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

Chapter 9. Secondary Metadata..... 1

Triggers..... 2
Comments..... 7
Scrubbers..... 8
Attributes..... 11
Hyperlinks..... 13
Type managers and element types..... 16
Native types..... 19
Summary..... 23

9
Secondary Metadata

We have seen two kinds of ClearCase objects:

- *mvfs* objects (elements and derived objects—let's say *files*)
- Metadata (label and branch types and instances)

ClearCase database are clearly structured (indexed) towards designing the first ones as the *first class citizens*.

This can be felt with the way in which queries about the former are faster than (non-trivial) queries concerning the latter.

We may notice that in recent releases of ClearCase, the `find` command has been extended to support searching for arbitrary objects (not only files), and thus also for metadata and types. So far in our experience, the efficiency for the latter has been the same as this of `lstype`, and the expressive power of the supported queries is still restricted. But who knows? This may change.

It is time now to turn our investigations onto some details of the primary metadata met already (labels and branches: see Chapter 6), and some less essential, yet useful kinds of metadata, which can be attached to both of the former kinds, as well as to most vob objects (but not to derived objects).

We'll find consistency of design, and the advantages of having type objects, suitable for maintaining information shared by many instances.

This chapter will thus cover:

- Secondary metadata types: attributes and hyperlinks, but also comments and triggers
- ClearCase scrubbing mechanisms concerning metadata
- User-defined types and custom type managers
- Native type managers, especially `text_file`.

Triggers

Let's first deal with triggers. They are not really metadata, but since there is a `trtype`, may be dealt with as such in some contexts.

Triggers are very popular. You'll find a lot of them from numerous web sites, and this in itself would be enough of a justification for us to leave them alone.

We'd like however to argument *against* using triggers—almost at all. Triggers are one of those slippery slopes on which newbie administrators jump joyfully, and which end up spoiling the joy of ClearCase for users, thus defeating any reasonable purpose. It is all too easy to write (bad) triggers, and all too difficult to get rid of them. The last point is based on experience: whereas it is easy to list the existing triggers (`ct lstype -obs -kind trtype`, `"-obs"` not to miss the ones which would have been removed), there is no way to know who depends on them, and in particular what scripts might break if you remove them. In practice, triggers are seldom removed: only when a bug can be found to have a bad impact. And seldom are they re-examined to check whether they still fit their original purpose (more on this in the further sections).

Triggers are vob local: in a trigger environment, things happen differently in different vobs.

They are not replicated (thank God!).

If they don't break on Windows, they'll break in snapshot views, or in the remote client, or via Eclipse, or with clearfsimport, or using cygwin, and the list is not exhaustive.

In order to write "portable" triggers, one needs a version of Perl that would satisfy some annoying and artificial constraints—be available and consistent in all contexts, be independent from the user view—and one starts using the one bundled with the ClearCase installation, which is limited and volatile (may change at the next upgrade), and... not meant for that! This clearly competes against maintaining Perl in a vob (see *Chapter 8*, *Tools Maintenance*).

In short, triggers are surprising. SCM is about making development manageable by users, making it easy for them to understand what happens and why. Sharing their experience and saving their efforts. Triggers are deceitful as a promise of a free lunch: they are hard to get right, hard to debug, hard to get coherent across the many contexts in which users may find themselves, and hard to keep right in the context of other triggers (in what order are they called?). And triggers are also unfair: users cannot practice creating ones, cannot usually skip them, or fix them.

There are two kinds of administrators; the skillful ones don't write triggers. Existing triggers have unfortunately been written by the others.

Maybe a few examples might help illustrating this. Let's pick them from a popular page in the IBM web site: ***The ten best triggers***.

NO_RMELEM

This is number one. Two scripts for UNIX and Windows, using the version of Perl bundled with ClearCase, respectively *Perl* and *ccperl*, and the same inline code: unconditional `exit 1`. The first comment is of course to question the need for this trigger. As we saw earlier, `rmelem` is a well-protected command. As a normal user, you won't be allowed to remove an element you do not own, which has any version not created by you, or which bears labels or hyperlinks. So, no real benefit.

What are its costs? This is clear: it prevents a user who notices he just created a wrong element (typical scenario: an evil twin of an existing element either in a different branch or a different directory) from removing it immediately before somebody else has any chance of accessing it by mistake.

