Professional Expertise Distilled

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

2
Presentation of ClearCase

Soon 20 years after its introduction, ClearCase is still one of the major **Software Configuration Management** tools on the market.

The concept of SCM is however not as clearly defined and agreed upon as some might pretend. ClearCase itself actually challenged any prior definition significantly by introducing novelties to what SCM might mean. These novelties are, as we'll try to show, what makes of ClearCase, still today, an original and valuable product. Instead thus of risking to fix too early the meaning of SCM, we'll offer a historical perspective, and aim at refining toward our conclusion how ClearCase contributes to define a concept both consistent and useful to the wide community of software users. On this path we choose however to consider SCM essentially as a *concern*, which might and should be shared virtually by anybody, not a concept for the exclusive use of specialists.

Our initial presentation will:

- Offer the above mentioned initial historical setup
- Focus upon the features that make ClearCase stand out
- Describe the elements of the implementation which will be referred to in the following chapters

In this book, we'll cover mostly what became known as **base ClearCase**, and leave the UCM extension (Unified Change Management). Justifications for the soundness of this choice will become clearer and clearer through the reading. Let's state in addition that UCM is already well, and exclusively, covered by existing literature.

SCM history

There's one thing history teaches us, and it is that it teaches us nothing. — Author forgotten

We have one duty towards history, and it is to rewrite it. — Oscar Wilde

A brief perspective of SCM history helps to debunk some myths about what to expect from the different tools. It is commonplace to read matrix comparisons, based on checklists of *standard* features. We try to show here that such comparisons are based on unsubstantiated assumptions: there is no "high-level" definition of SCM, which would be applicable to the tools. At least, the most useful functionalities offered by ClearCase cannot be expected on such bases: this is what we'll review next.

CM, *Configuration Management*, was born twice independently, and a third time as a synthesis. It happened first, in the automotive industry, in the 40s, as an accounting discipline. The second birth is this of **version control** or "source code management", and took place in the early steps of the software industry, in the 60s.

The record of this second birth is a bit murky, especially as it spreads over a period of more than ten years, and the people it involves undoubtedly knew and interacted with each other before they published anything. Let a slight bit of straightening help making it clearer, even if it may seem to conflict with some chronological evidence, as we mention it below.

In its early stage, software development was pure creation *from scratch*. This gave an extreme importance to sources: no source, no binary. Of course, this bias, although still strong in oldtimers' mind, doesn't make much sense nowadays anymore: we all get most of software in executable form first, and most often its functionality is the only (OK: the main) thing which matters. Only later, possibly, we become interested in reading the source code, maybe to make a fix or an enhancement.

Comparing text sources was made handy with the `diff` tool. Then the reverse tool was invented: `patch`, which allows you to apply diffs to a similar (but not identical) version of the original text. These tools made it possible to implement cheap (in terms of space consumption) personal backup. Soon again, it was understood that this backup constituted a trace of the recent evolution, which it was valuable to share. Let's admit it: it is not exactly the way it went. For ten years before Larry Wall actually wrote `patch`, people used `ed` line editor scripts, and this is what went into the first version control tools.

The third moment is this of the "official" birth of **SCM** (at least, with this acronym), as the merger of the two traditions, and it happened as early as in the 70s.

We have again to acknowledge that this presentation of history is sometimes received as outrageous, especially by proponents of the first *CM* tradition.

One cause of problems with this history (for example, SCCS was in fact known earlier than diff and patch), is that things became gradually simpler only after the inception of C and UNIX. But this didn't happen at once; several tools had already been developed for years in other languages, on other platforms, and were only later converted to C and ported to UNIX. The important date concerning every one of these tools is thus maybe not when it was created, but when its propagation reached people who made it successful.

ClearCase originality

Two main ideas make the originality of ClearCase, as an SCM tool:

- Presenting the version database via a file system interface (distinct from a sandbox or working copy)
- Auditing build transactions, and recording the system calls for opening file objects, to produce a graph of dependencies, and to base on it an identification mechanism

The two ideas are strictly speaking independent. ClearCase however builds upon their interaction.

Virtual file system

It is not obvious for people with an experience with other tools, and the prejudice of the reference to an abstract concept of *version control* (or source control), to give value to these ClearCase specificities.

Accessing the content of the repository **in-place** (as opposed to in a copy) allows developers to benefit from the tool management *earlier*: during the edition phase itself, and not only after the artifacts have reached a certain status (say, *buildable*, or *tested*), and are thus more stable. It should be quite obvious that the less stable the artifacts are, the more useful the management!

