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

13
The Recent Years' Development

This chapter is devoted to a cursory review of topics we did not deal with (at least in any depth) in this book, although they have been the focus of recent developments in ClearCase. We deliberately chose to spend time and effort on aspects left from existing literature, but also to focus on a consistent subset of ClearCase functionality. By this, we do not mean that the most recent developments in ClearCase would have been inconsistent: only that during recent years, ClearCase development has departed significantly, to our dismay, from the original project.

Let's consider this aspect in the (critical) light of the expertise gained so far, and review the following items:

- Triggers
- Snapshot views
- Express builds
- UCM
- Web access, the remote clients, and the Eclipse plugin
- CM API

Historical perspective

Although ClearCase was designed at Apollo, and originally developed at Atria, then PureAtria, its commercial success, especially on the Microsoft Windows platform, took place at Rational, and then IBM.

Most of what we have covered in this book concerns ClearCase as it was designed in the first phase of its development (referring to the above split).

The latter phase, besides taking care of upgrades and ports to new platforms, notably Linux platforms, has mostly addressed a different customer base than the former. The aspects of ClearCase developed more recently were those of an opaque product, aimed at consumers judging it as a black box, instead of as an integral part of their own conceptual toolbox.

The first ClearCase had been promoted as a tool set, not mandating a process. In 1997, the new owners from Rational thought that there was, on the contrary, a need for an out-of-the-box process. This perception matched the prospect of conquering the Windows market, whereas the original success of ClearCase had been gained almost exclusively on UNIX. This was a major cultural change. The user of UNIX command line is in charge of the process, whereas the user of a Windows GUI lets this one drive: choices have then to be made automatically, according to a pre-canned process, followed passively, without passion or any feeling of involvement. Mishaps will be reported, at best with a screen dump, to a black hole service desk, waiting for a magical "solution" in return; and this will happen only if the "problem" survives a reboot.

There may also have been a perception that concurrent products such as *Continuus* (which became *Synergy* and is now an IBM product) were gaining an edge with their radically different strategy, based on process enforcement. Last, Rational also wanted to integrate a full suite of products, starting with ones for bug tracking tickets (there were several, which eventually converged into *ClearQuest*), but also turning around visual modeling, the original area of expertise of the company (after Ada systems for the US Department of Defense).

Finally, there was a customer demand for *higher-level* concepts. This demand, borne in a *business* overtaking of the enterprise, after a long and controversial period of technological creativity, could dangerously well be met with the offering from the competition: components, activities, and tasks, driven from the bug tracking ticket perspective.

Triggers

Triggers were the first mechanism that could be used for process enforcement. They were indeed part of early implementations of ClearCase. We briefly handled them in *Chapter 9*, *Secondary metadata*.

Due to their popularity, triggers kept being expanded: they could be attached to new UCM or MultiSite events. They started being invoked in some remote client operations. They were not powerful enough, though, to implement the intended out-of-the-box process. This came as a disproof of the original claim that ClearCase provided the tools, and that the customers could arbitrarily design their process and use the tools to support it.

More or less following the same path as the UCM developers, we believe that triggers are not the proper technology to implement tailored support for one's own processes. We found wrappers as a more powerful choice, less surprising from the end user point of view, easier to debug and to override or skip in case of unexpected problems. We regret wrappers are not advertised to the same level (or rather, to a higher one) as triggers in the ClearCase documentation. In wrappers, there lies in our opinion, the best potential for Open Source contributions to ClearCase, at least for the time being.

Triggers remain an interesting tool for some prospective fixes (often temporary), especially limited to the context of a single vob. Then it is always a good idea to get the temporary fix (by the trigger) replaced by a more robust solution (for example, a wrapper). At this point, the temporary trigger can be removed. Let's note that the best way to disable a trigger is usually to `lock` it `-obsolete`. Add the comment first, and lock next, as the `-obsolete` option ignores any simultaneous comment. This allows us to document (in the comment field) the precise reason for the experimented inadequacy so that the same "fix" is not attempted again.

Snapshot views

We mentioned snapshot views very briefly in *Chapter 10*, *Administrative Concerns* and *Chapter 12*, *Challenges*. Snapshot views were introduced with the need to support work in a temporarily disconnected environment (typically on a laptop). They were not a great conceptual creation, as they merely implemented the traditional model of *sandboxes*, common to the older generation version control systems, still present under the name of *working copies*, in, for example, *subversion*.