The good point is that it is easy to work around:

```
$ PATH=~/bin:$PATH
$ cat <<eot > ~/bin/Perl
echo Skipped \$*
eot
$ chmod +x ~/bin/Perl
$ ct rmelem -f aaa
Skipped exit 1
Removed element "aaa".
```

Please note that we are not being facetious here when we write that this is a good point. Let us also trust the author that this is intentional! He could have put the full path to Perl (which would fail `rmelem` in a different way on Cygwin though). If he did not do it, it was precisely in order to allow knowledgeable users in a real need to work around the trigger. There remain several questions:

- Why wasn't this documented? Our belief is that it is because this is "security by obscurity". Triggers belong to the realm of naive security.

- How can experts such as this script author make such objective mistakes as to recommend such a trigger? Again, we can conjecture that he doesn't know. Two possible scenarios might explain why he wouldn't know:
    - He has been using this trigger for so long that he is now blind to the real behavior of ClearCase (`rmelem` was not always as secure as it is now): remember that in presence of triggers, you are not dealing with ClearCase anymore...
    - He only ever uses a vob owner account, and has thus never experienced the normal user situation. This would be a harsh criticism, if we wouldn't know some adverse environments in which tools such as sudo (which allows one to change uid just for running the few commands requiring super user rights) are banned "for security reasons".

CHECK_COMMENT

Again coming in platform-specific guises and using pathless bundled Perl versions, but with a more sophisticated behavior than previously, requiring a script with a full path.

As previously, let's first consider the intention: gently blame the author of a checkin for not feeding a comment. Let's go to the costs. There are several:

- Prompting the user interactively for every checkin without a comment, starting a GUI (unless on UNIX the `DISPLAY` variable is unset, in which case the interactive prompting is textual in the shell). This is a small penalty, unless one is performing a mass-checkin. In this case, the best is to abort (killing the process) and start again with a dummy comment; hopefully not at the end of a session which would perform a cleanup for a long task.
- Pushing people to write those comments. This is probably our worst critique: forcing users to write comments is counter-productive! It only ends up in ruining any value the comments might have, leading to the fact that nobody will ever bother to read them!

We may now share some comments on the implementation.
First, the installation was meant to be performed on Windows only; if one does it on UNIX, the use of double quotes leads the shell to interpret the double backslash (`-execwin "ccperl \\mw-ddiebolt\triggers\check_comment.bat"`) as a single one, explicitly escaped, which results in a wrong path and a failure of the trigger on Windows (fortunately allowing the checkin):

```
Can't open perl script "\filer\views\marc\check_comment": No such file ##
                                                                or directory
cleartool: Warning: Trigger script for "CHECK_COMMENT" returned failed ##
                                                                 exit status
```

Next, the use of two distinct files (a Perl script disguised as a Windows batch file, and a small wrapper with a `.pl` extension and invoking the former from UNIX, which is not shown on the IBM web page) is a common practice in the ClearCase distribution. It is completely unnecessary once one invokes Perl explicitly! Only one common and simpler file is thus needed.

Finally, the script is now shared from a filer, automounted as `/net/titeuf` on UNIX and known to Windows as `\\mw-ddiebolt`. In any case, this makes the trigger depend on network connectivity, which is a dangerous dependency for a command as useful as `checkin`. Fortunately again, failing the trigger does not fail the checkin. Summary on this trigger? No real benefit: 100% cost. With an implementation which was never reviewed critically!

REMOVE_EMPTY_BRANCH

Let's look at the third case in this list, which is again gradually more interesting (it uses environment variables).

First as previously, the intention. We may surprise you, but it is indeed often a good idea to remove empty branches! These would keep matching in the config spec after the user had forgotten about them, preventing her from being informed of new deliveries.

Now, straight to the implementation. This is only a Windows trigger. Do not set it in an inter-operating environment as it will prevent UNIX users from unchecking out anything!

This is the comment as with the previous trigger about the useless batch preamble. The interesting aspect in this code is the `...`comments, and by this we mean the commented `printf` commands. What they tell us is that triggers are hard to debug, even if it is possible to to set the `CLEARCASE_TRACE_TRIGGERS` environment variable, or to invoke ccperl with the `-d` flag starting the debugger.