Work performed in a **dynamic view** (the device to exploit this virtual file system abstraction: more on this below) is accessible for inspection by others (on *their* terms), which brings continuity. Developers can access one another's dynamic views explicitly, but the most interesting and important ClearCase feature is the *implicit* artifacts' (so-called **derived objects**) sharing, performed by the tool itself. This is decoupled from deliveries that imply an explicit decision and judgment from their respective authors: a statement about readiness with respect to conventional expectations or general usefulness.

Auditing, winkin

Derived objects are even more variable than source files (because they collect multiple dependencies). The decisive functionality of ClearCase is a mechanism allowing to share these, implicitly, and under the control of the tool. This certainly has a potential for saving time and space (avoiding needless builds, and copies), but the most valuable is that it sets up a foundation for managing complexity. It does it in two ways: first, by reducing the overall amount of separately distinguishable artifacts, and making real differences emerge (that is, raising the signal/noise ratio); then by introducing an objective structure (the dependency graph) into the space of configuration items.

The mechanism at the heart of **clearmake** (the build tool of ClearCase, focus of *Chapter 3*, *Build Auditing and Avoidance*) is based on auditing, and recording as dependencies files accessed during the execution of commands, such as those involved in builds. This happens by intercepting system calls. The records may be used later to decide that an existing artifact matches the requirements for a new one, the production of which may thus be avoided. Clearmake then offers, **winks in**, the old product in place of a needless new duplicate.

ClearCase is not exactly the only system building on these ideas: one can also mention ***Vesta***, and (for winkin) ***Audited Objects***, by David Boyce.

Also, let's note that recent developments of ClearCase do *not* build on those ideas, and thus depart significantly from what makes base ClearCase attractive: snapshot views do *not* most of the time give access to the versioned database (they do it in a restricted way, for example when using `ct find`, which connects to the view server, and fails if it cannot), and do *not* support build auditing. Note the use of `ct` to alias **cleartool**, as advertised in *Chapter 1*, *Using the Command Line*.

One problem is that these features have a price (mostly in terms of requirements on the network), and unless one uses them, this price becomes a penalty, comparing ClearCase to its competitors.

In addition, optimizing winkin involves collaboration from the users, and thus orients their effort. The same may be told of SCM in general (users may easily defeat any kind of manageability by choosing adverse practices: useless duplication, coarse granularity), but sophistication is even more dependent on user commitment. We'll focus in this book on ways to exploit sophistication to its full potential, and over.

The main concepts

ClearCase defines several sub-functions that build up a network of server-client relationships. These functions may be mapped onto different hosts. On every host running a ClearCase functionality, we'll find one *location broker* (`albd_server`) process, which plays the role of a postman. It is always accessible via a *well-known* address, at port 371, and will answer requests concerning the location of processes in charge of the other functions. Note that standalone installations, where all functions are run on a single machine, are possible.

A command line utility, `aldb_list`, is available to check that this essential process is up and running, and, from remote hosts, is reachable.

Its response will be a list of all ClearCase processes known on the host (which may be long). This may often be ignored (by being sent to `/dev/null`): then a one line status summary is sent to `stderr`, which gets displayed (since it is not redirected by the previous).

```
$ /opt/rational/clearcase/etc/utils/albd_list badh >/dev/null
noname: Error: Unknown host 'badh': Host not found
cannot contact albd

$ /opt/rational/clearcase/etc/utils/albd_list goodh >/dev/null
albd_server addr = 100.1.2.48, port= 371
```

Note that it is not enough to guarantee full operability (as other ports are needed for most operations).

We'll now review the main distinctive concepts, and the ways they are implemented:

- Vob
- Dynamic view
- Registry, region, site
- Config spec

Vobs and views

VOB stands for *Versioned Object Base*. Vobs implement in ClearCase the shared repositories for arbitrary data. Views, although nothing prevents one from sharing them, typically support the stateful access of multiple vobs by a single user.

Vobs and views are databases (embedded, object-oriented, not table based). At any time, the databases are kept in the memory of server processes (`vob_server`, `vobrpc_server`, `view_server` and `db_server`), but they are also saved to files, in storage areas. The memory images are read from the files, and the caches flushed back to disk at various points, for example, for backup purposes. In the storages, we find some pools with the main data, the database files, as well as some index files.

