element type should not be confused with the idea of a file type (text, zip, compressed files, XML, etc) which we will defined later.

3.1.3 Version

A version is a revision of an element. The basic idea is that a version of an element, a file for example, is created when we take the original and modify (revise) it in some way. We are left with two different physical elements but logically, we think of them as different versions of the same logical document. In other words, what we normally think of as a version in our everyday life is pretty much the same as we mean by the term in ClearCase.

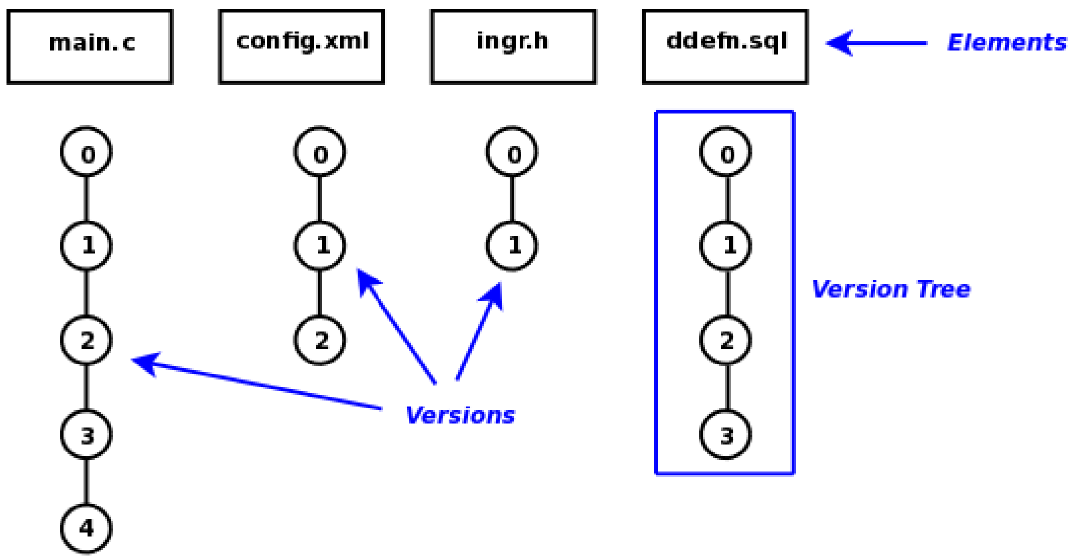
When we "version" an element, it means we make what we can think of as copies of the element at a known point that we can specify so that we can later, if we desire, return to an earlier version of the element. For example, if you are working in a word processor on a document, let's say your resignation letter, you know that it's a good idea to save the document at various points so that you will not lose your work in case the application should terminate unexpectedly.

This is not versioning because you can't go back to the state the letter was in two days ago. But suppose that each time you saved your letter, you saved it with a different file name: for example, your original might be letter.doc, and each time you save it you save it as letter1.doc, letter2.doc, and so on. While it is true that you have saved versions of the original letter, there are several problems with this process.

1. The only versions available to you are those you have remembered to save. Unless you save each with a different name (ie. as a new document) you will overwrite each version.
2. You are saving whole copies of the document, which means that if the document is large, saving copies will result in multiple copies of the document created to save the parts of the documents that are not changed.
3. The process can fail because you might forget to save a version, or you might save it with the wrong name.

What ClearCase does for you is automate this process and make it much more efficient. When you place an element under version control it means that you place it into a clear case VOB and let ClearCase maintain the various versions for you.

3.1.4 The Version Tree



Once an element is under versions control, ClearCase remembers every change made through the following process.

1 The user “checks out” a file from the VOB. Until a file is checked out, it is in “read only” mode and cannot be modified.

2 When a file is checked out, a copy of the appropriate version, and made available for editing.

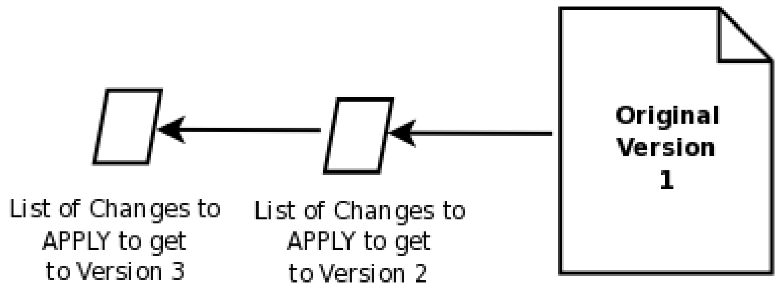
3 After the file is edited, the file is “checked in.” At this point, ClearCase records all of the “meta-information” about the new version, such as who made the edits and when, in various history and log files. All of the changes that were made are stored so that the new version and previous version that was checked out are both still available.

The versions that are built up in this was are called the *version tree* of a specific element.

3.1.4.1 Deltas

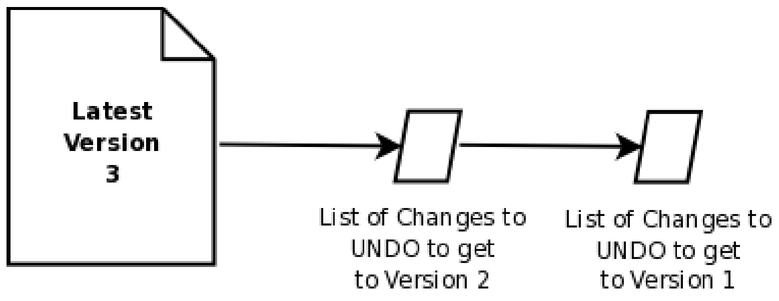
Most versioning systems, including ClearCase, are designed to be very efficient. If you created three versions of your letter and saved each one as a separate file in Word, then you would have three copies which all differ from each other in (probably) small ways.

Instead, ClearCase remembers the original and the set of changes you made in the original to create version 2. When you edited version 2 to get version three, ClearCase remembers the changes (the delta) to get from version 2 to version 3


Original Version 1
List of Changes to APPLY to get to Version 2
List of Changes to APPLY to get to Version 3

There are couple of different ways to store the version information. In a forward delta strategy, the original version is stored and then each set of changes (or deltas) is saved to allow the next version to be generated.

In the diagram above, if we wanted to bet version 3 of the document from the VOB, ClearCase would start with the original document and then apply the first set of changes to get version 2 and then then next set of changes to get version 3.