Let's uncomment those print statements, and on the contrary, comment the actual command away. Now, let's create a subbranch, checkout, and uncheckout:

```
$ ct lsvtree .@@/main/mg
.@@/main/mg
.@@/main/mg/0
.@@/main/mg/aa
.@@/main/mg/aa/0
$ ct co -nc .
Checked out "." from version "/main/mg/0".

$ ct unco -rm .

Trigger is fired \main\mg\0...

\main\mgXXXXXXXXXX Count : 10

Only version 0

\main\mg\0 Count : 0

ToDo : cleartool rmbranch -force \\view\marc\test\.@@\main\mg
cleartool: Warning: This uncheckout left only version zero on branch ####
                                                                  "\main\mg".
Checkout cancelled for ".".
```

There would have gone the `aa` sub-branch. We wondered what the code was actually counting: the number of `X` it had itself inserted and removed twice, but only to the version extended name of the argument version, not to anything below it in a possible version tree…

It is not the first such trigger we meet which is actually public, old, and dangerous.

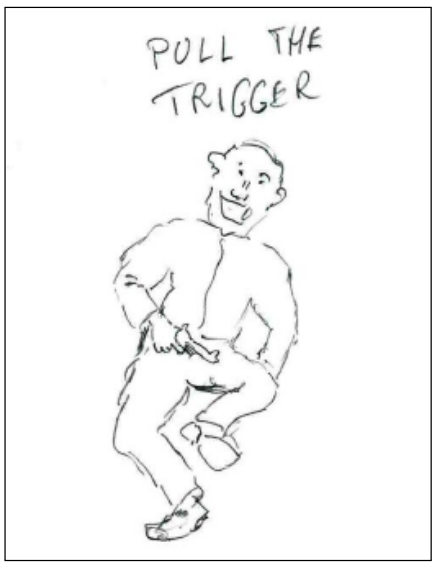
The last word on this case is that removing the branches should also be done in conjunction to `rmver`, and that doing it with a trigger is actually using the wrong weapon: this would force you to install your same trigger in all vobs. The same idea, with better code, should be implemented as a **cleartool wrapper**, only once.

In afterthought, these triggers (well, the first two) seem to us highly *political*. They are milder than many triggers in actual use. Triggers is such a touchy issue that IBM experts cannot openly write what we write here (and this is why we have to write it!): avoid them!

This page was first published in 2004, and last edited in 2006. It was labeled as *Introductory*, which it indeed is. It had been accessed more than 20000 times when we read it, and had been rated four out of five rated by 116 people.

Its title is undoubtedly misleading. It is a pleasant introduction to triggers, showing scenarios in a pedagogic order, certainly not a list of best examples. But it does show a lot.

One last point on the evil of triggers which didn't come up from the three examples reviewed, but would from another popular trigger (redundant in its purpose *with* `NO_RMELEM`): a trigger which changes the owner of new element to the vob owner. Besides sharing with `NO_RMELEM` a misguided intention, it displays a performance impact: cleartool will be invoked on any new element. This is enough to defeat any attempt at optimizing mass creation.



Comments

Next kind of exceptional metadata: comments. Comments are the lowest common denominator. The bottom of the bin. They are untyped. You cannot really search them explicitly (for example using the `ct find` command)... Well, here is what you can do:

```
$ ct lstype -kind lbtype -fmt '%n: %Nc\n' | grep foo
```

This searches for label types with a comment matching `foo`. The comment is displayed using the `-fmt` option, with the `%c` pattern, modified with `N`.

Only, this will not work well on multiline comments; the `N` modifier is only there for the last newline, allowing us to ensure that we produce at least one line per type, even if there is no comment. In case the match is on the second line, the type name will be filtered away.

OK, one can do better. Here is how we can handle the multiline comments properly, thanks to the ***ClearCase::Argv*** Perl module:

```
$ perl -MClearCase::Argv -e \
'$c=new ClearCase::Argv({ipc=>1,autochomp=>1});
for my$t($c->lstype([qw(-kind lbtype -s)])->qx){
  $t=~s/ \(locked\)$//;
  print "$t\n"
    if grep /foo/, $c->des([qw(-fmt %c)],"lbtype:$t")->qx}'
```

This is acceptable? Good.

`ipc=>1` ensures there is a single background cleartool invocation for all the types. `autochomp=>1` strips ending newlines from the output of cleartool commands. The `des` function produces a list, which is tested by the `grep` one. As an empty list is a valid list anyway, we may skip the `N` modifier in the format string.

Well, our point was anyway that the situation gets much better if we needn't parse multiline output and tell things apart that may or may not relate with each other. So, comments are useful only if used sparingly, and as a last, informal resort.