The databases are however mostly exposed as **mvfs** (for multi-version) file systems, a technology building upon *nfs* (and adapted to *cifs*/*smb* for Windows). Vobs are thus **mounted**, or **mapped**, while views need to be **started** or **set**. Both are known via their **tags**, which can be thought of as **mount points**. Only, the vob tag is a full mount point (which is a full path) whereas the view tag is only a subdirectory of the view root (`/view` on UNIX and `\\view` on Windows). Then, the mounted file system is actually a combination of both a vob and a view. On UNIX, it is customary (not mandatory) to mirror the view hierarchy and use a common root under which to mount all vobs (typically, `/vob`).

Note that neither this root nor the full vob tags are themselves versioned: they may be considered as common stubs, shared by all paths of all versions. They'd rather be kept very stable, and thus their names should not be too specific. In the example below, for a vob having `/vob/myvob` tag, one can easily rename any subdirectory under `/vob/myvob` with `ct mv` command, but it is not possible (or not recommended) to rename either the vob tag directory `/vob/myvob` itself, or the common vob mount root `/vob`. To be more precise, this is possible, but not *under ClearCase*, and creating a new tag would affect the history as well as the future.

```
$ ct lsvob /vob/myvob
* /vob/myvob /vobstorage/myvob.vbs public
$ mount | egrep 'myvob|vobstorage'
vobhost:/rootdir/vobstg on /vobstorage type nfs
vobhost:/rootdir/vobstg/myvob.vbs on /vob/myvob type mvfs
```

This shows the vob mount point: `/vob/myvob` (which is also the vob tag) of the MVFS filesystem physically stored on the host `vobhost` at `/rootdir/vobstg/myvob.vbs` **vob storage** directory. Note the `*` on column 1 of the output: this tells us the vob is actually *mounted* and not merely registered together with this tag. Be aware of the fact that one cannot "access directly" a vob on its vob storage and for example, check out something from there. The only possible way to access the vob is via a view, as explained below in this chapter.

```
$ ct lsview myview
* myview /viewstorage/joe/myview.vws
$ mount | grep -r "/view |viewstorage"
viewhost:/rootdir/viewstg on /viewstorage type nfs
localhost:/dev/mvfs on /view type mvfs
$ ls -ld /view/myview
drwxr-xr-x 38 joe jgroup 4096 Feb 15 11:16 /view/myview
```

And this shows the view mount point: `myview` (also the view tag), which is a subdirectory of the *root* MVFS filesystem mounted at `/view`; and the actual view storage directory is located on the `viewhost` at `/rootdir/viewstg/joe/myview.vws` directory. When the view is *set* (on UNIX) ClearCase performs a `chroot` system call to set the root directory to `/view/myview`, which explains that all vobs, but also all the contents of the UNIX root are found there. This obviously includes `/view` itself, which needs to be taken into consideration if recursively navigating directories with UNIX `find`. A typical case is this of running the GNU utility `updatedb` from a crontab, to support the `locate` tool. Again, the view is already started: the `view_server` process is running.

A vob is thus a multi-version repository, which stores in addition shared derived objects, and meta-data.

Views have some additional storage for private (non-shared) objects, but otherwise work as a filter on the vobs, presenting to the user and her tools, a consistent selection of the versions available from the vobs. They implement the support for the abstract concept of **software configuration**, and the **exclusion principle** this one satisfies: at most one version of every configuration item. Note that it is typical that one uses in the same time multiple vobs, but one single view. We'll pay more attention below to the mechanism used to express the version selection: the **configuration specification**, or **config spec**. Some operations (examining well-defined objects, like metadata) do not require to mount the vobs or set the view. But one needs to do both operations (`mount`, `setview`) to be able to navigate the vob as a file system, otherwise it shows empty (the normal default state of the vob mount point).

```
$ ct pwv
Working directory view: ** NONE **
Set view: ** NONE **
$ ct lsvob /vob/myvob
/vob/myvob /vobstorage/myvob.vbs public
$ ls -la /vob/myvob
total 10

dr-xr-xr-x 2 root root 1024 May 7 22:31 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..
```

No view is set, myvob is not mounted and the vob mount directory shows empty.

```
$ ct mount /vob/myvob
$ ls -la /vob/myvob
total 10
dr-xr-xr-x 2 root root 1024 May 8 12:11 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..
```

Just mounting the vob without setting the view is not enough.

```
$ ct setview myview
$ ct pwv
Working directory view: ** NONE **
Set view: myview
$ ls -la /vob/myvob
total 18
dr-xr-xr-x 2 vobadm jgroup 1024 Oct 4 12:21 .
drwxr-xr-x 126 root root 4096 May 5 10:53 ..

drwxrwxr-x 3 joe jgroup 24 Aug 24 2009 dir1
drwxrwxr-x 4 smith jgroup 46 Feb 1 2010 dir2
```

Now the vob is mounted and the view is set: the vob content is displayed according to the view configuration.