Latest Version 3
List of Changes to UNDO to get to Version 2
List of Changes to UNDO to get to Version 1

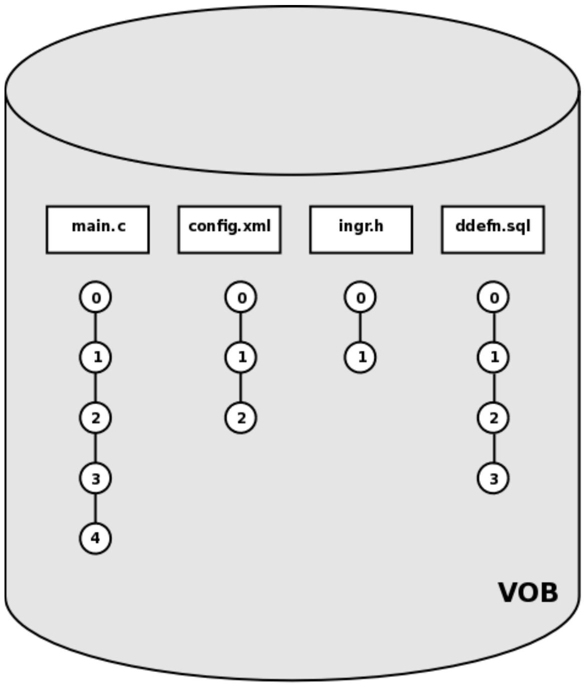
In the reverse delta strategy, a system keeps the most recent version to the document and remembers each delta or set of changes that were made to produce each version from the previous version. If our current version is version 3 and we want version 1, then the system would start with version 3 and then undo the changes that were made to get from version 2 to version 3 and then undo the changes required to get from version 1 to version 2.

By storing the deltas rather than copies, the overall performance in terms of both storage and speed are improved considerably.

ClearCase can be visualized as using the forward delta system but at the implementation level, it uses what is called am *interleaved delta* strategy where all deltas are maintained in a single structured file.

3.1.5 Versioned Object Base

We can now officially define a versioned object base (VOB) as a repository that stores versions of file elements, directory elements, derived objects, and meta-data associated with the versioned objects.



Normally a VOB looks like an empty directory when we create it. However there are certain kinds of special VOBs that can be created with a predefined internal structure. For example, a Project VOB (PVOB) is a special type of VOB that is used when UCM – Unified Change Management -- is used as the SCM process. If this is the case, then each UCM project belongs to a PVOB.

A VOB family is the set of all replicas of a particular VOB. We will not be seeing replicas in this course, but when a multi-site version of ClearCase is used, a single project or element may be worked on at different physical locations. To ensure that all locations are managed effectively, replicas or copies of the VOBs are used at the various locations. Obviously the amount of management when using a multi-site version of ClearCase is significantly more than when there in only one VOB to worry about.

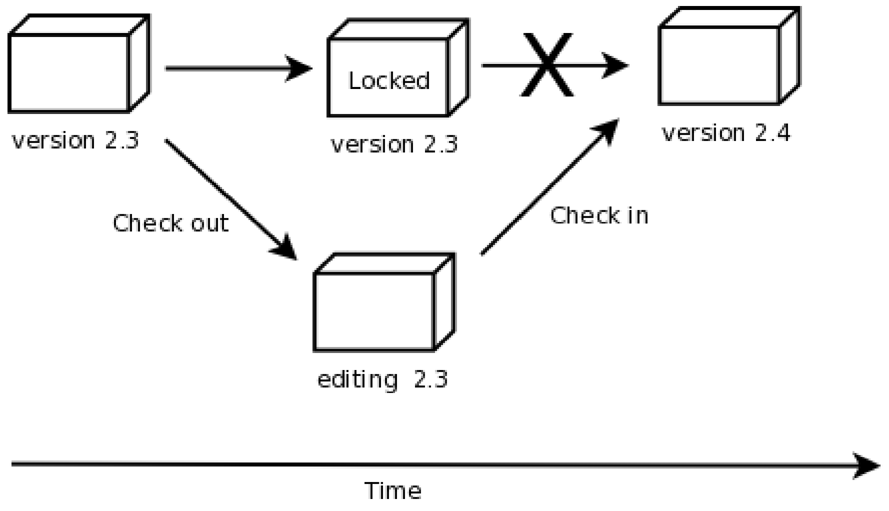
3.2 Checkout Models

The data as stored in the VOB is unusable by the user. Every time an element is

When a user checks out an element, ClearCase creates an editable copy in that user's view. When the user has finished their edits and checks in their edited element, a new version of the element is added to the VOB/

3.2.1 Reserved checkout models

The reserved checkout model allows a user to check out a file but then locks the checked out version so that one one can check out that file until the file is checked back in. This model is the safest model but can produce bottlenecks unless the probability of more than one user needing the same file at the same time is low.



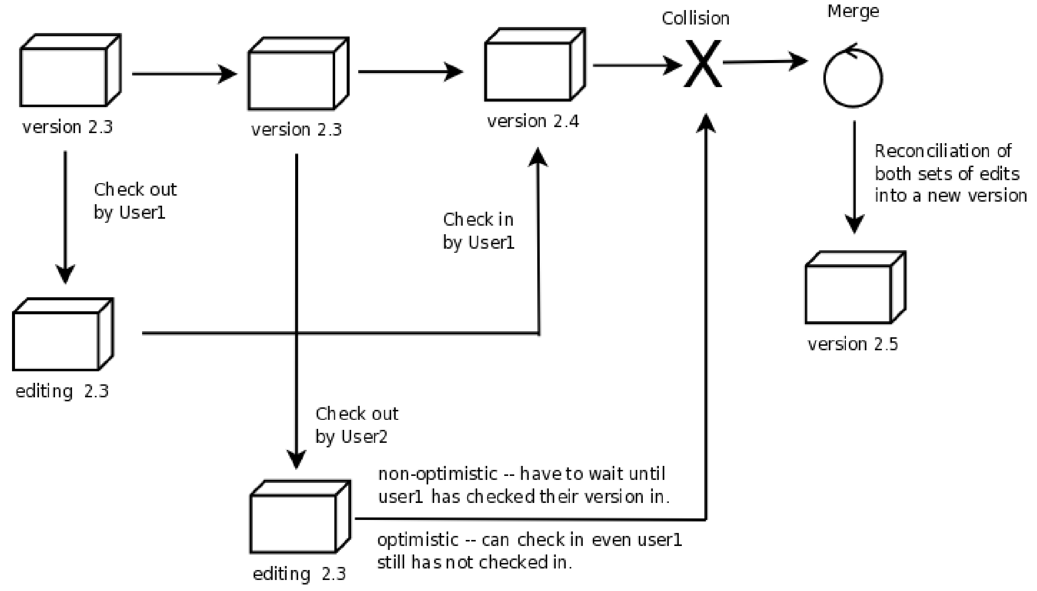
3.2.2 Concurrent checkout models.

ClearCase is designed to support large development efforts where we cannot assume that only one person at a time would be working on a particular file. To accommodate parallel development, ClearCase uses a concurrent checkout model to allow for more than one person to check out a specific element at the same time,

Concurrent checkout models come in two flavors.

3.2.2.1 Non-optimistic model

In a non-optimistic model, the order of checkout determines the order of check in. That means that the user who checks out the specific element first gets to check in the element. Suppose that user 1 checks out version 2.3. While editing, user 2 also checks out version 2.3. In the non-optimistic model, only when user 1's checkin as version 2.4 is complete can user 2 go ahead with their checkin attempt. Because the same checked out version of the element has already been checked in by user 1, the SCM repository will declare a collision situation that forces user 2 to perform a reconciliation of their edited version 2.3 and the new version 2.4. This reconciliation is usually performed with some kind of merge tool.