Snapshot views still require occasional connectivity, to `load` and `update` them. They are registered in the ClearCase registry, and their state is published, as much as it concerns checkouts (reserved or not). Because ClearCase doesn't support a concept of private branches (such as in *Mercurial*), working in disconnected mode may lead to conflict situations (*hijacking*) to be resolved eventually, when the connectivity is restored.

In a disconnected snapshot view, one may edit elements that one had previously checked out, *hijack* some more files, and build using one's private files. Problems with proper ClearCase operations start with the lack of licenses, and even earlier with the need for authentication from network servers:

```
$ ct ls foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.

$ ct catcs
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.
```

Of course, one may wonder whether these operations make sense in the context of a disconnected snapshot view, in which the versions have already been selected and the choice may not be revised until next time the connectivity is restored. A snapshot is after all no more than a dead copy.

The hijacking mechanism is actually the accepted way to work around this state of things in the disconnected mode, with the promise to restore consistency later when connected. Suppose one is working in disconnected mode; then one can only change the permissions (remove the read-only flag in Windows) on a downloaded version in the snapshot view. ClearCase commands are of course still unavailable:

```
$ ct co -nc foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.

$ chmod 755 foo
$ echo foo >foo

$ ct ci -nc foo
cleartool: Error: Unable to contact albd_server on host 'ccserver'
cleartool: Error: Cannot contact license server host "ccserver"
cleartool: Error: You do not have a license to run ClearCase.
```

When connection is again established, one can spot the hijacked version:

```
$ ct update .
Processing dir "bar\www".
Processing dir "bar\www\cc".
.......
End dir "bar\www\cc".
.
Keeping hijacked object "bar\www\foo" - base "\main\4".
...........
```

```
End dir "bar\www".
.
Done loading "\bar\www" (28 objects, copied 0 KB).

$ ct ls foo
foo@@\main\4 [hijacked] Rule: \main\LATEST [-mkbranch b1]
```

The discrepancy may be resolved either by checking in or by removing the offending private file.

First, the later option (clean up the hijacking file):

```
$ rm foo
$ ct update foo
Loading "bar\www\foo" (1654 bytes).
.
Done loading "\bar\www\foo"

$ ct ls foo
foo@@\main\4 Rule: \main\LATEST [-mkbranch b1]
```

Alternatively, the former option, check it in (checking out first to a separate branch to avoid a possible conflict with a checkout by another user):

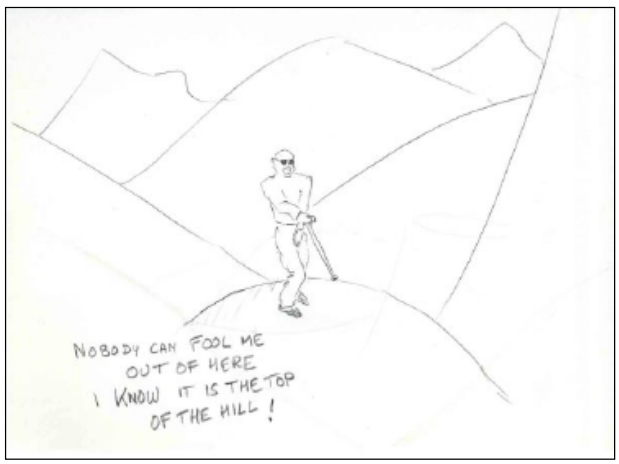
```
$ ct co -nc foo
Created branch "b1" from "foo" version "\main\4".
Checked out "foo" from version "\main\b1\0".
"foo" has been hijacked.
Do you want to use it as the checked out file?
(If not, it will be renamed.) [yes]

$ ct ls foo
foo@@\main\b1\CHECKEDOUT from \main\b1\0 Rule: CHECKEDOUT

$ ct ci -nc foo
```

Snapshot views present a real management challenge: the snapshot view directory is not recorded in the view storage, and may be virtually anywhere on the user laptop (or whatever accessible storage). This makes it impossible to service by administrators (backup, upgrade, and so on) even when the laptop is connected. This is especially true in the case of hijacked versions.

Building in snapshot views, clearmake behavior falls back to the traditional make model, using only local resources from standard file systems and producing view-private artifacts.

On a low level, this is more efficient than building in dynamic views in an MVFS file system: process communications are greatly simplified (only local), and neither auditing nor shopping for derived objects, with its sophisticated evaluation and decision process, takes place.

This appealed to many users, especially in cases when these didn't, for one reason or another, use or benefit from *clearmake winkin*. From our point of view, such users are effectively trapped, with no easy way out of their hole.