Note that these examples are UNIX specific: on Windows, one cannot *set* a view (the command is not supported), one has to start it, and (typically) to map it to a drive letter, so that one would have to choose different operations to display related effects. It is however typical that Windows users find vobs unmounted, since starting ClearCase there doesn't mount the public ones, as it does on UNIX.

The bulk of the data is not stored *in* the databases, but in **storage pools**, directory hierarchies hosting various kinds of **containers** (the files with the actual data). The view and vob processes will actually produce and maintain containers matching the requirements of the use context, and hand these to the external tools (editors, compilers, etc), that need thus not be aware of the SCM system. This also ensures that the performance is not significantly affected by using files *from a vob*, since in fact, the files are in the end used from a standard storage. Finally, the processes mentioned above run, and the files are stored, on server hosts: view, vob, or storage servers.

We'll not be back to the containers (source, cleartext, DO), and to the storage in general (protections) until *Chapter 10, Administative Concerns*.

Deeper into views

A first and obvious virtual file system's characteristic is that it offers dynamic access to the repository: there is no need to download/unload the actual data when setting or changing the view configuration, as the dynamic view works as a filter selecting versions according to the view's configuration specification and displaying them under the vob mount point.

```
$ ct setview view1
$ ls -l /vob/myvob
drwxrwxr-x 3 joe jgroup 26 Sep 26 2006 dir1
drwxrwxr-x 3 joe jgroup 49 Sep 26 2006 dir2
drwxrwxr-x 4 joe jgroup 517 Apr 9 20:45 lost+found

$ ct setview view2
$ ls -l /vob/myvob
drwxrwxr-x 3 joe jgroup 24 Jun 17 2008 dir1
drwxrwxr-x 3 joe jgroup 46 Jun 17 2008 dir2
drwxrwxr-x 4 joe jgroup 517 Apr 9 20:45 lost+found
drwxrwxr-x 3 joe jgroup 23 Jun 17 2008 dir3
```

Why does the same vob directory content look different? - Because the configurations of `view1` and `view2` are different (see more on config spec below).

Versioning mechanism

Let's also mention that (dynamic) views support path extension syntaxes to access if need-be arbitrary versions (the ones not selected otherwise by the current view configuration):

- One such syntax (**view extended path**) involves explicitly referring to a view different than the *current* one (the one *set* previously)
- The other (**version extended path**) uses *decorations* to specify the versions of path elements. The use of such extensions is signaled by a `@@`, which enters the space of versions of the element, presented as a directory tree. This `@@` works as a door into the hidden space: it need not be mentioned again, and all the names beyond it are interpreted within the space
- They may be mixed

Let's notice that we already met a few such cases in the *Teaser* chapter.

The view extended path:

```
$ cd /view/joe/vob/perl/lib
```

Here the view tag is `joe`, and the vob tag is `/vob/perl`, and `/vob/perl/lib` is a directory within the vob.

The version extended path:

```
$ ls -ld /vob/perl/.@@/APP/lib/APP
```

Here the vob tag is `/vob/perl`, `/vob/perl.@@/APP` signifies a version of the `/vob/perl` root directory labeled with `APP` label, and it is followed by `lib/APP` denoting version of the lib subdirectory also labeled by the same label `APP`. Note that vob roots display an exceptional behavior in requiring there an explicit mention of the `.` directory, to which the label applies. This is not needed for other directories but the vob root.

And another variant of the version extended path:

```
$ cat /vob/apps/joe/t/Makefile@@/main/jn/1
```

Here the vob tag is `/vob/apps`, and `/main/jn/1` signifies version 1 of the branch `/main/jn` of the `/vob/apps/joe/t/Makefile` element. Technically, `jn` is *cascaded* from the `main` branch. Note that `main` is a predefined branch type, and that there exist one and only one main branch on every element (which may at most be *renamed* or *chtyped* to another name).