Collision
Merge
version 2.3
version 2.3
version 2.4
Check out by User1
Check in by User1
Reconciliation of both sets of edits into a new version
editing 2.3
version 2.5
Check out by User2
non-optimistic -- have to wait until user1 has checked their version in.
optimistic -- can check in even user1 still has not checked in.
editing 2.3

3.2.2.2 Optimistic Checkout Models

In an optimistic model, the order of check out is irrelevant which means that either user 1 or user 2. could check in first. However, when the second check in attempt is made by either user, a collision is detected, and a reconciliation has to be performed.

ClearCase provides non-optimistic checkout by default, but can be configured to be optimistic per file or by default.

3.2.3 Why ClearCase Allows Concurrent Checkout

Recall our discussion on process earlier. If we have a well defined development process, then our development work is planned. That means that developers should not be overwriting each others work because their activities are organized by a work activity plan. Having collisions when two developers are editing the same parts of the file at the same should be very, very rare.

ClearCase assumes that those who use the product are actually automating and existing software development process with a robust work planning component. The idea is that if there is a conflict when checking file it, the conflict should be resolved at the project level by asking the question “Why are two people making incompatible changes to same file?”

Merge resolution tools do NOT resolve conflicts, they only flag them so they can be resolved at the project management level.

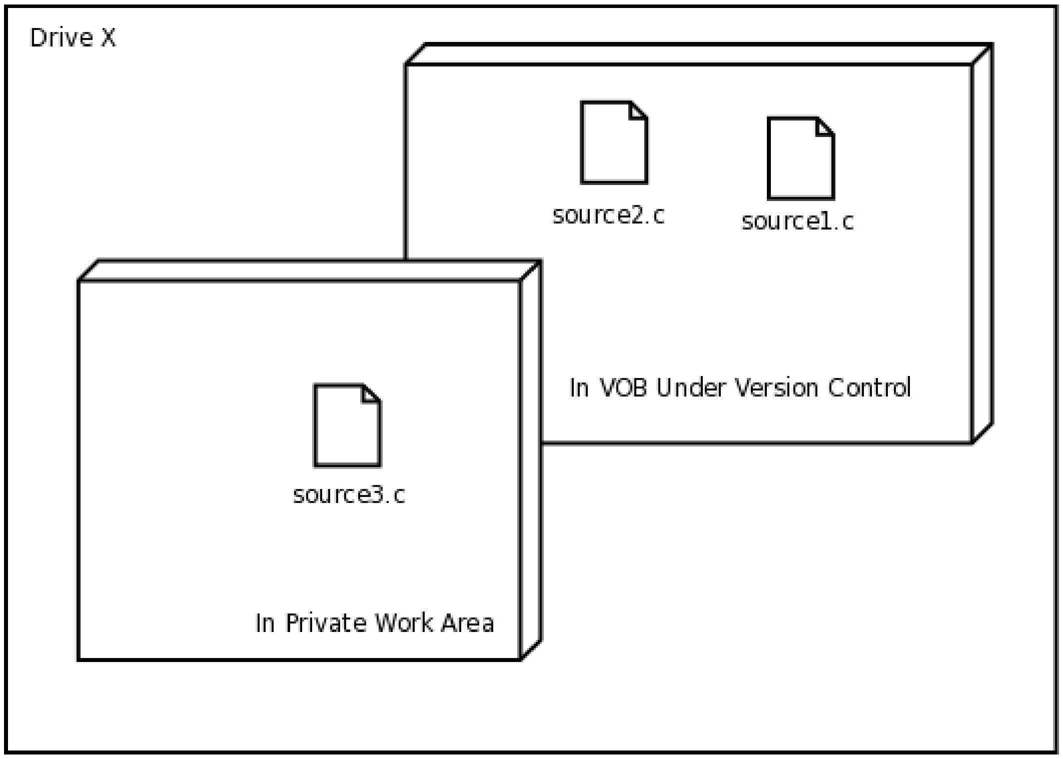
3.3 Working with your VOB

It is useful to think of a view as having two layers.

The back layer is made up of the elements that are under version control in the VOB

The front layer is the private workspace.

The elements in the back layer cannot be edited or modified until they are moved to the front layer. The elements in the front layer cannot be under version control until they are moved to the back layer..



In the diagram above, the files source1.c and source2.c are under version control and cannot be modified. Source3.c is not under version control but can be edited.

3.3.1 The Checkout/Checkin Model

Although you can see all of the files in the VOB, their default state is read-only which means that you can neither edit it nor remove it with standard operating system commands

```
% ls -l source1.c
  -r--r--r--   1 rod   user   168 May 13 19:30 source1.c
```

Remember that this is one version of the file element selected by your view, according to the rules in its config spec (we are just using the default spec at this point which typically shows the most recent version on some branch of the element's version tree).

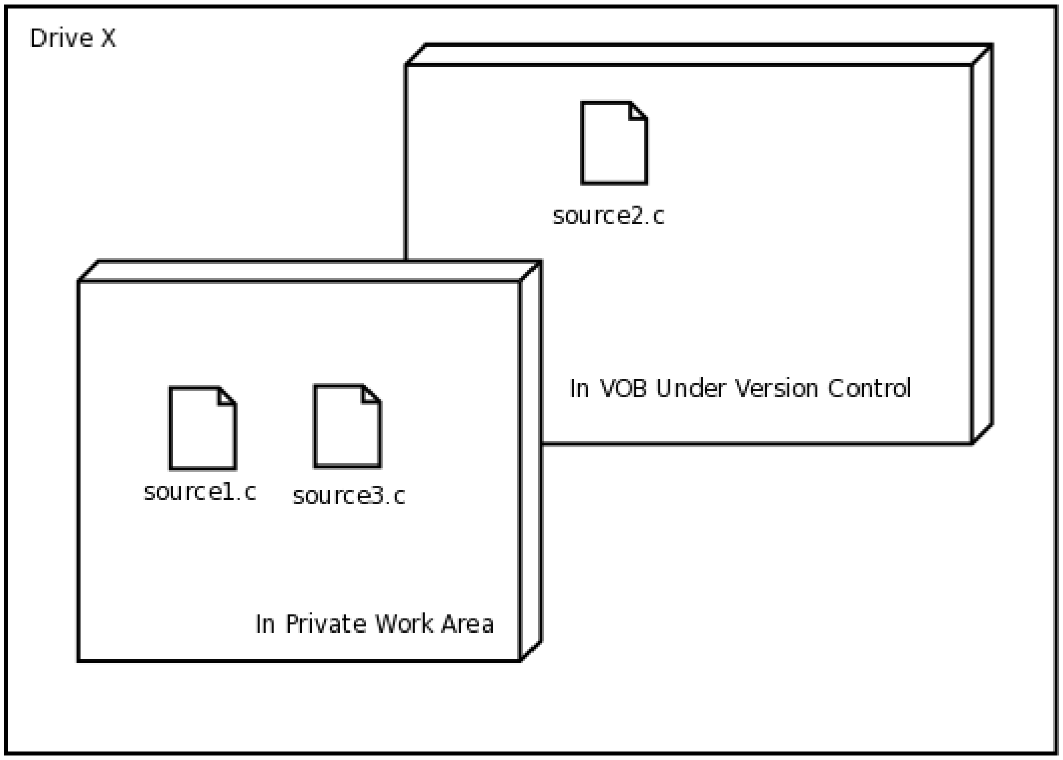
3.3.2 Checkout

Using the checkout command on the file produces an editable copy of the selected version "in place", meaning that the checked out file at the same pathname as the original element. The view manages this so there is no need to copy the file to another location in order to work on it.