One corollary is that if you have nothing special to say, the most efficient way to do it is by keeping silent, that is, avoid polluting the comment space with *forced* comments (OK: triggers was the previous paragraph).

Scrubbers

Don't be impatient: we're getting closer.

Our next digression on the way to metadata is on the issue of scrubbers. You noticed the plural? Yes, there are two, typically scheduled as tasks: `scrubber`, and `vob_scrubber`.

We shall be concerned here with the latter, so let's talk of the former first.

The *scrubber* (already mentioned in the *Removing derived objects* section of *Chapter 3, Build Auditing, and Avoidance*) takes care of maintaining the pools within reasonable disk space limits. Actually only two of the three pools: the *cleartext* and the *do* pools. Pools are directory trees in the vob storage outside the database that contain the actual data. The third is the source pool which stores versions of file elements. We'll review the most interesting case of source containers—the case of text files—in the end of this chapter. We'll keep on with pools in the next chapter. Please note now

that we started this chapter by mentioning two kinds of ClearCase objects: files and metadata. Files are actually produced, and thus scrubbed—not only derived objects, but versions as well (the *cleartext containers*). See more on the scrubber in *Chapter 10, Administrative Concerns*.

The vob scrubber deals with database items: events and oplogs. We'll be back to oplogs in *Chapter 11, MultiSite Administration*.

Note how events are classified in the way the vob scrubber deals with them (not uniformly).

For example, `mkattr` events are kept (by default) only for seven days, and the last one for an attribute type for 30 days, as reads from the `/var/adm/rational/clearcase/config/vob_scrubber_params` file:

```
   event mkattr  -keep_all 7 -keep_last 30
```

You can always check the log for the last runs, at least using perl:

```
$ ct des -fmt "%[replica_host]p\n" \
 replica:$(ct des -fmt «%[replica_name]p\n» vob:.)
beyond.lookingglass.uk
$ ct lsvob /vob/foo
* /vob/foo /vobstg/foo.vbs public (replicated)
$ ct getlog -host beyond.lookingglass.uk -full vob_scrubber | \
 perl -n00e 'print «$p$_» if $p and m%/foo%; $p=/^Number/?$_:q()'
Number of  Events      Events      Events      Kind of event
objects    before      deleted     after       by meta-type
----------  ----------  ----------  ----------  -----------
     1        162         79          83  versioned object base
     3        209         58         151  replica
     3          5          0           5  pool
     1          1          0           1  replica type
    13         13          0          13  element type
    76         78          0          78  branch type
  2545       7648       1138       6510  label type
    27         27          0          27  attribute type
    27         27          0          27  hyperlink type
  3376       3376          0       3376  directory element
  3623       3623          0       3623  file element
 18142      18147          3      18144  branch
 13959      88599      26403      62196  directory version
 17873      97310      27923      69387  version
     4          4          0          4  symbolic link
 95625          0          0          0  derived object
  3199        628        178        450  hyperlink
158497     219857      55782     164075  total in VOB
Event scrubbing done.
Oplog scrubbing done.
Finished VOB «/vobstg/foo.vbs» at 2010-08-29T07:14:10+01.
```

The `vob_scrubber` log is not formatted, which means that it is not trivial to know what exactly to wait for: in our example we are only interested in `/vob/foo`. Even if it is rotated relatively often, the log is still pretty large. We read it in paragraph mode (the `-n00` flag), and print when finding a suitable match, both the previous and the current paragraphs. This implies that we must remember the previous paragraph, and print it in case it was of the right kind. This is roughly what the small inline script does.

This processing allows us to understand the presence or absence of certain events in the `lshistory` output. In this case, we could verify that no `mkattr` events had been scrubbed for the last period. We knew the attribute was set recently to a label type (which was confirmed by the presence of the last `mkattr` events), but the question was to know whether it had possibly had another (incorrect) value previously. Now, we could show that no such event had been scrubbed.