Let's mention one extreme, but at times useful, case: the *fully qualified path*. This completely bypasses the config spec, ensuring that the path will works in any context. Note that it requires a view, which must be started:

```
$ ls -ld /view/joe/vob/perl/.@@/APP/lib/APP
```

Of course, the default use of views aims at accessing these same versions as if they were the only ones, as if the **elements** were plain file system objects (files or directories):

```
$ ct setview myview
$ ls -ld /vob/perl/lib
drwxrwxr-x 3 joe jgroup 32 Sep 01 2009 lib

$ ls -l /vob/apps/joe/t/Makefile
-r-xr-xr-x 3 joe jgroup 496 Oct 05 2009 Makefile
```

So, in ClearCase virtual filesystem, a file name is common to a whole family of instances. These instances may be versions of an element, or, probably more interestingly, instances of a derived object. Also, as we have already seen in the examples above, the view-selected version, or actually *any* version of the element can be accessed by *any* standard operating system tools, and not only by ClearCase-specific ones:

```
$ ct setview myview
$ ct ls /vob/utils/hello
/vob/utils/hello@@/main/5 Rule: /main/LATEST
$ perl /vob/utils/hello
Hello, world!
```

We have executed the version of the perl script `hello`, which was selected by the `myview` view. The actual selected version is `/main/5`, which means version 5 on the branch main.

```
$ perl /vob/utils/hello@@/main/br1/2
!dlrow ,olleH
```

And now we have executed another version of the `hello` Perl script, which was not selected by the view by specifying it explicitly: `/main/br1/2`, which is version 2 on the branch `/main/br1`.

The ClearCase versioning mechanism is *transparent* to the end user and to operating system tools.

It is essential (thinking again of derived objects primarily) that this property extends to private data and repository data: the location in the virtual file system gives access to the set via an instance selected by the view, yet any version may be accessed explicitly using an extended notation (introduced with `@@`).

This is how the derived objects are versioned (note the slightly different syntax):

```
$ cd /vob/test
$ ls -l
-rw-r--r-- 1 joe jgroup 0 Apr 11 14:50 all.tests
-r--r--r-- 1 sara jgroup 135 Apr 11 14:50 Makefile
-r-xr-xr-x 1 sara jgroup 381 Apr 10 14:55 pathconv

$ ct lsdo all.tests
--04-11T15:52 "all.tests@@--04-11T15:52.3380"
--04-11T14:50 "all.tests@@--04-11T14:50.3375"
```

This lists all versions of the derived object `all.tests`.

The non-selected versions can be accessed directly using the same `@@` syntax as for the version-controlled elements:

```
$ ct des all.tests@@--04-11T15:52.3380
```

Views properties

Views have some properties: one of them (**text mode**) concerns the presentation of text, and the conversion of end-of-line sequences, that we mentioned in the last chapter. The typical application, in views intended for use with Windows tools, is to add the expected carriage returns to lines of files stored in UNIX mode. This occurs at *cleartext generation* (more on this in Chapter 10), even for read-only access (often from tools, including compilers). One can see that this is superior to the option of explicitly modifying the data at checkin (and checkout). The downside is to restrict the views once and for all to be exclusively used on either platform (otherwise, views may be tagged in two regions: in both Windows and UNIX, and thus shared, with some possible benefits). Obviously, and mostly for symmetry, other patterns of use are possible, including the default transparent mode: no conversion.

The commands to examine and change those properties are:

```
$ ct lsview -l -prop -full <view tag> | grep Text
```

where the output text signifies the following modes:

`Text mode: unix` - transparent (default)

`Text mode: msdos` - insert carriage returns (when presenting to the user the stored text, and remove them when storing)

`Text mode: strip_cr` - strip carriage returns

Note that these view's text mode properties cannot be modified after the view has been created (even with the `ct chview` command).

We'll leave vob creation (a somewhat rare event, even if users are encouraged to practice creating and maintaining private vobs of their own) for later, but let's mention the commands to create and remove views that should be used much more often and by everyone:

```
$ ct mkview -tag joe -stg -auto
$ ct rmview -tag joe
```

The first form implies the existence of pre-defined **view storage locations**, which frees the user from storage issues concerns (and thus disk space, and backup). Such pre-defined view storage locations can be created and listed by the following commands:

```
$ ct mkstg -view ...
$ ct lsstg -view
```

In case there are a few view storage locations defined for the same site, one of them is picked up at the view creation while using the `-stg -auto` option (the algorithm to select one location among others is documented in ***technote 1147041:*** for dynamic views, it supports one way to exclude some locations, favors local ones, and otherwise picks one at random—unfortunately not taking details such as the version of ClearCase on the server into consideration).

The three text modes mentioned above can be specified using the `-tmo` option (in the `ct mkview` command) with one of the following arguments: `transparent`, `insert_cr` or `strip_cr`.