For example, the element was created by another user, rod, who owns all its "old" versions. But the checked-out version belongs to the user who performs the checkout — in this example, ernie.

```
% cleartool checkout -nc source1.c
   Checked out "source1.c" from version "/main/2".
% ls -l source1.c
   -rw-rw-r--   1 sakai    user      168 May 19 19:31 source1.c
% vi source1.c
```

And the checked out version can now be edited



3.3.2.1 Checkin

When you decide to save the file, you issue a checkin command. This adds a new version to the version tree as the successor to the version that was checked out, and removes the editable copy of the file.

```
      % cleartool checkin source1.c
         Checked in "source1.c" version "/main/3".
```

3.3.2.2 Un-checkout

Once an element is checked out, you can cancel the check out without doing a check in, for example, if you decide that you don't want to modify the file after all.

```
      % cleartool uncheckout  source1.c
```

Whether you end the checkout via a check in or an uncheckout, the file reverts to its read-only status.

```
      % ls -l  source1.c
        -r--r--r--   1 rod      user      233 May 19 19:44
source1.c
```

The checked-in version is placed in VOB storage, and immediately becomes shared data, available to all users. In particular, your view now selects this newly-created version.

There are also shorter versions of these commands, since they are so common.

```
      % cleartool co anelement

      % cleartool ci anelement

      %cleartool unco anelement
```

These can also be done right in the Explorer menu by right clicking on the files and choosing the appropriate options.

3.4 Creating Elements

Remember that files and directories are both elements.

3.4.1 Creating new files

To create a new file, you can use whatever process you would normally use to create the file. The new file will be a view-private entity (i.e. not part of the VOB) until you take the next step to put it under ClearCase control.

To put a file under ClearCase control you must first check-out the directory that will be containing the file, issue the command to tell ClearCase to put the file under version control, and then check-in the directory containing the new file.

The **mkelem** command tells ClearCase to create a new element from the file.

```
% cleartool co thedirectory
% cleartool mkelem thefile
% cleartool ci thedirectory
```

A couple of things to note about the **mkelem** command. We can specify a type so that we can create directories and other types of files. Leaving off the type option forces ClearCase to do its best to ascertain tech file type.

Second, unless indicated otherwise, the **mkelem** command automatically checks out the created file which means you have to check it back in.

Examples:

Create file element, source3.c allowing the file-typing mechanism to determine the element type. Do not check out the initial version.

```
% cleartool mkelem –nc –nco source3.c
  Created element "source3.c" (type "text_file").
```

Create two directory elements and check out the initial version of each.

```
% cleartool mkelem –nc –eltype directory libs include
   Created element "libs" (type "directory").
   Checked out "libs" from version "/main/0".
   Created element "include" (type "directory").
   Checked out "include" from version "/main/0".
```

3.4.2 Creating New Directories

The *mkdir* command creates one or more directory elements and is equivalent to the *mkelem –eltype.directory* command Unless you specify the *-nco* (no checkout) option, the new directory is checked out automatically. A directory element must be checked out before you can create elements and VOB links within it.

```
% cleartool mkdir –nc subd
   Created directory element "subd".
   Checked out "subd" from version "/main/0".
```

3.5 Deleting elements

To remove an element you should use the cleartool **rmname** command. This command removes the name of the specified element from the directory in which it is contained. Similar to placing an element under ClearCase control, you must first check-out the directory containing the element to be removed, do the cleartool **rmname** of the element, and then check-in the new version of the directory.

```
% cleartool co thedirectory
% cleartool rmname anelement
% cleartool ci thedirectory
```

After this sequence, *anelement* is no longer listed in subsequent versions of *thedirectory*. It is, however, still listed in previous versions of the directory. If, at a later date you want to re-add anelement to a new version of thedirectory (or any other directory), you can do so easily.

3.6 Working with Check outs

In a multiuser environment, it is very likely that a given element will have several checkouts at the same time because either:

1. several branches of the element may be under development; checkouts on different branches are mutually independent; or
2. there can be multiple concurrent checkouts of a single version.

3.6.1 Viewing Check outs

The *lscheckout* command lists all the current checkouts of one or more elements:

```
% cleartool lscheckout -long util.c

04-Mar-11.12:12:33     John K. Smith (smith@hobbs)
 checkout version "util.c" from /main/37 (reserved)
 by view: "/hobbs/home/smith/views/930825.vws"
26-Feb-11.08:59:02     Sally R. Jones (jones@hobbs)
 checkout version "util.c" from /main/debug/8 (reserved)
 by view: /hobbs/home/jones/mainvu.vws
"incorporate david's comments"
```

Some other examples:

```
% cleartool lscheckout -long -me
```

The -me option lists only those elements that you have checked out.

```
% cleartool lscheckout -cview -avob
```

The -cview option narrows the search to elements checked out in the current view. The -avobs option specifies to search in all VOBs.

3.6.2 Checkout and Checkin as Events

The *lscheckout* command determines all of an element's checkouts by examining event records, which are stored in the VOB database of that element. Each *checkout* command creates a checkout version event record; *lscheckout* lists some or all such event records.

Similarly, the *checkin* command writes a create version event record to the appropriate VOB database. In general, every ClearCase operation that modifies a VOB creates an event record in the VOB's database, capturing the “who, what, when, where, why” of the operation: login name of the user who entered the command, kind of operation, date-time stamp, hostname, user-supplied comment.

You can use the lshistory command to display some or all of the event records for one or more elements:

```
% cleartool lshistory util.c
  25-May-12.15:45:19     John K. Smith  (smith@hobbs)
        create version "util.c@@/main/3" (REL3)
        "special form of username message for root user
         merge in fix to time string bugfix branch"
  25-May-12.15:44:05     Sally R. Jones (jones@hobbs)
         create version "util.c@@/main/rel2_bugfix/1"
        "fix bug: extra NL in time string"
  25-May-12.15:43:03     Sally R. Jones (jones@hobbs)
         create version "util.c@@/main/rel2_bugfix/0"
  25-May-12.15:43:03     Sally R. Jones (jones@hobbs)
         create branch "util.c@@/main/rel2_bugfix"
  25-May-12.14:46:21     Sally R. Jones (jones@hobbs)
         create version "util.c@@/main/2"
         "shorten HOME string"
```

3.7 Importing Files

One way to move existing files into the VOB is to import them. ClearCase has a number of utilities that allow you to import files from other SCM tools into ClearCase. However, for simplicity, we will only look at the command that imports files from the file system,

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 which is a way for ClearCase to figure out what it is importing.

The *clearfsimport* allows you to see what sorts of things would happen if you did the import without actually importing anything. To get this "test it out" functionality, use the *-preview* option.

You should be in the VOB directory when you use this utility

Example

In this example we preview the VOB /vobs/projectx/src that is to be populated with the contents of /usr/src/projectx. Recursively descend all directories encountered and follow UNIX symbolic links to the target objects. To actually do the import we would rerun the command without -preview

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

ClearCase Comprehensive

Module Four

Working with Views

We do not see the world as it is but rather as we are
Anias Nin

4.1 Views

So far we have been using the default view and focusing on the elements in the VOB. In this module, we will be looking at the view in more detail and how it works to show us a particular point of view when we look at the contents of a VOB.

