IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden

Marc Girod **Tatiana Shpichko**

[PACKT] PUBLISHING enterprise professional expertise distilled

Table of Contents

4
Version Control

Dear Reader,

You have already entered the second part of our book. We stated our main thesis (didn't you notice?). What remains for us is to make the dream live! It will be a feast and a fight, of minute details, on a narrow path between slippery slopes.

The CM tradition was built on top of versions of source artifacts, crafted by hand: its world was a construction ex nihilo. By contrast, our world is always already there. Furthermore, it is not a cookbook trying to blow a sparkle of life into an assembly of carbon atoms: on the contrary, it is a frenetic fight to make sense of a universe of life all around us, and which we are part of. The only stability there is the one we are responsible for: meaning. ClearCase made it possible to put SCM back on its feet, after so many years of its standing on its head.

Derived objects are found in sets of instances: they meet there the fundamental requirement for **configuration items**, as set by the theoretical SCM framework. In fact *they* represent the prototype. Elements are what ClearCase names the hand crafted source artifacts. They can be considered as degenerate derived objects: the leaves of the dependency tree. Contrary to the derived objects, which build up a deep structure (the dependency tree), they are found at the surface of the software configuration space: they are flat, and therefore opaque from a *generic* point of view (but we do not forbid *specific* analysis, using file formats aware tools, and interpreting the data in ways that make structure arise).

We'll focus in this chapter on this small subset of the configuration items: the most simple, the most humble ones.

Historical concerns are neither jeopardized nor forgotten. We'll show here how they are supported by ClearCase, in ways that only minimally depart from standard expectations:

- Making new elements
- Check out and check in, at least in simple cases, and the subtleties of managing memory
- Looking for differences between versions
- Answering some misguided critiques

Making elements

Elements, i.e. potential families of versions, may be created either empty or from view private files (but not from private directories). The command is `mkelem`, and there exists an `mkdir` shortcut for directories (equivalent to `mkelem -eltype directory`). The same operation, if carried out from the GUI, is termed *add to source control*.

Whether or not obvious, let's note that one needs to **checkout** the parent directory before adding new elements, and to **check** it **in** afterwards before others may access the result of the operation: one cannot create an element without assigning it a path. Note that a newly created vob comes with a checked in root directory.

We noticed a restriction: *not from private directories*. It is worth mentioning that many such orthogonality glitches have been ironed away by David Boyce, in his wrapper (or actually wrappers) available from CPAN: ***ClearCase::Wrapper*** (and ***ClearCase::Wrapper::DSB***). Let's take this as an example: he offers `mkelem` a `-recurse` option (among others) which, one soon forgets, is missing from the original cleartool.

If adding several elements to the same directory, it is usually better to check out the directory once, add all the elements, and check in the directory once. This creates fewer events in the history, results in a smaller *version tree* of the directory, and is simply faster.

The `mkelem` command has flags that allow to deal with details, which otherwise would require some user interaction:

- Giving a comment (`-c`) or not (`-nc`). The comment is duplicated between the element and the first version if checked in simultaneously.

- Avoiding to check out the new element from the initial version `/main/0` (`-nco`). The default behavior is to check out the initial version and remove the view-private copy of the data. Note that the `/main/0` version is always empty, and in case the `-nco` option was used along with a non-empty view-private file, the element with the specified name gets created, and the view-private file is renamed to yet another view private file with the additional extension `.keep` (in order not to **eclipse** the element created in its `/main/0` version).
- Checking in the initial version *(*`-ci`), which got checked out by default, and create the version 1 on some branch (depending on the **config spec**).

In relation with the default checkout behavior, it is important to set the config spec correctly. It must contain the following line in order to make element creation possible:

```
element * /main/0
```

This line can also be:

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

We mentioned this in the presentation of config specs in Chapter 2.

As any element gets initiated with (an empty) version 0 on the branch `main`, and in order to check it out (may be to another branch), it must be selected by the config spec.