The most important way in which the user affects her views, is however the value of her **umask** at the time of creation (note by the way that views are meant to be created by their actual user, and not by *root*). The umask is in UNIX a bit pattern used to restrict the access rights of files created, from a basic (`777`) pattern: *read*/*write*/*execute* (one bit for each), for *owner*, *group*, and *world*: altogether 9 bits. The value `777` (in octal) represents thus a full pattern: every bit set for every entity, since `7` is `4 + 2 + 1`. The umask value is subtracted from it and the result is used to set the view permissions. In practice, the disputed issue is to decide whether or not to grant write access to group, that is to choose between the values `002` and `022` for the umask: filter nothing (`0`) away from the owner, and *write* from "world" (`2`), but choose to treat one's collaborators along to the former or the latter model:

```
$ umask
022
# The view permissions would be then 755
$ ct mkview -tag test1 -stgloc -auto
...
It has the following rights:
User : joe: rwx
Group: jgroup : r-x
Other: : r-x
```

```
$ umask
002
# The view permissions would be then 775
$ ct mkview -tag test2 -stgloc -auto
$ ct lsview -l -prop test2 | egrep "Owner|Group|Other"
Owner: joe : rwx (all)
Group: jgroup : rwx (all)
Other: : r-x (read)
```

Note that the *execute* right is necessary on directories.

The view write permission allows the user to modify the state of the view, such as by setting the config spec, or by checking files out or in.

The current value of the umask still governs the permissions set to files (view-private).

ClearCase modifies the write access of elements according to whether they are checked in (read-only), or checked out (writable). In a read-only view, checkouts are forbidden. Note that one cannot create a read-only view by setting one's umask to `222`. One has to use the already mentioned `chview` tool.

The issue of the group write permission is met in practice in three cases:

- The one we just mentioned: the view creation. Users are typically afraid that views created with a permissive mask might be shared (usually not a good idea), but this also allows to fix problems during vacations (there is a range of options: checkin, uncheckout, unreserve, remove view, with different restrictions and applying to different contexts, including snapshot views and Windows; see Chapter 10 for more details), or to chase stale file handles in some but not all views.
- For winked in derived objects: DOs produced under a restrictive umask will be read-only for the other members of the group. They may still be winked in, and may still be deleted (depending on the access rights to the parent directory), but they cannot be overwritten. This means that builds will eventually fail (after having first succeeded and winked in some results): clearly a nuisance. This is fortunately easy to solve, by using the CCASE_BLD_UMASK makefile macro, thus overriding the view umask under clearmake.

- For directory elements: directories created with a restrictive umask will not be modifiable by others, who will not be able to create new elements, checkout existing ones without redirecting the data elsewhere, or create derived objects and even simple view-private files. This may seem like a mine field, as the problems are typically not detected at once. One may of course (with appropriate rights) change the protection of elements, but one may meet further issues in presence of MultiSite: protection events are subject to settings in the replica objects, and may be filtered. We'll touch this in due course (see *Chapter 11*, *MultiSite Administration*).

Registry, License, and even Shipping servers

The ClearCase installation is concentrated (apart for the data itself: vobs and views). In UNIX, these are found under two directories: `/opt/rational/clearcase`, and `/var/adm/rational/clearcase`; in Windows, both parts are by default under `\Program Files\Rational\ClearCase`, the latter concentrated as a `var` hierarchy below this root. The former part is considered read-only and not preserved during upgrades (installations). Note that there is also a `common` tree (`/opt/rational/common` in UNIX and `C:\Program Files\Rational\Common` in Windows) with some read-only non-preserved contents.

View and vobs are **registered** in a central location, a directory `/var/adm/rational/clearcase/rgy` on a dedicated server (optionally backed up on another host), in a few flat text files collectively known as the registry. They contain records of the *objects* themselves and of the *tags* used to access them. The realm of a single registry maps to the span of **regions** (also recorded in the registry), and (possibly with subtle differences, rather to be avoided) with this of **sites** (in MultiSite context). The relevant commands are:

```
$ ct hostinfo -l
$ ct lsregion
```

One typical use of regions (see our *Teaser* chapter), is to support Windows and UNIX environments over the same vobs (and even optionally views). For reasons explained in the previous chapter (the different path naming schemes), a vob must be tagged in both in different ways, and regions make this possible. Each client host (even the registry host itself) can only belong to a single region, and can only access vobs and view with tags in this region. This is configured in a `config/rgy_region.conf` text file, under the `/var` hierarchy (not to be confused with the `rgy/regions` file, on the registry host, in which the names of the various regions are stored). But a ClearCase client is allowed to query the tags used in other regions, and to check the identity of a registry object by using its **uuid**: universally unique identifier.