Even if a VOB is active, you cannot access it directly. All user-level access to a VOB must go through a ClearCase view. When we mount a VOB, it appears as a directory on our local file system, but if we locate to that directory without a view, that VOB's mount point appears to be an empty directory. However, when seen through a view, a VOB appears to be an entire directory tree. Each file and directory in this tree in an element, which has a version tree containing all of its historical versions.

What a view does is act as a translator that converts the contents of the VOB from the internal ClearCase format to something that looks like an ordinary file system directory or file

4.1.1 Listing Views

Just like for VOBs, we use a **lsview** command to list all the available views.

```
% cleartool lsview
  *   mainRel5     /net/hobbs/viewstore/ainRel5.vws
      anneRel5     /net/hobbs/viewstore/anneRel5.vws
  *   anneTest     /net/hobbs/viewstore/anneTest.vws
      nordTest     /net/hobbs/nord/nordTest.vws
      nordRel5     /net/hobbs/viewstore/nordRel5.vws
      nordRel4      /net/hobbs/viewstore/nordRel4.vws
```

We can also check which views are available in the Explorer, however the Explorer sometimes needs to be refreshed because it's list of views may get “stale.”

4.1.2 Snapshot and Dynamic Views

There are two kinds of views in ClearCase.

4.1.2.1 Dynamic Views

Dynamic views use a real-time connection to VOBs. Anytime a change is made to a VOB from anywhere, if we are looking at that VOB through a dynamic view, we see that change as soon as it is made.

Anytime we make a request of the VOB, like a file system command, the view in real time issues a request to the VOB to construct the appropriate file.

A dynamic view requires a live and constant connection to the VOB. This can produce problems in some organizations where the network traffic of the dynamic view is not compatible with the network infrastructure – especially security protocols.

4.1.2.2 Snapshot Views

A snapshot view does not require a connection to the VOB. When it is created, the view makes a one time batch request to the VOB to construct a copy of itself, and then saves this copy in a local directory. We then use this local copy of the VOB as if it were the VOB.

A snapshot view can be used even when we are disconnected from the network. However, because we are not connected, it is possible for the snapshot copy to get out of sync with the actual VOB.

When we reconnect to the network, we have to update our snapshot view so that we get all the latest changes from the VOB and also to send our local changes back to be incorporated into the VOB.

4.1.3 Creating a View

Unlike creating a VOB which is an activity you are unlikely to do often if at all, you may find yourself creating various views to assist you in your work. There are two ways to create a view, either at the command line or through the View wizard.

4.1.4 mkview

The **mkview** command creates a new view as follows:

1 Creates a view storage directory. The view storage directory maintains information about the view. Along with other files and directories, the directory contains the view’s config spec and the view database.

2 Creates a view tag, the name by which users access a dynamic view. Snapshot views also have view tags, but these are for administrative purposes; users access snapshot views by setting their working directory to the snapshot view directory (for example, using the cd command.

3 For a snapshot view, creates the snapshot view directory. This is the directory into which your files are loaded when you populate the view using update. This directory is distinct from the view storage directory.

4 Places entries in the network’s view registry; use the lsview command to list view tags.

5 Starts a view_server process on the specified host. The view_server process manages activity in a particular view. It communicates with VOBs during checkout, checkin, update, and other operations.

Each view must have a unique, descriptive name, called a view tag. A view tag must be a simple name containing no special characters. While spaces are legal, it is better to not use view tags that contain spaces. The name choose a name should help you determine the owner and purpose of the view. Names like myview or work do not describe the view's owner or contents; if you work with more than one view, such generic names can lead to confusion.

,

4.1.4.1 Creating a Dynamic View

```
      % cleartool mkview -tag atag  storage-location
      % cleartool mkview -tag fixupview -stgloc -auto
```

The user defined name *atag* is the view-tag or the "name" of the view you have chosen. The s*torage-location* is the location of a place on a disk where the view can store information.

4.1.4.2 Creating a Snapshot View

Very similar to the dynamic view.

```
% cleartool mkview -snapshot -tag tagname -stgloc -auto dirspec
```

Where *dirspec* is the path to which you want your source code or view work area to reside. For snapshots the *-tag tagname* is optional. If it is omitted, ClearCase will make one up based on your username and the workspace you specify in *dirspec.*

4.1.5 Using a View

In Unix we use a view by issuing the **setview** command.

```
      % cleartool mkview -tag qa_view /home/views/qa.vws
      % cleartool setview qa_view
```

In Windows, dynamic views are mapped to network drives, to use a view, just locate to the associated network drive.

For snapshot views, the view element selection is executed once when the view is created, so to use a snapshot view, we just locate to the local copy of the VOB.

For now, let's suppose that the existing view gordons_view is available for your use. The easiest way to use a view is to “set the view”. Setting a view creates a new shell process that you can use to work with any active VOB since after setting a view, you can work with any VOB, much as if it were a standard directory tree:

That means that you can navigate around the VOB using cd, ls, and other standard Unix type file system commands. It also means that you can view and edit files with cat, more, vi, emacs, or whatever other tools you use, including tools that analyze files like grep, sed, awk, and so on.

The intent in ClearCase is to make the actual operations on the files and directories as close to the usual sorts of operations as possible.

For example:

```
% cleartool mount /vobs/design    We mount a public vob
% cd /vobs/design                 Locate to the directory
% ls -F                           When we list -- nothing there
%
% cleartool mkview -tag myview -stgloc -auto      Create a view
% cleartool setview myview                       And Set the view
% ls -F
  bin/   include/   lost+found/  src/   test/
```

4.2 Labels

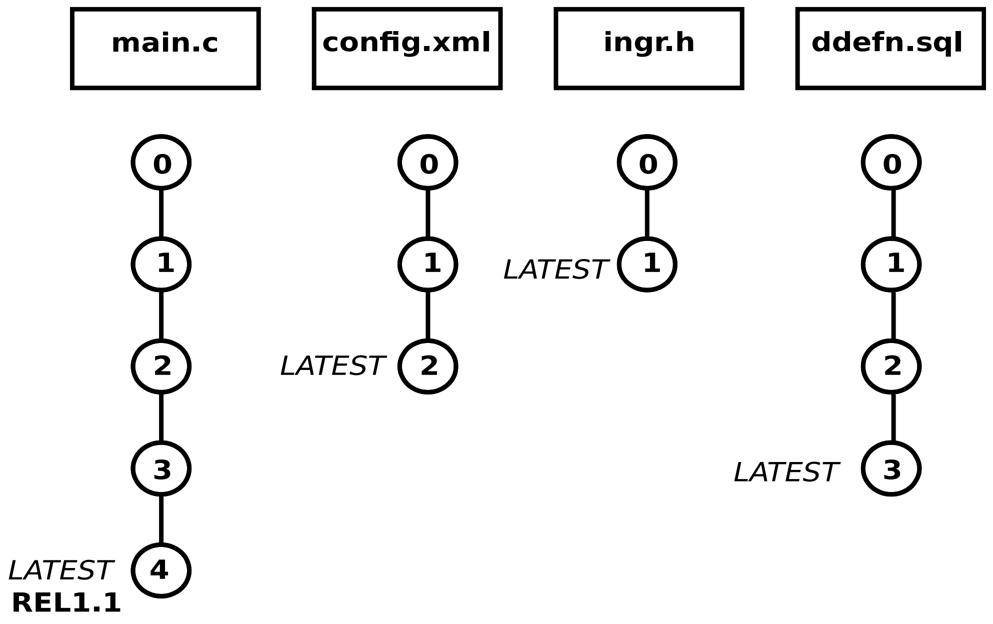
Some versions of an element may have special significance, for example it was the version that went into a specific release.