Elements have a type, which affects the way they are stored and the way their versions may be compared. This type is determined from their name, and possibly their contents, unless it is given explicitly with an `-eltype` option. We'll develop on these in *Chapter 9*, *Secondary Metadata*, but may already mention the existence of a **magic file**, with rules to determine the type of the new element. The default ClearCase magic file (`/opt/rational/clearcase/config/magic/default.magic`) falls back to `compressed_file` if no other suitable match was found.

Mass creation of elements requires other tools:

- *clearfsimport*, if importing from a flat directory structure
- Tools of the *clearexport* family, followed with *clearimport*, if importing version trees from other tools to ClearCase
- *cleartool relocate* to export data from one ClearCase vob to another

We'll cover those in *Chapter 8*, *Tools Maintenance*, and *Chapter 10*, *Administrative Concerns*. The development model encouraged by ClearCase is continuous, with early publication (under user/consumer control) of small increments. One easily understands that the issue of mass creation should be considered rather exceptional. Again, it is worth mentioning that a tool far more flexible and configurable than *clearfsimport* is David Boyce's *synctree* (see Chapter 10).

The last source of elements is other replicas, but this aspect belongs to *Chapter 5*, *MultiSite*.

If you remember our *Presentation of ClearCase* (*Chapter 2*), you'll recall that a critical aspect of the creation of a new directory is the *umask* of the user: this drives the ability for others to produce derived objects or to add new elements to this directory, and even to checkout existing elements without complex syntax (see later). In most cases, directories should be *writable to group*.

Checkout and checkin

Elements under ClearCase are fully accessible (but read-only) as **checked in**, e.g. to tools such as compilers. **Checking out** implies thus in general terms an intention to *modify* an element (we'll discuss later in this chapter the special case of tools requiring that some files be kept writable). A version checked out becomes a view-private file and is not shared with others (unless via a view-extended path) until it's checked in. This is why checking out is at best deferred as late as possible, and also done on a file basis, as needed. In the same way, checking in should happen as soon as possible, as it provides a fine-grain backup: it is easily decoupled from *delivery* to others. The set of checked out files should therefore be, at any moment, minimal, and give an indication of the precise focus of one's current work. So, we can say that the rule of thumb in ClearCase is: *Check in often, check in soon!*

We mentioned earlier that in order to check out a file, one needs to be allowed to save its data in a directory. If the parent directory is owned by another user, and is not group writable (assuming you belong to the same group), checking out will fail. You need then to specify another directory where to save the data, with an `-out` option. When the time comes to check in your changes, you can use the `-from` option.

Checking in an identical copy of a text file is suspected to be an error (it doesn't make much sense, as it just clobbers the version tree, and gratuitously invalidates derived objects that would have been built using the first version). To force ClearCase to accept this, you must use an `-identical` flag. Note though that this behavior depends on the element type manager: if the type manager does not compute version-to-version **deltas** (contrast `text_file` versions which are stored as deltas—see Chapter 9—to `compressed_file` ones which are not) a new (even identical) version is always created for the element. Most of the time, what makes sense is however to undo the checkout (you changed your mind and actually do not need to modify the file). Then the correct option is obviously to **uncheckout** the element. The fact that this is a distinct command is a minor nuisance, especially when you process several files at a time. David Boyce's wrapper serves this with a `-revert` flag to `checkin`.

There is obviously more to say about checking out and back in, in the context of a branching strategy, but we'll defer it until *Chapter 6*, *Primary Metadata*, after we have taken into account the requirements that may arise from MultiSite.

Versioned directories

Early version control systems only supported text files. It soon became obvious that names are equally important as any other data, and that the pressure to change them, as well as the layout of file systems, made **directory versioning** necessary. The only aspect of directory data which is versioned in ClearCase is the name of entries: the size and the time stamp of files vary freely (they are those of the versions, and thus depend on the selection), the ownership and the access rights are common to the whole elements, and thus non versioned—shared among all the versions.

In order to rename a directory entry (itself a file, a symbolic link, or a directory), to create it, or to remove it, one needs to first check out the parent directory. To move a file, one needs to check out both the source and the target directories. And conversely, the modification is private until the directories are checked in back.

lost+found

At the root of every vob, in the `/main/0` version of it, there is a `lost+found` directory (there is one also at times in the `.s` directory of the view storage: see below).