Express builds

The low-level performance gain obtained in snapshot views can be emulated in dynamic views as well, thus retaining the comfort of synchronous updates.
A first way is to use the `-v` flag of clearmake, restricting configuration lookup to the current view and thus disabling winkin. The derived objects produced are still recorded, with an incurred cost.

The next step is thus to create views with a special `-nsh` flag, instructing them to produce non-shareable derived objects. This way, clearmake is effectively downgraded to a standard make tool.

UCM

UCM was an attempt to raise the level of support, in other words to support "higher-level" concepts directly: components, as a way to structure sets of files; activities and tasks, as a way to attach some meaning to change sets; streams, as a way to relate and transcend labels and branches.

Unfortunately, it didn't build upon the existing sophisticated features of ClearCase (dynamic views, clearmake winkin, and derived object management), and resulted in lowering the build support.

It also forced users to make decisions upfront, and made it from hard to nearly impossible to revise them later.

It finally capitalized on a branching strategy, which was state-of-the-art at the time, but proved counter-productive in retrospect: one dedicated integration stream (and integration branch type), where all the changes should be *merged* to (*delivery*) from a number of development streams (branches). The latter would be kept in sync (again, by merging) with the integration stream (*rebase*). Streams did bring a welcome improvement over plain old labels, with a way to support *incremental* labeling, a functionality that we had unsuccessfully requested in base ClearCase and finally had to match with *label type families* in our ***ClearCase::Wrapper::MGi*** wrapper.

UCM claims to implement SCM *best practices*, and to raise the level of abstraction, so that the end user would not need to deal with low-level SCM tool concepts such as individual elements, and could access a representation of the whole software architecture.

Let's take a closer look at what these best practices are and how they are implemented.

Boot-strapping the simplest UCM project, be it only for one single user at the start, already involves quite a few steps. One must define and create at least the following minimal set of UCM artifacts: a project vob, a component within this project vob, an initial baseline for the component, a UCM project itself, an integration stream for the UCM project, an integration view on the integration stream, and an activity. Usually at least one development stream with a *separate* development view is created as well.

After creating and configuring all of the above, and getting all this finally to work, one can assume that the basic "out-of-box process" skeleton is in place. Now, changing one's configuration back and forth should be a piece of cake, so to speak. It turns out not to be quite so. One can happily make a code change in the development stream, deliver it to the integration stream, and create a baseline there. This one can also be *recommended* in the integration stream, and the development stream can be rebased with it. Usually at this point in the UCM literature the reader is promised an ocean of future possibilities such as compare baselines, merge baselines, create a child stream from the baseline you want, rebase the child stream with the recommended baseline, or even find the changes between baselines and undo/modify/redo them. But simple things first. Let us not be distracted from paying attention to one obvious thing: how easy it is to roll back, that is, what happens if you change your mind and would like to get back to the previous (initial in our case) baseline? Let's see: you seem to be able to recommend the initial baseline back in

the integration stream instead of the newly created one. So far so good! But when you *synchronize* your integration view with the stream (which is surprisingly needed even for dynamic views!), your view still selects the latest changes you made, which are not part of the initial baseline. It's time to check the integration view config spec (even this is discouraged in UCM: the only changes allowed to config spec are vob load rules, which don't make any sense for dynamic views of course):

```
$ ct setview intview
$ ct setcs -stream # Synchronize with the stream
$ ct catcs
ucm
identity UCM.Stream oid:a6cffe7b.6980496a.aaab.11:56:4d:2c:a4: ##########
                    b5@vobuuid:5af58e42.640210d5.b8a4.00:c0:61:20:6a:53 1

# ONLY EDIT THIS CONFIG SPEC IN THE INDICATED "CUSTOM" AREAS
#
# This config spec was automatically generated by the UCM stream
# "PROJ_Integration" at 2010-10-15T10:19:42+01:00.
#
# Select checked out versions
element * CHECKEDOUT

# Component selection rules...
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                                .../PROJ_Integration/LATEST
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                       INITIAL -mkbranch PROJ_Integration
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                       /main/0 -mkbranch PROJ_Integration

end ucm

#UCMCustomElemBegin - DO NOT REMOVE - ADD CUSTOM ELEMENT RULES ##########
                                                        AFTER THIS LINE
#UCMCustomElemEnd - DO NOT REMOVE - END CUSTOM ELEMENT RULES

# Non-included component backstop rule: no checkouts
element * /main/0 -ucm -nocheckout

#UCMCustomLoadBegin - DO NOT REMOVE - ADD CUSTOM LOAD RULES #############
                                                        AFTER THIS LINE
```