If the events you'd like to inspect have been scrubbed, there is still one recourse, at least in replicated vobs: `dumpoplog`. This is a useful yet under-documented command, which produces, slowly, a very verbose output. You need again to be prepared to post-process it, in a flexible way. Here is one example, run within a vob, with a view set:

```
$ mt dumpoplog -l -vreplica wonderland -name -since 7-Aug | \
  perl -n00e 'print "$1 $2\n" if /^op= mklabel.*^op_time= ([\
w:-]+).*^obj_oid= .*?\(.*?: (.*?)\).*^lbtype_oid= .*?\(FOO\)/sm'
2010-08-09T10:57:55Z /vob/foo/.@@/main/1/demo.txt/main/1
2010-08-09T10:57:55Z /vob/foo/.@@/main/1
2010-08-09T10:59:01Z /vob/foo/bar@@/main/1
...
```

We use again the paragraph mode, and now we parse the result in multiline mode (`m` modifier) where the "`.`" wildcard may match newline characters, allowing nevertheless the beginning and end markers, respectively "`^`" and "`$`", to match at line boundaries (`s` modifier). In the regular expression, we successively match word constituents (`\w`) extended to colons and dashes (in order to match time stamps), and sequences of arbitrary characters in non-greedy mode ("`.*?`").

The preceding example shows which versions have been labeled with `FOO` labels, and this possibly after the `mklabel` events have been scrubbed.

Now, oplogs may be scrubbed too. It is even both recommended and recommendable to scrub them. This just doesn't happen by default, and thus requires editing the host global param file `vob_scrubber_params` mentioned earlier (probably the best way), or a per vob setting, a file with the same name placed in the vob storage area. The setting is to keep oplogs for a certain time, and the recommendation is to keep at least two months of oplogs to be able to reproduce sync packets that would have been lost before being imported on remote sites.

Attributes

Attributes are the preferred alternative to comments. They are preferred because their **typing** allows to avoid depending on parsing informal text. This typing involves two levels:

- A low-level typing determining the format of the values and a set of operators of the query language applying to them
- A declarative level, reserving a name which formally identifies the type

Attributes may be attached to different vob objects: file or type objects.

The existence of options restricting their application to elements, per version or per branch, may bias their use toward the first kind (file, rather than type objects). Attributes are also often used to store status information—values that may vary. These are uses for which the `-shared` option (in a MultiSite context) would be as questionable as for labels or branches.

We tend to apply attributes mostly to objects of the second kind, almost only label types; and with stable values, so that the `-shared` option is safe and justified. Attributes allow to attach information at the right level of stability in a structure of inter-related types, thus offering significant flexibility. The same attributes may be used on different sites, in conjunction to locally mastered and applied labels:

```
$ ct mkattype -nc -shared AAA
Created attribute type "AAA".

$ ct mkattr AAA '"aaaa"' lbtype:ZOO
Created attribute "AAA" on "ZOO"
```

The requirement to enclose string values within double-quotes may surprise, and leads at times to errors. It doesn't concern the other value types.

We'll take an example of applying (unshared) attributes to versions, and it is from the ***ClearCase::Wrapper::MGi*** Perl module. We use attributes there to record the fact that a label of a certain type hierarchy was removed, and from which exact version on. The background is that for every label family, we maintain a chain of incremental fixed label types, recording the successive positions of a common floating label type. Suppose our floating type is named `FOO`, then the incremental types, applied at every stage to the concerned versions only (therefore offering better performance and manageability), are named `FOO_1.00`, `FOO_1.01`, ..., `FOO_2.34`, and so on.

First, we create a family label type, which is identified to a floating label (`FOO`). Its first increment, a fixed label `FOO_1.00`, is created automatically (note that `ct` is aliased as usual to the standard cleartool, and `ctx` is aliased here to the cleartool wrapper, actually from ***ClearCase::Wrapper***):

```
$ alias ctx ct
alias ctx='/usr/bin/cleartool.plx'
alias ct='/opt/rational/clearcase/bin/cleartool'
```

One may also set the `expanded_aliases` shell option, or prefer to define functions instead of aliases:

```
$ function ctx {
  /usr/bin/cleartool.plx $@
}


$ ctx mklbtype -nc -fam FOO
Created label type "FOO_1.00".
Created label type "FOO".
Created attribute type "RmFOO".
```

We can create the successive increments with the `mklbtype -inc` command:

```
$ ctx mklbtype -nc -inc FOO
Created label type "FOO_1.01".
```

We use a two-level numbering scheme to guarantee that the types may be sorted naturally, and to leave some space for growing the size of the namespace if need-be (for example, jumping to `FOO_2.000` after `FOO_1.99`). This strategy seems to be enough to allow using only the floating label to name the baseline in development config specs, and to produce afterwards a list of config spec rules equivalent to any situation in the past, for example:

```
element * FOO_1.23
element * FOO_1.22
…
element * FOO_1.00
```