This is a special directory, which has only a `/main/0` version and cannot be checked out. One may *remove* its name from a new version of the root directory, but this doesn't affect its real existence (as it is always accessible via the version-extended path `<vobtag>/.@@/main/0/lost+found`), so this is therefore a questionable move.

Elements are moved there when they are not referenced from anywhere anymore (they are then called *orphans*).

The simplest scenario is the following: the user checks out a directory, creates a few elements there, and then changes her mind (maybe the intended elements have already been created elsewhere). She unchecks out her directory: the newly created elements are moved to `lost+found`.

In this process, the files get renamed, in order to avoid possible name clashes: a compact form of the element's oid is appended to the original base name:

```
$ ct unco -rm .
cleartool: Warning: Object "a" no longer referenced.
cleartool: Warning: Moving object to vob lost+found directory as ########
                                        "a.90d36f02668949f7b000811fe3624a51".
$ cd lost+found
$ ls -ld a.90d36f02668949f7b000811fe3624a51
drwxrwxrwx 2 joe jgroup 0 Feb 20 18:09 a.90d36f02668949f7b000811fe3624a51
$ ct des -fmt "%On\n" a.90d36f02668949f7b000811fe3624a51@@
90d36f02.668949f7.b000.81:1f:e3:62:4a:51
```

Note that every version has a distinct oid: in order to get the element oid, we had to append `@@` to the name.

Moving unreferenced files to `lost+found` is a safety feature on the behalf of ClearCase: it protects the user from the unexpected consequences of innocuous commands.

Other scenarios are probably less innocuous, as they may include removing versions of directories or even whole directory elements.

There are typically two things one might want to do with files in `lost+found`:

- Restore them, if they were actually valuable. This is easily done with the `mv` (move) command, using the basename and dropping the oid, but provided one knows where to restore them, as the original path information is lost:

  ```
  $ ct co -nc .
  $ ct mv /vob/apps/lost+found/a.90d36f02668949f7b000811fe3624a51 a
  $ ct ci -nc .
  ```

- Remove them completely. This is a good practice, not only to save disk space, but also to make it easier for others to manage their own files in the future.

The tool to perform the cleanup is **rmelem** (remove element). Removing elements destroys information in an irreversible way, and should thus not be done lightly. But precisely for this reason, this tool is safe to use: apart for administrators, only element owners may use it, and only if strict conditions apply (no metadata attached to any version, and all branches created by the owner):

```
$ ct mklabel LABEL foo
Created label "LABEL" on "foo" version "/main/br/1".
$ ct rmelem -f foo
cleartool: Error: Element "foo" has labeled versions
cleartool: Error: You must be the VOB owner or privileged user to #######
                                                    perform this operation.

$ id -un
joe
$ sudo -u adm cleartool mkbranch -nc br2 bar
Created branch "br2" from "bar" version "/main/5".
$ ct rmelem -f bar
cleartool: Error: Element "bar" has branches not created by user
cleartool: Error: You must be the VOB owner or privileged user to #######
                                                    perform this operation.
```

This means that in practice, one can only use it to correct a mistake just after it occurred, and before any other user actually used the element. Therefore, administrators who disable the use of `rmelem` are misguided and do create more problems than they solve.

Note that in order to use rmelem, especially in `lost+found`, you do not need to check out the parent directory; but you do need uncheck out the element (if it happens to be checked out) before it can be removed.

```
$ ct rmelem -f a.90d36f02668949f7b000811fe3624a51
cleartool: Error: Can't delete element with checked out versions.
cleartool: Error: Unable to remove element ##############################
                                    "a.90d36f02668949f7b000811fe3624a51".

$ ct unco -rm a.90d36f02668949f7b000811fe3624a51
$ ct rmelem -f a.90d36f02668949f7b000811fe3624a51
Removed element "a.90d36f02668949f7b000811fe3624a51".
```

In the above scenario, the user should have removed her elements before unchecking out the directory.