Our project's name is `PROJ`. The initial baseline of the component `comp1` was called `INITIAL` (and was based on the set of versions labeled with the label type `INITIAL`). The new baseline we just created has the name `BASELINE1`, which is actually a ClearCase label type `BASELINE1` attached to the newly created set of elements. But ...it doesn't appear in the config spec of the integration stream! And actually no matter how many baselines one creates or recommends in the integration stream,

none of them will ever be mentioned explicitly in the integration stream config spec: the integration stream is always `. . ./PROJ_Integration/LATEST`; that is, no matter how you configure your project, you will always see only the latest versions in the integration stream and this cannot be configured differently! Hardly a best practice, is it? By recommending the baseline one only records the current label, which is a good idea in itself, but the record does not go any further, as far as the integration view is concerned! This also means that if one chooses to work in a "minimal" UCM project having just one integration stream, and no development streams, there is no way to manage one's configuration at all.

Now it should be a right moment to recall that IBM Rational recommends not to modify or even look at UCM config specs ("because you don't need to")—we may now understand better why...

So, the UCM integration stream *always* points to the latest versions. If you want even to take a look at a previous baseline, not to mention working on it, you need to do it in a separate child (development) stream.

So, how about the development stream? First we rebase it:

```
$ ct setview devview
$ ct rebase -baseline BASELINE1
   Advancing to baseline "BASELINE1" of component "comp1"
   Updating rebase view's config spec...
   Creating integration activity...
   Setting integration activity...
   Merging files...
   Checked out "/vob/bar/comp1/foo" from version ########################
                                              "/main/PROJ_development/4".
   Attached activity:
    activity:rebase.PROJ_development.20101112.115451@/vob/ ###############
            pvob  "rebase    PROJ_development on 11/12/2011 11:54:51 AM."
   Needs Merge "/vob/bar/comp1/foo" [to /main/PROJ_development/ #########
CHECKEDOUT from   /main/PROJ_Integration/3 base /main/PROJ_development/2]
   ******************************
   <<< file 1: /vob/bar/comp1/foo@@/main/PROJ_development/2
   >>> file 2: /vob/bar/comp1/foo@@/main/PROJ_Integration/3
   >>> file 3: /vob/bar/comp1/foo
…
$ ct rebase -commit
```

The new baseline is visible in its config spec, so this is already something to start with:

```
$ ct setview devview
$ ct setcs -stream
$ ct catcs
ucm
identity UCM.Stream oid:595d3b4a.bb0943f1.23d4.15:ac:01:e8:73: ##########
                    52@vobuuid:5af58e42.640210d5.b8a4.00:c0:61:20:6a:53 3

# ONLY EDIT THIS CONFIG SPEC IN THE INDICATED "CUSTOM" AREAS
#
# This config spec was automatically generated by the UCM stream
# "PROJ_development" at 2010-10-15T13:57:05+01:00.
#
# Select checked out versions
element * CHECKEDOUT

# Component selection rules...

element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                                .../PROJ_development/LATEST
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                      BASELINE1 -mkbranch PROJ_development
element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." #########
                                       /main/0 -mkbranch PROJ_development

end ucm

#UCMCustomElemBegin - DO NOT REMOVE - ADD CUSTOM ELEMENT RULES ##########
                                                      AFTER THIS LINE
#UCMCustomElemEnd - DO NOT REMOVE - END CUSTOM ELEMENT RULES

# Non-included component backstop rule: no checkouts
element * /main/0 -ucm -nocheckout

#UCMCustomLoadBegin - DO NOT REMOVE - ADD CUSTOM LOAD RULES #############
                                                        AFTER THIS LINE
```

Okay, this looks alright. But then can we change it back to the initial baseline? The answer is NO. See what happens if we try.
We recommend back the initial baseline for the integration stream of our `PROJ` project. And then trying the rebase:

```
$ ct pwv
Working directory view: devview
Set view: devview

$ ct rebase -recommended
```

```
cleartool: Error: Can't revert to earlier baseline "INITIAL" ############
                  of component "comp1" because the stream has made changes
                  based on the current baseline.
cleartool: Error: Unable to rebase stream "PROJ_development".
```

Why should this be impossible? This is because there is a hidden trick in the above mentioned development stream config spec: what UCM actually does at rebase (and similarly at the deliver as well) is physically merging the versions carrying the label `BASELINE1` to *all* the versions already having a `PROJ_development` branch. That means the `element ... BASELINE1` rule in the config spec (`element "[2ba58ea2640201c6a8a300d048306e57=/vob/bar]/comp1/..." BASELINE1 -mkbranch PROJ_development`) comes *too late*: only in second. It only applies to versions not having a `PROJ_development` branch yet, that is, that have never been checked out in the views on this development stream. That is why reverting to a previous baseline `INITIAL` is simply not possible, as it is not possible to "un-merge" the latest versions on the `PROJ_development` branch from the ones carrying `BASELINE1` label. Without that harmful merge, the revert would be just a matter of replacing the `BASELINE1` with `INITIAL` in the development view config spec.