In order to mark a particular version, we can attach a label to it. Labels must already be defined in a VOB in order to be attached to a particular version of an element. We do this by issuing a make label type command.. In the example below we create a label REL1.1 and then attach it to the current version of a file util.c

```
% ct mklbltype REL1.1
% ct mklabel REL1.1 main.c
```

By convention labels are always uppercase. Because they uniquely identify a particular version of an element, the same label cannot appear on two versions of the same element.

The above code would produce the following version tree.



ClearCase has has several automatic labels, the one we will see most is the “LATEST” label which allows us to always refer to the latest element in a version tree without needing to know the verision number.

4.3 Deleting Views

You can either use the Explorer menu to delete a view or do it at the command line

4.3.1 Deleting a Dynamic View

```
% cleartool rmview -tag fixupview
```

Since the view doesn't actually contain anything, we don't have to worry about losing anything, unlike the case of removing a VOB.

4.3.2 Deleting a Snapshot View

Since we have specified the view to be at a specific location, that is all we need to specify to delete the view.

```
% cleartool rmview dirspec
```

4.4 Branching and Merging in ClearCase

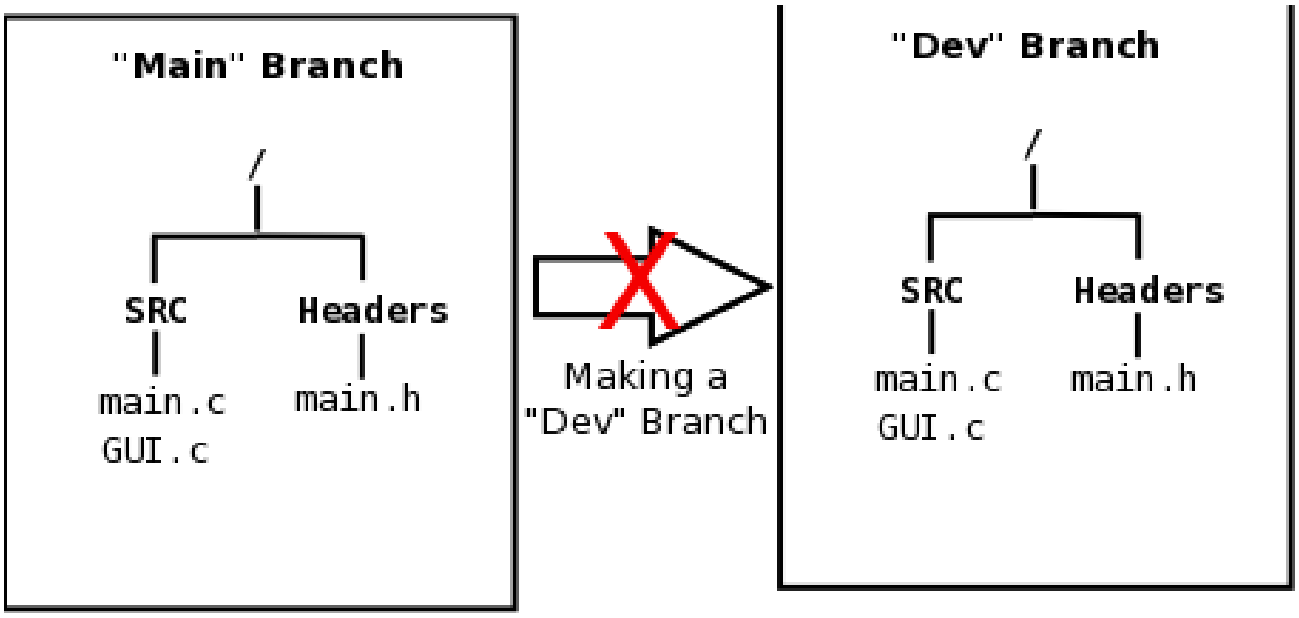
One of the criticisms that is often leveled at ClearCase is that it does not enforce any specific branching model or protocol, which means that anyone who knows the appropriate ClearCase commands can execute branches and merges. Often with chaotic results.

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.

4.4.1 Branching Concepts in ClearCase

One of the common misconceptions that the average user of ClearCase as is that branches are sort of like containers. We create a branch container and then copy our files into it.



The terminology is a bit confusing because when we say that we “make a branch” it does sound exactly like that is what we are doing. There is often the idea that the branch is a complete and independent copy of the source tree (or VOB in some people's minds).

However to strictly correct, we need to say that we “branch a file.” Branching is process of creating a copy of a file and then attaching a label to it.

4.4.1.1 Creating a Branch Label.

Since a branch in a copy of a file, we need a label or identifier to tell the original file from the copy. The rule in ClearCase is:

Every file in ClearCase has either a user defined branch label or the default label “main”

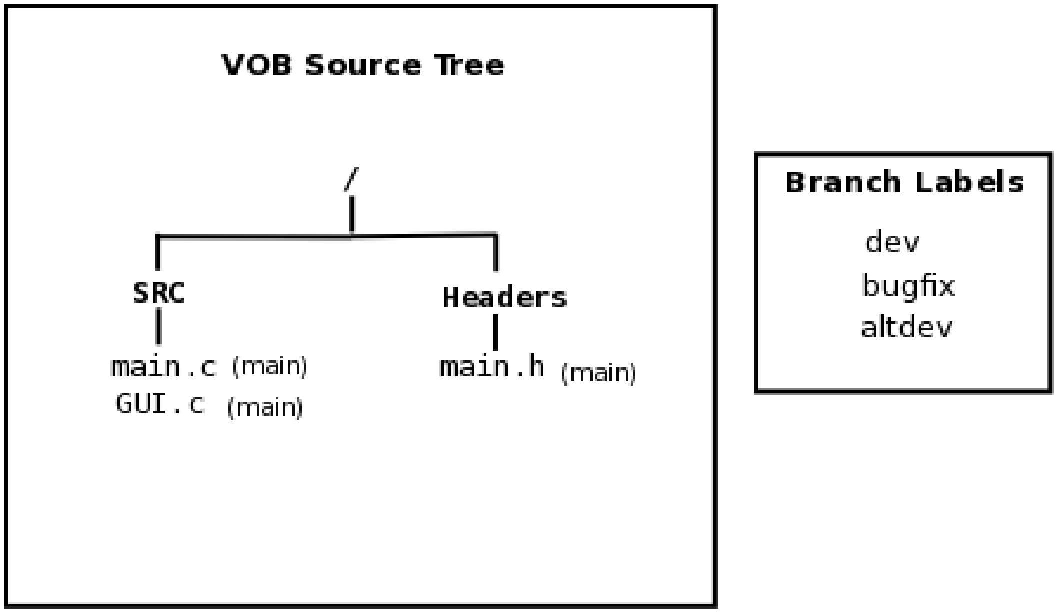
We need then to have a pool of branch labels to choose from which are created by the project administration using the **`mkbrtype`** command.

```
% cleartool mkbrtype -c “main development branch” dev

% cleartool mkbrtype -c “release branch for bugfixes” bugfix

% cleartool mkbrtype -c “government development version” altdev
```

Looking at the source tree in the previous diagram, the ClearCase view of the tree includes the branch labels.