Cleaning up `lost+found` of others' files is an administrative procedure, and we'll handle it in Chapter 10. Let's however warn the user of two things:

- One should *not* use rmelem recursively from `lost+found` (on the output of a `find` command): the fact that a directory is not referenced anymore does not guarantee that its contents is not.
- It is normal for the number of cataloged elements to *grow* as a result of removing some directories: their content was not *yet* in `lost+found`, as it *was* referenced; not so anymore after one removed their parent directory.

The view `lost+found` directory, which we already mentioned, is located in the view storage directory, inside the `.s` subdirectory: `view-storage.vws/.s/lost+found`. Unlike the vob `lost+found`, it does not exist by default. It only gets created when user executes `cleartool recoverview` command to make the *stranded* files available explicitly. Stranded view-private files are files, whose VOB pathname is not accessible at the moment (e.g. in case the vob in question was unmounted or removed), or whose directory name inside the vob is not available if e.g. not selected by the view's current config spec. Such stranded files can sometimes become available again, for example, if the vob gets mounted (the former case) or the view config spec changes to select the file's parent directory (the latter case). The user can also decide to move the stranded files explicitly to her view's storage `lost+found` directory by using the recoverview command with `-sync`, `-vob`, or `-dir` options. The moved files are accessible with normal operating system commands.

Removing files

As an executive summary of the previous paragraph, it is important for the user to understand the difference between:

- Removing a name from a directory (`rm` command, short for `rmname`)
- Removing an element (`rmelem` command)

The former is reversible: it boils down to making a different version of a directory. The latter is not (unless by recovering from backup, but this involves restoring the state of the full vob, thus losing events more recent than the destruction of the element).

Restoring a file removed from a directory version does not require any special privilege or extraordinary skills. We'll review below the tools and the procedure, but first a couple of prerequisites: a recap on view extended path and a brief review of the version tree.

Looking at the view extended side of things

If we take a closer look at a particular version of some directory (using the version extended syntax), two interesting observations can be made:

- All the entries seem to be directories (even if the entry name refers to a file element)
- The timestamps and sizes do not directly relate to those in the directory listing inside a view:

```
$ ct ls -d .
.@@/main/mybranch/20 Rule: .../mybranch/LATEST
$ ls -l .
-r--r--r-- 1 joe jgroup 133 Jun 1 2010 file1
drwxrwxrwx 2 vobadm jgroup 0 Oct 31 2009 dir1

$ ls -l .@@/main/mybranch/20
dr-xr-xr-x 1 joe jgroup 0 May 18 2010 file1
drwxrwxrwx 2 vobadm jgroup 0 Jan 15 2009 dir1
```

The `file1` element represented as a directory, is actually the whole version tree of this element referred by its family name, `file1`, and it is the view's job to resolve it to a single actual version. We can navigate explicitly through the whole version tree to get to the same version as selected by the view:

```
$ ct ls file1
file1@@/main/mybranch/6 Rule: .../mybranch/LATEST
```

We would also find out that the element's "strange" date and size shown in its directory version listing are actually the date and size of its initial version, `/main/0` (given for reference purposes only, as indication of the version tree "root"):

```
$ ct ls file1@@/main/0
-r--r--r-- 1 joe jgroup 0 May 18 2010 file1@@/main/0

$ ls -l .@@/main/mybranch/20/file1/main/mybranch/6
-r--r--r-- 1 joe jgroup 133 Jun 1 2010 ###################################
                                  .@@/main/mybranch/20/file1/main/mybranch/6
```

Version tree

One gets a good grasp of an element by displaying its **version tree**:

```
$ ct lsvtree stddefs.mk
stddefs.mk@@/main
stddefs.mk@@/main/0
stddefs.mk@@/main/mg
stddefs.mk@@/main/mg/1 (MG_3.38)
stddefs.mk@@/main/mg/5 (MG_3.39)
stddefs.mk@@/main/mg/6 (MG, MG_3.40)
stddefs.mk@@/main/mg/8
stddefs.mk@@/main/mg/foo
stddefs.mk@@/main/mg/foo/2
stddefs.mk@@/main/mg/p
stddefs.mk@@/main/mg/p/1
stddefs.mk@@/main/mg/p/p1
stddefs.mk@@/main/mg/p/p1/1 (TTT)
```

One recognizes the version extended path syntax, branches, and versions (leaves ending in a number), and as the last decoration in the default view, the list of labels applies to every version. Note that by default, only "interesting" (first, last, labeled, base of branches) versions are shown. One gets the same output for directory objects.