The only possibility to make use of the former baseline is to create a *new* development stream (`PROJ_dev1`) from the integration one using the needed baseline (`INITIAL`) as its foundation (that is to branch off a new branch from the integration one using the `INITIAL` label).

Likewise, the UCM components must be read-only in a project in order to be able to change their initial baseline later on. Otherwise, in case of any changes (even the dummy ones) made to a component, rebase becomes forbidden. Furthermore, removing a component from a UCM project stream would typically not be possible either:

We must admit such practices are quite commonplace indeed, but can they still qualify as *best*?

It already seems that UCM makes its motto from: "no way back", and we'll be back on this.

But the UCM delivery by merge model is even more questionable. This is especially felt in large projects, involving many developers.

Any element version change in UCM has to be reported to an *activity*. Generally speaking, having activities as a means of tracking one's own (or others') changes is not a bad idea in itself. Developers can find it quite useful for their own bookkeeping purposes.

Performing a delivery to the integration stream takes place in the scope of whole activities: their *change sets* are being delivered (merged) to the integration stream. This delivering by merge occasionally brings surprises, as we explained in *Chapter 7*, *Merging*: presumably fixed bugs may re-appear either in the integration or in the rebased development streams. Non-trivial merges hidden among massive amounts of trivial ones may yield, on either the delivery or the rebase way, what appear as random results. Bulky merges from multiple development streams may interfere with each other, and a bug fix delivered from one development stream may unintentionally get overwritten so that the original bug is restored.

Preventing non-trivial merges becomes the goal of policies such as forcing a rebase prior to delivery. This is however a two-edged sword, as it affects the data being delivered that ought to be re-tested. This in turn extends the overall delivery time, thus increasing the risk of collisions, which will force rebases, *ad nauseam*.

It is sometimes recommended to lock the shared integration stream while building in it to validate the result of one's delivery before *completing* it; this to ensure a tight serialization of deliveries, prevent activity changes, and avoid to find numerous elements checked out as an effect of other pending deliveries. Both alternatives of either locking or leaving it open to changes have significant drawbacks.

Do we need to remind our readers that these problems affect radically less *in-place deliveries* (refer to *Chapter 6*, *Primary Metadata*), based on labeling, which are faster, thus offer less occasions for collisions, are reversible, hence offer the option to solve the problems out of the way of competitors, and do not modify the data therefore do not require a new testing phase as part of the delivery process proper? The scalability problems we describe here, growing faster than the size of the system by any measurement, are typical of UCM, not a fatality of parallel development in ClearCase! One should remember not to mix the actual parallel development concept and UCM or UCM-like ways of using base ClearCase.

The intention behind UCM activities was interesting, but unfortunately ClearQuest integration largely impacted their usefulness. ClearQuest imposes a number of additional restrictions to the already tight UCM environment. Typically, the ClearQuest-UCM integration is used to impose a number of policies requiring that all the activities are created beforehand in ClearQuest by the designated roles (such as *project manager*) and preferably assigned to the developers: in the best case a developer could try herself to select one out of a set of activities created in advance. Additional restrictions can apply to activities delivery, status change, and so on. This

is clearly an expensive (and very heavy) overkill. It tends to distract developers, who end up finding workarounds and developing their code in their own "sandboxes", outside of UCM and ClearQuest altogether.

Let's now take a brief look at a few typical UCM problems or rather use cases.