However, this holds only as long as the `FOO` label was not removed from a version at any point. To support this case, we set an attribute when we remove a label (as well as to previous versions bearing a label of the family). In our example, the attribute type would be `RmFOO` (it must be specific to the label type), and the value records the increment at which `FOO` was removed. The version will of course still bear the fixed label corresponding to the increment at which `FOO` was applied, so that it is correctly considered part of any equivalent config spec corresponding to an increment in the range until this at which `FOO` was removed.

```
$ ct des -fmt "%Nl\n" foo.txt
FOO FOO_1.14
$ ct des -ahl -all lbtype:FOO
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.23@/vob/foo
$ ctx rmlabel FOO foo.txt
Removed label "FOO" from "foo.txt" version "/main/mg-013/18".
Created attribute "RmFOO" on "foo.txt@@/main/mg-013/18".
$ ct des -fmt "%Nl\n" foo.txt
FOO_1.14
$ ct des -aattr -all foo.txt
foo.txt@@/main/mg-013/18
 Attributes:
    RmFOO = 1.23
```

Every config spec rule produced (in the list above) must thus be corrected as follows (here for increment `1.15`):

```
element * "{lbtype(FOO_1.15)&&!attr_sub(RmFOO,<=,1.23)}"
```

The main problem with attributes attached to types, is that queries are inherently inefficient, and that their performance is tied to the number of types, and thus decreases as the size of the system grows, which is clearly an undesirable property. The same is of course true of queries for file objects (using the `find` command), but these ones are better supported by the indexing strategy of the ClearCase databases.

To face this problem, one needs to revert the strategy from *query* to *navigation*, and to rely for this upon a new structure, based on hyperlinks.

Hyperlinks

We already know of two dimensions, or hyperspaces, in which to structure software configurations: the most trivial, and least interesting one is directory trees; the most interesting one is audited dependencies. Hyperlinks provide a third layer (or rather, dimension). What makes directories less interesting is that they record the contingent history of the development, and thus lose a real value of relevance: they tell you about your background (which has its value, for sure), not about your actual life situation. The audited dependencies are real stuff; furthermore, they are collected for free. Sometimes, they lack flexibility: there is a lot of interesting stuff beyond plain reality.

Hyperlinks have this flexibility. For a start, they may relate other objects than elements: label types for instance.

We already mentioned in the previous chapters the hyperlink mechanisms ClearCase uses for merging (`Merge` hyperlinks in *Chapter 7*, *Merging*) and global typing (`GlobalDefinition` hyperlinks in *Chapter 5*, *MultiSite Concerns*).

We use them quite extensively to build label type families, with a floating label type representing the current baseline (and applied to all the elements present in a configuration), and fixed types recording the successive increments of the development history, and only used indirectly to recreate an equivalent config spec pointing to a past configuration.

We described this earlier in the *Attributes* paragraph: the successive increments are linked together with hyperlinks of `PrevInc` type (defined in ***ClearCase::Wrapper::MGi***). The top of the linked list, quite a volatile information, is accessible via the floating label type (a well-known and stable name, even if the set of versions it is applied to changes frequently) with an `EqInc` hyperlink. Both kinds of hyperlinks are maintained by the `mklbtype` function of our cleartool wrapper.

The following transcript illustrates the relationships explained above between the floating label `FOO` and its increments (the fixed labels of the `FOO` family, such as `FOO_1.22, FOO_1.23` and `FOO_1.24`).

The family type floating label `FOO` is linked to the current increment `FOO_1.23`:

```
$ ct des -ahl -all lbtype:FOO
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.23@/vob/foo
```

This is how the successive incremental label types are linked together:

```
$ ct des -ahl -all lbtype:FOO_1.23
FOO_1.23
 Hyperlinks:
    PrevInc -> lbtype:FOO_1.22@/vob/foo
    EqInc <- lbtype:FOO@/vob/foo
```

Now we add the next increment:

```
$ ct mklbtype -nc -inc FOO
Created label type "FOO_1.24".
```

And see how the label hierarchy is re-arranged:

```
$ ct des -ahl -all lbtype:FOO lbtype:FOO_1.24
FOO
 Hyperlinks:
   EqInc -> lbtype:FOO_1.24@/vob/foo
FOO_1.24
 Hyperlinks:
    PrevInc -> lbtype:FOO_1.23@/vob/foo
    EqInc <- lbtype:FOO@/vob/foo
```

This is the point for a syntactical note: there is a difference in support for describing hyperlinks and attributes. Or rather, the syntax we used in the above example may also be used for attributes:

```
$ ct des -aattr -all lbtype:ZOO
ZOO
 Attributes:
   AAA = "aaaa"
```

...but there is another handy syntax for attributes, which has no equivalent for hyperlinks:

```
$ ct des -fmt '%[AAA]NSa\n' lbtype:ZOO
"aaaa"
```

This prints the value (`aaaa`) of the `AAA` attribute. As one can see, this syntax is flexible and allows to tailor one's needs so as not to require post-processing (except to get rid of the double-quotes).