Let's make a note concerning the topological structure of the version tree: there may be many branches, cascaded on top of each others at various depths, but there is only one *main* (even if the type may be changed or renamed, with the risk of surprising users): this is something one cannot change. The main branch serves as basis for all the other branches. Otherwise, the main branch is not in any way more important than the other branches from the user perspective. The ClearCase novices often tend to exaggerate the importance of the main branch and are inclined to think that all of the development must be done in the main branch or at least eventually merged to it. Nothing of that kind! In ClearCase no branch is more equal than others!

This tool also offers a graphical option (`-g`) which, in our experience, is very popular among users; although we suspect it to be responsible for the creation of many evil twins —see later— as it allows the user to bypass, without warnings, the config spec selection. In the spirit of our *Chapter 1*, *Using the command line*, let's however note what (in addition to display space) one loses by using it: line orientation, i.e. the possibility to pipe to other tools, to filter and tailor one's output:

```
$ ct lsvtree -s stddefs.mk | \
perl -nle 'print qq(des -fmt "%Vn %Nc\\n" $_) if m%/[1-9]\d*$%' | \
  cleartool
/main/mg/1
/main/mg/5
/main/mg/6 optimized
/main/mg/8 tools in vobs and build.tag rule
/main/mg/foo/2
/main/mg/p/1 Peter's enhancement
/main/mg/p/p1/1
```

These are the perl-formatted commands for cleartool, instructing it to show only the version and the comment (if any), on the same line, this exclusively for non-0 versions (skipping the branches). Note the use of the powerful `-fmt` option, documented in the `fmt_ccase` man page. The focus is thus on showing only relevant information, in a minimalistic way.

We even actually use an `lsgen` tool, from a `ClearCase::Wrapper::MGi` CPAN module, to show only the last relevant *genealogy* of the version currently selected (that is the versions that have contributed to it), in reverse order from the version tree.

Recovering files

The first step in recovering a file is to find a former directory version in which it was still referenced. Several tools could be used to explore the version tree. We find that for directory elements, `lshistory` is more convenient than `annotate`. One has again to filter the relevant information, and thus first to present it in a suitable format. In the example which drove our previous chapter, we had, at some point, a "definitions" makefile for programs (`stdpgmdefs.mk`), which later became useless and was removed. In what directory version exactly?

To answer this, we'll use a command made of three parts in a pipe:

- Produce the history data
- Normalize the lines of the output
- Filter

The second part is unfortunately complex, especially as a one-liner: it is however boilerplate. The output of `lshistory` for directories in not line oriented: for checkin event, the version is displayed (we actually output first, before the version, a numeric time stamp for sorting purposes, which we drop at print stage), followed with the comment recorded. The latter is a multi-line list of differences between versions. The role of the Perl script is to reproduce the prefix information found on the first line of every event report, to the beginning of every following one. It performs its task in a very typical way for Perl scripts: by creating a *hash*, i.e. a data structure indexed by keys. The keys here are the prefixes (i.e. the time stamp plus version) information, and the data is a list of all the file changes. When the input is exhausted, in an `END` block, the script prints out its data structure, in two nested loops, with the outer one sorted on the time stamps.

```
$ ct lshis -d -fmt "%Nd %Vn:%c\n" . | \
 perl -nle 'next unless $_;
  if(/^(\d{8}\.\d{6} [^:]+):(.*)$/){
    $k=$1;push @{$t{$k}},$2}else{push @{$t{$k}},$_};
  END{for $k(reverse sort keys %t){
    for(@{$t{$k}}){$k=~s/^[\d.]+ (.*)/$1/;print"$k $_"}}}' | \
 grep stdpgmdefs.mk
/main/mg/2 Uncataloged file element "stdpgmdefs.mk".
/main/mg/1 Added file element "stdpgmdefs.mk".
```

The file was thus present in version 1, but removed already in version 2 of the `/main/mg` branch of the current directory.