Even apart from merge-related issues, and whether with ClearQuest integration or without it, delivering a dedicated subset of one's activities can be quite challenging. Although one can reassign the modified versions from the change set of one activity to another, this is often not enough to eliminate some *dependencies* (for example, one of the activities contains a change to a version of the vob root directory) between the activities, so one is often forced to make a bulky delivery including *all* the activities:

```
$ ct deliver -act act1
cleartool: Error: Activity "act2" must be added to activity list.
cleartool: Error: Version "/vob/foo@@/main/PROJ_development/3" ##########
      from the activity "act2" is missing from the required version list.
cleartool: Error: The list of activities specified is incomplete.
cleartool: Error: Unable to deliver selected activities.
cleartool: Error: Unable to deliver stream "PROJ_development".
```

For delivering a set of activities, as nearly for everything in UCM, the recommendation is to use the (slow...) GUI. When the project grows, the usability of GUI operations, such as delivery, suffers, and one starts to meet all kinds of strange errors, like shown in the next screenshot:

The command line delivery command may help in this situation:

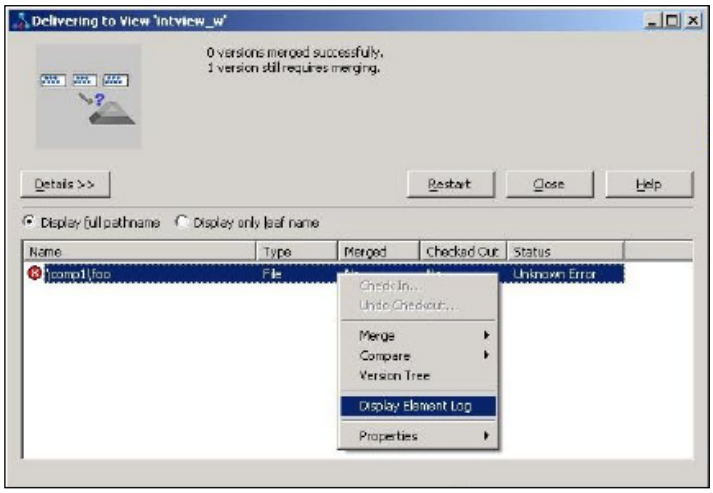
```
$ ct deliver
Changes to be DELIVERED to default target stream in project "PROJ":
 FROM: stream "PROJ_development"
 TO: stream "PROJ_Integration"
Using target view: "intview".
Activities included in this operation:
  activity:act1@/vob/bar joe "act1"
```

Every UCM stream requires a separate view per user involved. Several UCM projects co-exist usually in parallel (for example, a main release project, a maintenance project, and so on), and every project can have many streams (some of them for the purpose of branching off a former baseline as explained above), so the number of views grows very soon. The view names tend to be very long (for example, `userid_ucmproject_int`, `userid_ucmproject_dev`) to allow distinguishing between one another's views and even between one's own views, which also becomes a source of confusion and errors. Especially in the absence of clear transcripts and logs, due to the encouraged use of GUIs, even simple situations can become puzzling for not very experienced users.

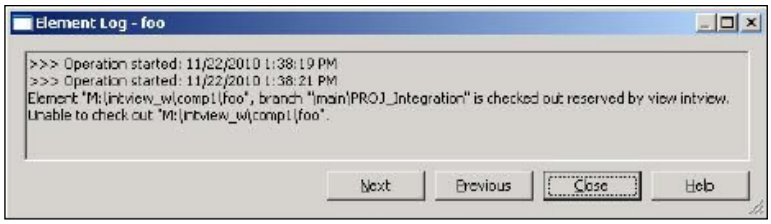
Let's give an example scenario. A user tries to *deliver* (merge) her *activities* from her *development stream* using her *development view* to the project's integration stream. She gets an error message saying that the element `foo`, which she attempts to merge, is in a checked-out state. Actually, even getting this information is not obvious; as with the *ClearCase Project Explorer* GUI, it is hidden behind the red icon next to the element name in the *Merge Tool*, and the *element log* (actually the GUI pop-ups below are Windows UCM client-specific, as the similar UNIX interface seems to be somewhat less messy):

There are not many options except for clicking **OK** on this not too descriptive pop-up, which results in still one more similar one:

After this a third pop-up window follows, with at least some details this time, but one must know to right-click on the item having the red icon on the left and choose **Display Element Log** to get any actual information about the problem:



And finally one is rewarded with the actual problem description (note that log reading abilities are required though):



And if using Eclipse with the *ClearCase plugin*, the message she gets asks her to do a "resource restoration" ("resource" is the Eclipse term for ClearCase *element* and "restoration" presumably stands for *checkin*). But trying to "restore resource", by *updating* her development view, doesn't help, as she cannot see any problem with her `foo` element and it does not seem to be checked out. The user might conclude that there are some synchronization problems between the repository and her view. The actual problem is that the element is being checked out in another view, for example, in one of the integration views (hers or somebody else's). As the integration stream always uses the `.../Integration_branch/LATEST` model, as we explained above, such an element left in a checked-out state can prevent users from making deliveries. This is just an illustration of a very simple problem. And such minor issues turn out to be very tricky and time-consuming to solve because of the artificial and unmanageable complexity of the UCM environment.