Now we can branch the individual elements using the **`mkbranch`** command

```
      % cleartool mkbranch -nc dev main.c

      % cleartool mkbranch -nc bugfix main.c

      % cleartool mkbranch -nc altdev main.h

      % cleartool mkbranch -nc dev main.h
```

4.4.2 Creating a Branch

Creating a branch involves the following two operations:

1. Create a branch type to enable the instantiations of branches. You do this by running the mkbrtype command.
2. Instantiate a branch. To do so, you use the mkbranch command.

To create a branch type object, you use the cleartool mkbrtype command.

```
% cleartool mkbrtype -c “bugfix branch” debug
```

The above command creates a branch type object named debug which can be used only once in an element’s version tree. The -c option allows for entering the comment “bugfix branch” Instead of the above we could have created more than one label

```
% cleartool mbrtype -nc -pbranch debug bug102
```

This command creates two branch types: debug and bug102. The -nc option indicates that you do not wish to provide a comment.

4.4.2.1 Instantiating a Branch

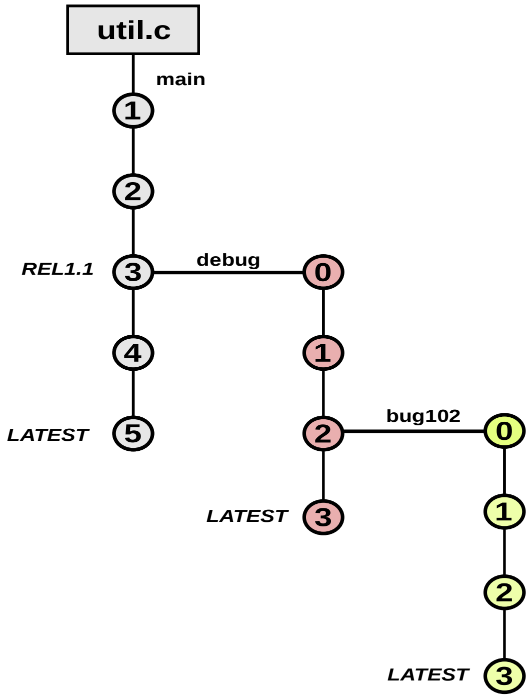
After you have created the branch type objects, you can create branches of that branch type. You do this by running the mkbranch command. The mkbranch command creates a new branch in the version trees of one or more elements. By default, the branch is checked out unless you use the -nco (no checkout) option.

In most cases, branches are created when you check out files or directories. The mkbranch rules in your config spec specify when new branches should be created (we will see this in the next module)

```
% cleartool mkbranch -nc debug util.c
```

This command creates a branch of the type debug off the version of util.c in the view and checks out the initial version on the branch.

A typical version tree could look like this:



4.5 Version Pathnames

```
% ct ls -l
      version                   Makefile@@/main/3
      version                   main.c@@/main/4
      view private object       util.c
      version                   ops.c@@/main/CHECKEDOUT
                          from /main/3
```

In this listing we see the full pathnames for the various objects in the view.

4.5.1 Elements, Branches, and Versions

Each ClearCase VOB stores version-controlled file system objects, termed elements. An element is a file or directory for which ClearCase maintains multiple versions. The versions of an element are logically organized into a hierarchical version tree, which can include multiple branches and subbranches.

Some elements may have version trees with a single branch in which case the versions form a simple linear sequence. But typically, users define additional branches in some of the elements, in order to isolate work on such tasks as bug fixing, code reorganization, experimentation, and platform-specific development.

In ClearCase version trees:

1 Each element is created with a single branch, named main, which has an empty version, numbered 0.

2 ClearCase automatically assigns integer version numbers to versions. Each version can also have one or more user-defined version labels (for example, REL1, EL2_BETA, REL2).

3 One or more branches can be created at any version, each with a user-defined name. (Branches are not numbered in any way.)

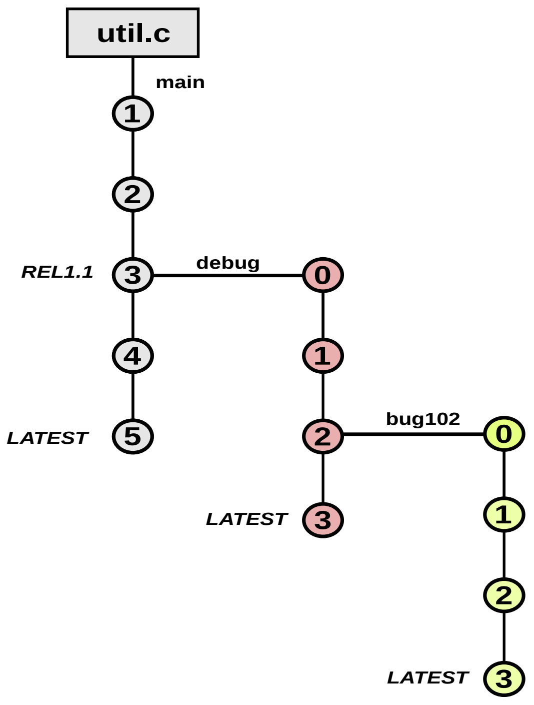
4 ClearCase supports multiple branching levels.

5 Version 0 on a branch is identical to the version at the branch point.

4.5.1.1 Version-IDs

Each version of an element has a unique version-ID, which indicates its location in the version tree. A version-ID takes the form of a branch-pathname followed by a version-number .

For example, the LATEST versions at the end of the branches in the tree at the top of the next page have these version-IDs:



/main/5

/main/debug/3

/main/debug/bug102/3

Note that the following are also equivalent

/main/3 = /main/REL1.1

/main/debug/bug102/3 = /main/debug/bug102/LATEST

4.6 Extended Pathnames

Each view selects only one verison for a given element. Through a view, whatever verison of the source file th ais selected is referenced by the uslwhich we will call *util.c* . But the VOB also contains all other historical versions of the element, all of them potentially accessible through the file name *util.c.*

ClearCase has a VOB-extended pathname scheme, which enables you to access any version of an element, no matter which version is selected by your view. You can also reference other components of an element's version tree: its branches, and the element itself.

An element's version tree has the same hierarchical structure as a directory tree. This makes it natural to “embed” the entire version tree in the file system under the element's pathname:

util.c	standard name of an element
util.c@@	extended pathname to the element object
util.c@@/main	extended pathname to the element's main branch
util.c@@/main/debug	extended pathname to a subbranch of the main branch
util.c@@/main/debug/3	extended pathname to a version on the subbranch

Think of the extended naming symbol (@@) as “turning off” transparency, allowing you to specify a particular object in the VOB database.