Next, we check out the version of the directory into which we want to resurrect the file, and use the `ln` tool (link) to duplicate the entry found in version 1. We thus create a **hard link** (by opposition to *soft*, synonymous to *symbolic*, which the same tool would produce if used with an `-s` flag). "Hard link" is the technical term, in the UNIX tradition, for the *name* used to record an entry in a directory object. The same file object may thus be hard linked multiple times in different directories, or different versions of a directory, under the same or different names.

```
$ ct co -nc .
$ ct ln .@@/main/mg/1/stdpgmdefs.mk .
$ ct ci -nc .
```

Giving the directory as a target, as we did, we retain the original name of the file—`stdpgmdefs.mk` in our example here.

We could, in theory, have used the merge tool (we'll come back to it in *Chapter 7*, *Merging*). However, this would have offered us *all* the changes, interactively if we selected so. The number of such changes is variable, and may be large (we could check in advance, and even script the interaction). In the end, merge would however (for directories) create a hard link in exactly the same way.

Hard links

We have already mentioned **hard links** a couple of times. We saw that they may be used to select several versions of the same element (with scoped rules in the config spec), and that they may be used to restore removed files.

The *long* output of the `ls` command in UNIX mentions the number of hard links pointing to the same file (and thus in a vob, to the same element):

```
$ ls -dl bin
drwxr-xr-x 2 admin locgrp 1564 Jan 9 2009 bin
```

In case of directories, the number starts from 2 because of the "`.`" alias, and is incremented by 1 for every sub-directory because of the "`..`" alias there.
These two special cases are handled in a special way, and never result e.g. in labeling errors (attempting to label multiple times).

Until a recent version of ClearCase, there was no direct support for finding the other names of a given element. This is now fixed, with two possible syntaxes:

```
$ ct des -s -aliases -all t1
t1@@/main/3
 Significant pathnames:
 /vob/tools/tls@@/main/1/t1
 /vob/tools/tls@@/main/1/t2


$ ct des -fmt "%[aliases]Ap\n" t1
/vob/tools/tls@@/main/1/t1 /vob/tools/tls@@/main/1/t2
```

Evil twins

This issue of recovering files brings us to the topic of **evil twins**, that is, the consequence of not caring for possible file recovery, while facing for a second time, the need for the existence of a file one removed at some point. First, let's stress that this plague is not an exclusive disease of ClearCase, quite on the contrary. The basic scenario is thus the following (only the barest bones!):

```
$ ct co -nc .
$ echo first > foo
$ ct mkelem -nc -ci foo
$ ct ci -nc .
```

then later:

```
$ ct co -nc .
$ ct rm foo
$ ct ci -nc .
```

and later again:

```
$ ct co -nc .
$ echo second > foo
$ ct mkelem -nc -ci foo
$ ct ci -nc .
```

The problem is that the two instances of `foo` are different elements that accidentally share the same name (in these two versions of the same directory). From a ClearCase point of view, they have nothing in common (as their name is not one of their properties): they have a different oid, a different version tree, a different history... You may recover the first one (under a different path name) and select arbitrary versions of both in the same view (i.e. the same software configuration), without ClearCase being able to inform you of possible inconsistencies. If they do share some commonalities from a semantics point of view (the user's intentions), an opportunity to communicate this to the tool was missed.

This last characterization goes deep into the spirit of SCM (or of the ClearCase supported new generation thereof): one must *use* the tool in order to benefit from it. This doesn't mean that the tool drives and the user obeys, but it tells that one doesn't quite get the power of a Ferrari if one drags it with oxen: a sophisticated tool serves its users in relation to their investment in it.