Another function is typically localized on a separate server: holding the license database. ClearCase supports two alternative licensing schemes, functionally intended to be equivalent (*floating* licenses, per simultaneous user): a traditional, proprietary one, based on a simple flat file, and a more recent one, using the *de facto* standard FLEXlm, and supported with tools to provide duplication, backup, and monitoring (these functionalities may be achieved with the former, but only using ad hoc tools).

One last kind of servers (hosts) may occasionally be met: shipping servers. Their function relates to MultiSite, and whereas it is made necessary for crossing firewalls (with restrictions in the opened port ranges), shipping servers may also be used for other purposes (for example, buffering or dispatching between multiple vob servers on a site). See thus Chapter 11.

Config specs

ClearCase offers (or should we say *offered*, from the point of view of the UCM extension, which, as told earlier, we'll ignore for now) a mechanism to select systematically and implicitly consistent sets of versions for multiple elements, across many vobs. This mechanism is called **configuration specification**, and is a property of views. Views are thus considered as the ClearCase implementation of the abstract concept of **software configuration**, already mentioned.

The specification happens through an ordered list of rules evaluated in turn for every accessible element until one matches. Only the first match matters. Note that it is often a different rule which matches in the context of different elements. A rule selects a version (out of many) of an element.

To check which rule actually applied for a given element, use the `ls` command:

```
$ ct ls -d foo
```

The `-d` flag in the above example is useful if `foo` is a directory. What matters, for build avoidance purposes, is the result of the selection, the version selected, and not the rule used (hence, not the config spec). Config specs should be considered merely as convenience tools. However, to remain convenient, they have to be well understood, and thus to be kept simple.

It is a common mistake (sanctified by UCM) to follow the slippery slope of making them too complex, by piling up many ad hoc rules specific to small contexts, and finally to 'free the user from this complexity' by generating them: one has once again thrown the baby with the bath water.

The specification ought to be consistent, and this consistency is best achieved by limiting the number of rules, and thus augmenting their scope: less is more. Then, there is no reason anymore to generate the config specs: the main result achieved by doing this is that the user doesn't feel that consistency is her concern.

A set of simple commands is provided to operate on config specs as text. Here are the man pages:

```
$ ct apropos 'config spec'
catcs Displays the config spec of a view
edcs  Edits the config spec of a view
setcs Sets the config spec of a view
```

In addition to these, the man page documenting the syntax is `config_spec` (`ct man config_spec`).

The most typical example of config specs is:

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

This is called the **default config spec**, and may be restored in a given view, with the command:

```
$ ct setcs -default
```

Beware however:

- The previous config spec (in place before running this command) is lost: ClearCase does nothing to version it, or back it up
- This will actually set the config spec stored on the local host as `/opt/rational/clearcase/default_config_spec` (one might consider it a breach of the principle of least surprise to change it, but it may still be the case...)

Seldom (but see below for one case) will you *not* want to select versions you'd have checked out yourself (what the first line in the default config spec says).

The second line will however on the contrary seldom be sufficient for your needs. It is valid only as a default, ensuring that you'll access at least one version of every element you can reach: there is by construction always a `main` branch in every element (unless you have changed its type, or renamed the main type in the vob—again a questionable move if you intend not to surprise your users), and there will always be a `LATEST` version on this branch, even if it may be version 0, which is an empty one. Note however the more important restriction at the end of the above sentence: "every element *you can reach*". An element is reachable only from a certain path (or more, in case of hard links); it is not uncommon that one of the directories on this path will be selected in a version not containing (yet or anymore) the next name in the path chain: some elements may thus not be reachable at all, in a the current view, using the current config spec!

Let's rather consider the following, more useful, config spec:

```
element * CHECKEDOUT
element * .../branch/LATEST
mkbranch branch
element * LABEL
element * /main/0
```

It anticipates on *Chapter 6, Primary Metadata,* for the concepts of labels (here in UPPERCASE) and branches (lowercase). The first line selects the user's checked out versions.

The second line assumes the user is working in an own branch (`branch`), and her check-ins are always made to a distinct *workspace* in the **version tree**, in which she has write access. This device is meant to support parallel development. By this line in the config spec, the check-outs will also be done from the `branch` in case it already exists for the element.

The third line specifies the beginning of the "branching off" section: that is, for those elements that do not have the branch `branch` yet, it will be created from the element version labeled by the label `LABEL`, as specified on line four. Typically, such a label denotes a code baseline, for example, some stable release, which you'd like to use as a base for the further development.

The last line is mainly for creating new elements: any element has the default version 0 on the main branch (`/main/0`), and it will be branched off to the version 1 on the branch `branch` right away by this rule. The same rule will however also catch the case of elements that were not part of the baseline (that is, not labeled by `LABEL`), ensuring that no discontinuity will arise from taking existing versions, which might introduce conflicts with other files.