The use of integration branches, instead of main ones, suggests a MultiSite setup: the advantage is to allow the coexistence of several integration streams, whereas there can only be one single *main* branch at the root of any element. Integration streams are bound to projects, so that in order to have two, one needs to create two different projects, based on the same components, enabling inter-project delivery policies: each project performs inter-project delivery of the other project's baseline to its own integration stream. This helps avoid the unmanageable *posted* deliveries (the recommended practice under MultiSite), one other major woe of UCM. Another option could be to have a project with a dummy integration stream and two child (development) streams, each mastered at its own site, and to perform inter-stream deliveries from each other's baseline to one's own stream.

As we already mentioned in Chapter 9, the standard ClearCase *clearimport*, *clearexport*, and *cleartool relocate* utilities do not work with UCM vobs (*clearfsimport* does work).

A last note, well known as it is: one can easily convert one's vobs to UCM, but there is no way back. There is no documented way to convert a UCM vob to base ClearCase.

If the vob has originally been converted to a UCM component from a base ClearCase vob, one can make it a base ClearCase vob back, by explicitly removing the hyperlink to the Admin UCM vob:

```
$ ct des -l vob:/vob/comp1
versioned object base "/vob/comp1"
...
 Hyperlinks:
  AdminVOB@738@/vob/comp1 -> vob:/vob/pvob

$ cleartool rmhlink AdminVOB@738@/vob/comp1
cleartool: Warning: An AdminVOB hyperlink to a UCM PVOB is being removed.
This can cause serious problems with UCM.
If desired, this hyperlink may be replaced using the command:
 cleartool mkhlink AdminVOB vob:/vob/comp1 vob:/vob/pvob
Removed hyperlink "AdminVOB@738@/vob/comp1".
```

Note that the UCM branches and versions are preserved in the vob (and can be accessed, for example, with version-extended path names), so a manual conversion is possible:

```
$ ct lstype -kind brtype
--11-22T12:30 joe branch type "main"
 "Predefined branch type used to represent the main branch of elements."
--11-22T12:53 joe branch type "PROJ_development"
--11-22T12:39 joe branch type "PROJ_Integration"
```

```
$ ll .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ ##############
                                                              PROJ_development
total 6
-r--r--r-- 1 joe jgroup 0 Nov 22 12:58 0
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 1
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 LATEST

$ ll .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ ##############
                                                            PROJ_development/1
-r--r--r-- 1 joe jgroup 4 Nov 22 12:59 .@@/main/PROJ_Integration/1/ #####
                               foo/main/PROJ_Integration/PROJ_development/1

$ cat .@@/main/PROJ_Integration/1/foo/main/PROJ_Integration/ #############
                                                            PROJ_development/1
foo
```

We tried here to show with concrete examples how UCM submits developers to additional constraints, hence increases the overall complexity of the development environment, and reduces developers to a passive role. We are deeply aware of the fact that we shall have failed to convince aficionados (and there are many). These might retort either with further escalation (creating yet another stream as an example) or blaming decisions already taken (the processes for ClearQuest integration already in place). What we face here is a case of non-*falsifiability*. The net result of this failure to achieve undisputable conclusions will in any case be that developers will leave the field to experts: the tool is not *their* tool, it is imposed on them and felt as an external requirement they have to satisfy.

UCM has systematically been presented as "enhanced" ClearCase. This is a myth, which may only hold superficially. UCM not only ignores the main assets of ClearCase but it undermines them. The clearest case is maybe how a delivery by merge forces a full rebuild by failing to promote the derived objects produced in the development branches. This could be addressed in either of two ways: by performing the rebuild as part of the delivery before the baseline is updated, or by letting everybody compete to build. The former would make the delivery longer yet, and is thus not chosen in practice. The latter results in race conditions and thus the production of equivalent derived objects, which pollute the DO pool and jeopardize winkin at large. This is again only an example. Other examples could be the impossibility to exploit labels produced with `mklabel -config` or the general inability to handle the config specs of UCM views. The bottom line is clear: UCM is deeply incompatible with an effective use of base ClearCase, as the one we have tried to present throughout this book.

Web access and remote clients

Internet drives and federates the best news software brought to us during the past 10 years. Hence it's a must to provide web interfaces, that is, to allow accessing ClearCase from the new universal tool: the browser. This does of course bring some real benefits: making ClearCase accessible from mobile phones (although the smarter ones provide a decent terminal interface), tunneling access through one single port (ouch), and allowing to support a centralized model, including over WAN connections.