Something else ought to be understood from this abstract characterization: *evil twins* are not just a matter of the trivial scenario depicted above—this is just a particular case of a more general and deeper issue. Evil twins are not a topic to be handled away with crude (or even clever) *triggers* (more, or may be less, on them in *Chapter 9, Secondary Metadata*). It is utterly irrelevant to the real issue that the twins are *files* bearing the same *name* in the same *directory*: what matters is that the user missed an opportunity to represent a commonality in a way which would elect tool support.

And this is all what SCM is about: make it possible for the user to focus on creative activity, letting the tool take care of reproducing anything already done, earlier or elsewhere.

So... you meet evil twins, what can you do? Unfortunately, in the general case, nothing! Pick the one and hide the other away. But removing it (with `rmelem`) will break any reference to it, and especially any config record having used it. It is thus a matter of convention to decide whether to follow our earlier advice and leave a track of one's own uses, by applying a label: *to read is to write*—as you change the status of what you read, so why not make it explicit? In such a case, you might remove versions and even elements not bearing any *active* label (bound to a config record). Save the data as a new version of the twin retained (in a suitable branch, which sets requirements that we'll address in Chapter 6), and possibly move the *passive* labels (and other metadata) there. Such decisions obviously are subject to trade-offs and are difficult to justify.

Eclipsed files

We mentioned the word **eclipsing** in the paragraph on mkelem. This situation results from a conflict between a view private file and an element bearing the same name. It may happen in different situations, but the most likely scenarios involve changing one's config spec, thus making accessible some elements that were not available previously. In case of such a conflict, the view private file wins, i.e. *eclipses* the vob element.

This may easily be detected with the `ls` command:

```
$ ct ls foo
foo
foo@@ [eclipsed]
```

Of course, `catcr -union -check` would also complain about this file if it was used in a build.

It is simple to delete or rename away such files. Another way to handle them is to use the same method as for *stranded* files (see earlier, in the *lost+found* paragraph), with the following command or a variant thereof:

```
$ ct recoverview -dir $(ct des -fmt "%On\n" .@@) -tag mg
Moved file /views/marc/mg.vws/.s/lost+found/8000043e4c0d3fc3foo
$ rm /views/marc/mg.vws/.s/lost+found/8000043e4c0d3fc3foo
```

Writable copies

Eclipsing files may be an answer to a question we set aside at the beginning of this section: making writable copies of element versions, for tools requiring it. This is arguably a better answer than keeping the versions *checked out unreserved* (which is nevertheless still an option).

The third option which has our preference is to create a branch of the directory, from which to remove the elements and to place the view private copy there. The reason for our preference is the clarity of using a dedicated branch type, allowing for some sharing (not the actual data) among collaborators using the same tool. It also makes it convenient to compare the different copies (although with the checkout unreserved variant, one might use some other view for reference).

Differences and annotations

The analysis of changes we did in the case of directories is of course available for text files as well. We say "text files", but it is actually a matter of element types and their type manager support for the related functionality. As already promised, we'll treat element types in more detail as part of the secondary metadata in Chapter 9. Let's say now that the implementation of basic functionality is shared, so that `text_file` represents in fact more than itself and even more than its own sub-types.

As our readers may guess, we would not recommend the use of the graphical option of the `diff` tool, which results in putting all the responsibility for spotting interesting data onto the user, making it impossible to use any fancy tool to assist her. The default output of `cleartool diff` also surprises us, which presents the differences side by side, thus truncating the lines (under usual circumstances).

We mostly use the `-diff` (but `-serial` goes too) option to retain all the data in a greppable format.

Note that one may of course use external tools, such as `ediff-buffers` under GNU emacs, to compare multiple versions or add colors while retaining the advantages of text mode.

The `-pre/decessor` option might be the single reason to prefer the `cleartool diff` command to external diff tools (so handy that it is even implicit with ***ClearCase::Wrapper***):

```
$ ct diff -diff -pred stddefs.mk@@/main/mg/4
9c9
< LTAG := $(patsubst %so,%.tag,$(LIB))
---
> LTAG := $(patsubst %.so,%.tag,$(LIB))
```

The angle bracket prefixes obviously distinguish the two versions being compared; the line numbers and the type of change are expressed in compact format.