As one may see, the main concern is to ensure continuity of work via consistency of the rules: avoid that files might appear or disappear, or that the versions selected might differ in unexpected ways, as a result of one's actions (checking out or in, creating a branch, applying a label to a version currently selected).

```
element * LABEL -nocheckout
```

This last example is one in which own checkouts are excluded (both producing any, and selecting any which would have been produced in the same view before setting this config spec): clearly this is suitable for a read-only access of existing versions, but possibly also for building, hence for creating derived objects, and thus "writing" in this sense. This kind of setup is useful to reproduce a configuration recorded as one single label type; maybe for validating it, making sure that nothing critical was forgotten (such as environment variables, view private files, or files outside the vobs, which would affect the build one would want to replay).

There are several other functionalities available to edit config specs (see the `config_spec` man page):

- **Scope patterns**. We recommended to make rules as general as possible. There may however be some restrictions to the extent of this advice, such as the fact that metadata (and thus label and branch types) is vob local. Some rules may thus make sense only in the context of certain vobs, but not of others. Scope patterns extend (or rather restrict) the `*` character we have used until now. Scope rules based on vob tags pose an interoperability problem: the vob tags are only guaranteed to be valid in the current region. The solution is to use a syntax mentioning the vob family uuid (with the tag as a useful comment):

  ```
  element "[899d17d4.769311d9.9735.00:01:83:10:fe:64=############
                                           /vob/apps]/..." LABEL
  ```

  There is another use: work around the exclusion principle, and allow to select two versions of the same element in the same view. This is clearly an exception, probably temporary, but it is essential to be able to allow it: the user creating a new element instead of a new version (*evil twin*) would even be worse, and cannot easily be prevented. The idiom is to use a hard link and a different name (or path), and to add an exceptional rule for the alternate name.

  ```
  element libfoo.so.26 FOO_26
  element * LABEL
  ```

  This example assumes that the current version of `libfoo.so` is selected by the generic rule with `LABEL`, and preempted in the special case of the `libfoo.so.26` name (a hard link of the `libfoo.so` element).

- **Include rule**. It is possible to share config spec fragments, by including them into one's config spec. This is obviously the mark of a suspicious complexity, but again, may be justified in certain contexts. Note a non-intuitive problem when the included fragments are modified: the changes do not propagate to the user config spec until she re-compiles it, which happens with the command:

  ```
  $ ct setcs -current
  ```

In doubt, one may enquire when was the config spec last compiled with:

```
$ ct lsview -l -prop -full
```

grepping for the `Last config spec update` row, or by looking at the time stamp of the `.compiled_spec` file in the view storage directory.

A second problem may surprise users: the versions of the files actually included are all evaluated using the config spec of the current view, from where one runs the `setcs` or the `setview:` any rules found earlier in the same config spec do not affect this selection! This is documented, simple to understand in theory, but counter-intuitive in practice.

- **Time clauses**. One may try to protect oneself against changes that happened after a certain time stamp, and which would creep in via the `LATEST` builtin pseudo-label type: in effect, to freeze the time at a date in the past. Let's bring your attention to a gotcha under MultiSite: one may learn new things about one's past! This may happen via importing a delayed packet. The technique of using time clauses, which seems careful, is thus not a panacea. We'll come back to MultiSite concerns, and how to deal with them safely in Chapter 5.
- **Block rules**.The last *goody* in the somewhat too rich config spec syntax is an exception to the too simple description with which we opened this paragraph: *block* rules—meta rules that apply onto other rules, bracketed within a block. This may be used for `mkbranch` and `time` clauses, and, let's admit it, rather helps introducing clarity than the other way around.

This review of config specs did not intend to be exhaustive: for this, we refer our reader to the man page. What we do want to stress is that config specs may be written, and even read by users, unless one errs on the side of excessive complexity. The need to version config specs (either by use of the include clause, or by using the `setcs` command to set one's config spec from a file) should be felt as a red flag. The opposite strategy is clearly wiser: design one's config spec so that it may be stable, and remain small and simple. This may be achieved by using **floating labels** (see Chapter 6).

Summary

This concluded our review of the basic concepts:

- The virtual file system metaphor
- Auditing and winkin
- Vobs and views
- Registry, license and other servers
- Config specs

Remember that we also opened up a perspective on thinking of SCM at large, which we'll nourish during our travel through ClearCase.

The scenes are ready? The play may start!