The two latter arguments are actually good news only from an administrator's point of view, especially with the mythical, but so easy to sell to management, 'security' concern in mind, but maybe also with the similar *cost saving* one (that is, from *absolute* points of view, which do not have to take anything else into consideration, such as securing what? Or what does it cost to save?)

Anyway, there's already been several waves of web interfaces to ClearCase, and more are expected. This is actually one of the most active development areas. ClearCase comes with a *Rational Web Platform* (the ClearCase Web server, under `/var/adm/rational/clearcase/ccweb` directory), which allows for creating *web views*: a special kind of snapshot view with its storage located on the dedicated web server, and the view root directory on the local client workstation.

The web server must run a *ClearCase client*, and be connected to the license, registry, and vob servers.

The user must just have an account on the web server: no ClearCase client installation or connection to the vob/registry server is needed.

The original interface (so called ClearCase Web Interface), now obsolete, worked in a standard browser.

A *Rational ClearCase Remote Client* (CCRC) is also available, accessing the web server, but offering more sophisticated functionality and running locally on the user workstation. A first version was a *native* client, now replaced by one based on Eclipse technology. It may use web views.

The ClearCase Web interface stopped being supported before it was taken away. Where one can still access it, one may expect to get all sorts of Java exceptions, such as `Error: "java.lang.IllegalStateException: Timer already cancelled`, maybe related to the version of Java used on the browser. This interface always was extremely limited, even comparing to CCRC. CCRC has been the major focus of attention lately. In the latest releases even some initial command line interface has been implemented (it had always been just a GUI).

CM API

We didn't mention the CM API, which came out with v7.1, as we have no experience of using it.

It is not the first attempt at offering an API to extend ClearCase. The very first one was a C API. Then came the ***ClearCase::CtCmd*** Perl package, distributed on CPAN and intended to be linked with the ClearCase shared libraries; next the *Rational ClearCase Automation Library (CAL)*, exclusively on Windows.

Let's note that neither CtCmd nor CAL were compatible with *Perl/ccperl*, the Perl bundled with ClearCase installation respectively on UNIX/Windows (actually, the version of ccperl distributed with v7.x has the `Win32::OLE` required to access *CAL*, as has *Common/ratlperl*).

ClearQuest came with its own cqperl, different from ccperl. Both tools now use ratlperl, with cqperl and ccperl being kept only for backwards compatibility. None of these attempts at offering an API to support customer extensions and customization were wholly satisfactory.

The CM API is a Java API, and it is common to ClearCase and ClearQuest. We understand it is bound to the CCRC (thus still limited in functionality), and intended to supersede the use of cleartool (frightening to us as this may sound!). We do not feel compelled (yet?) to jump on this bandwagon.

We further skip products farther away yet from the ClearCase "umbrella"—Build Forge (the relatively recently acquired *Continuous Integration* offering of IBM) and Rational Team Concert. To us, both products show the disaffection of ClearCase concepts, and replace noticeable parts of its architecture with new tools. For instance, Build Forge does address some of the missing functionalities in clearmake support, for Java and web development. It does it in ways incompatible with the specific solutions in clearmake.

Summary

In this chapter, we didn't so much give our readers assistance to make use of the features of ClearCase we dealt with, but rather, as it happens, reasons to avoid them. The bulk of the new functionalities are centered around the UCM extension, which we consider ...catastrophic.

Working with UCM doesn't mean UCM working for you: it is you who are working for UCM.

Our advice is bluntly simple: do not use UCM.

ClearCase was, and 20 years after its conception, still is a revolutionary product. Being revolutionary, it took the risk of showing a different path: putting SCM upside down and making information emerge from relationships audited at build time, instead of flow down from intentions expressed at design time. In its attempt to conquer the wide hence profitable market of PC-based software development, Rational, and then IBM, refocused the ClearCase product away from its avant-garde concerns back into the traditional mainstream. It is not that the problems that ClearCase was meant to tackle would have been solved, quite on the contrary: the mainstream chose to ignore them to bury them under more fluffy activity, bureaucracy, and colorful reports. It is a fallacy to pretend that this shift could happen without penalizing the idiosyncratic ClearCase. For 10 years, we have been waiting for a resolution of the conflicts. It is time now to admit that the chasm can only get wider.

Our conclusion could seem to be a very negative view of ClearCase if it wasn't balanced with all the useful and promising functionality we have found in ClearCase, and which we have described at length in this book. In this context, our advice is only the most effective one we may deliver.