We also use hyperlinks across vob boundaries, and there mostly ones of the predefined `GlobalDefinition` type, even if we do not set an admin vob for this purpose (see Chapter 5).

The main problem with the concept of admin vob is that there is only one `AdminVOB` hyperlink type (which is used to link vob objects between one another), and its effect is global. This therefore constrains other sites in an excessively inflexible way, which ends up defeating collaboration. Client vobs cannot really be used without their admin vob, and importing vobs from different sites, which wouldn't have been submitted to a centralized control from the beginning, invariably results in conflicting requirements to use incompatible admin vob hierarchies.

Fortunately, `GlobalDefinition` hyperlinks may be created explicitly, by our wrapper or other scripts, and existing instances are obeyed to functions such as `lock`, `rename`, and so on without having to submit to the constraints of admin vobs.

A final note on hyperlinks is that one may use textual hyperlinks as an alternative to attributes. These hyperlinks do not reference any other object but instead a new text value. This may be useful if one needs to create an object to hold the value, for example, in order to be able to transfer its mastership, or if one wants to create several instances of the same type "attached" to the same object.

Type managers and element types

ClearCase comes with a good few native types, but also allows the users to define their own. At least, this was the intention, and since the very beginning—one sees there that ClearCase was a product of the flamboyant times of object orientation, and carried ideas experimented in *Apollo Domain* typed file system.

The list of native types found depends on the vob feature level, and it has increased over time. Notable additions had to be made to support file formats that departed from the implicit assumption that text should break on new lines (so as to be generically printable, without the need for a parser, aware of the specific syntax). This concerned the HTML and XML files produced by some popular tools, mostly on Windows. More recent additions concern the variants of UTF, to support Unicode in different contexts.

The magic files

One may use element types *explicitly*, with an `-elt` option to `mkelem`, or with the `chtype` command, but the stereotypical way is to use them *implicitly*, via the **magic file** mechanism, in the old UNIX tradition.

A `default.magic` file is provided by the ClearCase installation, with rules to take into use all the existing native types... and more. The syntax of magic files is documented in the `cc.magic` man page.

Here is a sample extract from it:

```
$ cat /opt/rational/clearcase/config/magic/default.magic
# Check stat type
directory : -stat d ;
block_device : -stat b ;
char_device : -stat c ;
socket : -stat s ;
fifo : -stat f ;

# Match by name without examining data
program compressed_file : -name "*.[eE][xX][eE]" | -name "*.bin" ;
object_module compressed_file : -name "*.[oO][bB][jJ]" ;
shlib library compressed_file : -name "*.[dD][lL][lL]" ;
```

```
zip_archive archive file : -name "*.[zZ][iI][pP]" ;
tar_archive archive compressed_file : -name "*.tar" ;
…
# assumed to be text
java_source source text_file : -name "*.[jJ][aA][vV][aA]" ;
...
# catch-all, if nothing else matches
compressed_file : -name "*" ;
```

It is made of rules, with a first match logic similar to this of config specs. Each rule consists in two parts: an ordered list of types and a set of selection criteria. When a selection criterion matches, the types are tried in order, and the first one found in the current vob is used; if none of them is found, an error is spit. If no selection criterion matches, an error is also spit, but `default.magic` has a catch-all rule to preempt this case and use then `compressed_file`.

Let us note while we are here that `default.magic` anticipates the creation of some user defined types, such as `java_source`. We shall fulfill this precise anticipation in the next section.

ClearCase design does not intend the users to modify `default.magic`. On the contrary, it suggests that they would create new magic files in other directories, and take them into use via a `MAGIC_PATH` environment variable (a local `.magic` file in every user's home directory will also be searched by default, and this one might of course be a symbolic link to a site-wise shared copy). It is wise to restrict oneself to provide in local `.magic` files only specific rules and criteria to override some critical element types, and to keep the standard path to `default.magic`—with its catch-all rule—last in the `MAGIC_PATH`. This ensures that one doesn't shut oneself off from the benefits of upgrades from further ClearCase releases.