Version-extended pathnames can be used in standard commands, either alone or in conjunction with standard pathnames. For example, this command searches for a character string in the “current” version and in two “historical” versions:

```
% grep  "InLine"  monet.c  monet.c@@/main/13  monet.c@@/RLS2.5
```

The entire version tree of an element is embedded under its standard pathname. This enables users to access any (or all) historical versions through pathnames, using any program. The following variant of the above grep command shows the power of this file system extension:

```
% grep  "InLine"  monet.c@@/main/*
```

In this command, a standard shell wildcard character (*) specifies all the versions on an element's main branch.

4.7 The View as a Version Filter

ClearCase enforces the use of a view to access a VOB. Because all of an element's versions are accessible through the element's standard pathname, the view uses the selection rules in its config spec to select one of the versions. This means that a view makes a VOB appears to be a standard directory tree to system software and third-party applications. In other words, ClearCase is *transparent* to that software. Remember the design principles from earlier.

4.7.1 Configuration Specifications for Views

Each ClearCase VOB contains all the versions of the file system objects in a particular source tree. The view dynamically filters the contents of all available VOBs so that the actual contents of a selected version remain in VOB storage; the version is then presented into the name space of the view.

A config spec has the form:

<element type> <file name> <version selector>

For each element in a VOB, this rule would be applied in the following manner

1. Is the element of the type <element type>? If there is no match, skip this rule.
2. Does the element's file name match <file name>? Is not, skip this rule.
3. Is there a version matching the <version selector>? If so, then return that version from the VOB, if not, skip this rule.

There are a couple of conventions that make writing the rules easiier. For example using “element” as the <element tupe> matches every type. And for the <file name>, we can use the same sort of fiel name globbing that we use in a bash shell.

For exampe the following rule always produces a result. It reads “for any element with any file name, return the latest verison on the main branch.” Since there always a main branch and a latest version, the rule aways produces a result.

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

Simlarily, the following rule reads “if the element is of type c_file and has a file name ending in either .h or .c, if there is a verison with the tag REL1.1, return that verison.”

```
element -eltype c_file *.[ch] REL1.1
```

If there is no verison with the tag REL1.1, then this rule will fail.

We can also reference a branch as well.

```
Element * .../dubug/LATEST
```

This rule says “for any element with any file name, it there is a debug branch in the verison tree, then return the latest verison on that debug branch.

Each view is acutally a server process: view_server. The config spec rules, allow the view_server process to convert each file system reference to an element, meaning the normal file name like “util.c” into a reference to a fully qualitified specific version of that element, like util.co@@/main/debug/7. \

The process works the following way.

1 Any reference to a path name, either explicitly ro implicitly such as via an applikcaiton like a compiler, is identified by the ClearCase MVFS,which processes all pathnames within VOBs, which then passes the pathname on to the appropriate view_server process.
2 The view_server attempts to locate a version of the element that matches the first rule in the config spec. If this fails, it proceeds to the next rule and, if necessary, to succeeding rules until it locates a matching version.
3 When it finds a matching version, the view_server “selects” it and has the MVFS pass a handle to that version back to the user-level software.
4 The process stops as soon at the first matching rule. If more than one rule matches, only the first matching urle is used.
5 If no rule matches, nothing is returned.

Every view is created with the default config spec:

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

Which basially says “for any element, if there is a checked out version, then show that version, otherwise show the latest version on the main branch.” Notice again that this config spec will return some verison for every element in the VOB.

4.7.2 Working with a View and Its Config Specs

After you create a new view, you may need to revise its config spec, in order to select a particular configuration of source versions. Every view is created with the default config spec:

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

I

n many organizations, new development takes place on the main branch, while special projects take place on subbranches. The default config spec is appropriate for new development work in such a situation.

There are several ways to reconfigure a view with a non-default config spec:

Copy a project-specific file — The ClearCase administrator may publish the pathname of a file containing the correct config spec for your project. Use the *setcs* command to reconfigure your view; then use *catcs* to confirm the change. For example:

```
% cleartool setview gomez
% cleartool setcs /usr/local/lib/munchkin_proj
% cleartool catcs
.
. <new config spec displayed>
```

Include a project-specific file: Instead a copying a file, you can incorporate its contents with an include statement:

```
% cleartool setview gomez
% cleartool edc
 .
      . use text editor to remove all existing lines,
        and then add this one:
  include /usr/local/lib/munchkin_proj
```

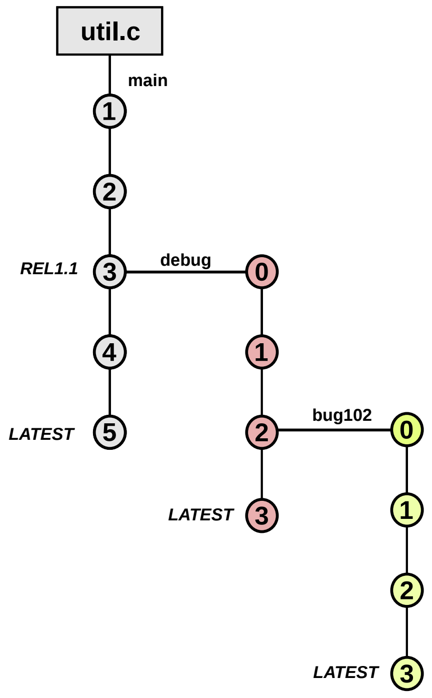
If the administrator subsequently revises the contents of that file, you can update your view's configuration by having the view_server reinterpret its current config spec:

```
% cleartool setcs -current
```

4 **Compose your own config spec:** There is no single method for composing a set of rules that go into a config spec. The language described in the config_spec manual page has many features, and can be used to create many “special effects”.

4.7.3 Some Examples

Using the verison tree below.



The follwing config specs produce the following versions. The matching rule is italics

```
element * CHECKEDOUT
element * /main/LATEST  => (util.c@@/main/5)
```

```
element * CHECKEDOUT
element * .../debug/LATEST => (util.c@@/main/debug/3)
element * /main/LATEST
element * CHECKEDOUT
element * .../big102/LATEST => (util.c@@/main/debug/bug/02/3)
element * /main/LATEST
```

```
element * CHECKEDOUT
element * .../bug102/LATEST => (util.c@@/main/debug/bug/02/3)
element * .../debug/LATEST
element * /main/LATEST
```

```
element * CHECKEDOUT
element * REL1.1           => (util.c@@/main/3)
element * .../bug102/LATEST
element * .../debug/LATEST
element * /main/LATEST
```

```
element * CHECKEDOUT
element * /main/LATEST  => (util.c@@/main/5)
element * REL1.1
element * .../bug102/LATEST
element * .../debug/LATEST
```

The examples show how important the ordering is in the rules and why the "/main/latest" rule always needs to be the last rule in the config spec since it always matches – i.e. nothing after that rule will ever be tested.

Similarily the first rule should always be the first rule of you would never be able to see any of your checkouts – you would check out a file and it would vanish but the whole point of checking it out is to be able to work on it.