The `annotate` tool is less frequently used (having no counterpart in standard UNIX, although similar functions exist in several version control tools).

Let's admit that its interface is (powerful yet) cumbersome, so that one doesn't want to compute the suitable arguments to make the output greppable every time:

```
CLEARCASE_TAB_SIZE=2 ct annotate -out - -nhe -fmt \
 "%Sd %25.-25Vn %-8.8u,|,%Sd %25.-25Vn %-8.8u" -rm -rmf " D "\
 stddefs.mk | grep LTAGS
2010-05-16 /main/mg/3 marc  D LTAGS := $(patsubst %.so,%.tag,$(LIBS))
2010-05-16 /main/mg/5 marc LTAGS := $(addsuffix build.tag, $(dir ########
                                                          $(LIBS)))
```

We were interested in the definition of the `LTAGS` macro: this tells us that the current definition was introduced in version `/main/mg/5`, by `marc` on May 16.

Grepping is really essential with this tool because of the amount of data it may generate: annotate is in general too verbose to be useful otherwise!

This implies that you:

- Don't want to "elide" prefixes from any line
- Want to get rid of headers

You also typically want to align the content on the same column, after a prefix, and thus:

- Set fixed sized formats, relatively short
- Set them wide enough to be useful, and truncate the versions from the left (align to the right)
- Use a short tab size (2)
- Use a short format to indicate deletions: "D"

We have used this format with very little variation: mostly had to add the `-nco` flag when needed, and to trade the `-rm` one for `-all`, to get a view of other branches.

Misguided critiques

The subversion Red Book presents some surprising rhetoric in favor of the model it supports. It designates as *Lock-Modify-Unlock*, the alternative model, which would thus be, for naive readers, this of ClearCase (among other SCM products). The subversion model is elected to similar abstraction under the label: *Copy-Modify-Merge*. The problems attached, according to subversion authors, to the former model, are fortunately trivially and routinely avoided by ClearCase users: these are problems of the `/main/LATEST` syndrome, that is, serialization problems bound to the use of one single branch (per element).

A ClearCase model naturally aims at avoiding the negative aspect of *locking* (in fact **reserving** in ClearCase terms—locking being reserved for some other usage), that is, modifying the state of the system for the next user. On the other hand, it pays attention to the beneficial aspect of reserving: communicating to others an intention and the focus of an ongoing task.

ClearCase does offer a concept of unreserved checkout, but there is hardly any reason to use it (one should use branches). Anecdotic functionality justified by the mere fact it was cheap to implement. There also is an `unreserve` command to move a checked out version out of the way. But on the second thought, why would one try to undo what somebody else did and then ... to do exactly the same thing oneself? Perhaps a better idea would be to branch off to an own branch (see Chapter 6).

Another trendy commercial in favor of subversion rides the wave of *atomic commit*. This was even so powerful as to force some kind of support to be introduced in a recent version of ClearCase (7.1.1), on *customer demand*! No special support was ever needed in ClearCase to allow for atomic publication: this is the domain of labels. The concept of atomic commit builds on many wrong assumptions: that checkin and publication are necessarily coupled; that checkin always involves a mass of files, and thus a major discontinuity. These abominations should be corrected where they happen, instead of resorting to smoke puffing games.

Summary

This chapter was lighter than the previous one, wasn't it? We showed that ClearCase, despite being a ground-breaking monument in the history of SCM, does also handle neatly the most basic and humble functionalities of:

- Making new elements
- Making new versions of existing ones
- Giving a clear and manageable (lending itself to tailoring) access to the resulting complexity

All this without resorting to Shoot-Yourself-In-The-Foot GUIs and other fads of some misguided competition.

The next chapter is again more important in terms of ClearCase originality. It shows how the requirements set upon the local network, by the build avoidance mechanism, led ClearCase architects to design the precursor of all distributed SCM systems. In many ways, still ahead of its successors.