Remember that the catch-all rule will preempt any further rule if found too early. This might happen if the built-in path was not the last in `MAGIC_PATH`, or if magic files with names later than `default.magic` in sorting order would be placed there: they could never be reached. Even if any directory in `MAGIC_PATH` may contain an arbitrary number of magic files, it may be wise to keep those to a bare minimum, for fear that they might compete against each other, with the arbitration of the alphanumeric order of their respective names, which the `MAGIC_PATH` mechanism is precisely meant to avoid.

What may be recognized here is yet one special case of the fundamental challenge of SCM: how to manage sharing, how to propagate changes made locally, so that all benefit from them without surprises and unanticipated conflicts? ClearCase unfortunately falls short of offering a satisfying answer in the context of element types.

User defined types

All the types the user might define do not present the same challenges.

Type without a new manager

We mentioned the existence of a potential `java_source` type in the `default.magic` file. This is an insightful provision, as we'll see in *Chapter 12*, *Challenges*. Leaving until then the rationale for such a change, let's consider implementing this type now.

This is a good example of a trivial yet useful—nay: mandatory— type definition. The time stamp of java source files should be this of the last file saving: it ought to be preserved at check in, which is not the default behavior. The command is simple:

```
$ ct mkeltype -c 'Preserve time of java sources at checkin' \
 -super text_file -ptime java_source
```

Note that there is no mention of a `-shared` flag: element types are shared by default (they may be used at other replicas). There is a restriction related to this: they cannot be changed afterwards in replicated vobs (using the `-replace` flag of the `mkeltype` command will yield an error):

```
$ ct mkeltype -rep -super binary_delta_file java_source
cleartool: Error: Can't redefine element types when VOB is replicated.
cleartool: Error: Unable to replace definition of element type ##########
                                                          "java_source".
```

It may be a good idea to make the type `-global` (unless it is only a cosmetic issue: you decide). In any case, this type will have to be created (or maybe copied) into every vob in which it is needed.

However, it will be replicated properly, without any additional configuration, and will be directly usable in other replicas, both explicitly and implicitly.

New type manager

How about creating a more ambitious type—a type which would be associated with a new type manager?

We'll explore here only the most simple case, hence avoid writing code to implement some type manager methods in ways specific to a new type (which is covered in an IBM ***white paper*** by Laurent Maréchal). Examples of useful managers could be: managing tar (and other container) files, by actually extracting them and maintaining the resulting directory trees (with a program similar to ***synctree***); or managers providing compare and merge capabilities for file types requiring an ad-hoc graphical editor.

One manager we might want to implement would manage source files in such a way that new branches spawning from the root of the version tree would not result in a full copy of the base version in the source container.

Type managers are a collection of methods. In UNIX, these methods are grouped in directories under the `/opt/rational/clearcase/lib/mgrs` directory. Most of the entries there are actually symbolic links, with the help of which the actual code is shared among multiple types (including implementation of type inheritance). In Windows, all the methods are listed instead in a single `map` file (in `C:\Program Files\Rational\ClearCase\lib\mgrs`).

The most trivial implementation we are alluding to would be to create a (completely useless) symbolic link to an existing type manager, let's say `text_file_delta`. Let our symlink be named `foo`, and let's explore the implications of creating a new element type using it:

```
$ ct mkeltype -nc -super text_file -man foo foo_file
Created element type "foo_file".
$ ct co -nc .
$ ct mkelem -nc -elt foo_file foo
Created element "foo" (type "foo_file").
Created branch "mg" from "bar" version "/main/0".
Checked out "foo" from version "/main/mg/0".
$ ct ci -nc .
```

So far, so good (note that for this to work, you actually need to be using a local view). But what about editing this file in another view, hosted on another host? You'll get an error until you install the type manager on the second view server. On Windows, you'll have to do it using the map file. In any case, remember to take a backup of your changes, in order to be able to reapply them after every ClearCase upgrade; it is hard to predict which ones will overwrite the map file, or tamper the `mgrs` directory.

The same issue awaits you with MultiSite: a prior type manager installation will be needed on all vob servers to allow the export and import of sync packets containing events involving the custom type manager. ClearCase doesn't assist you with this, beyond giving a clear error message, which is unfortunately easy to miss, and to lose while rotating the synchronization logs.

Native types

ClearCase native element type managers differ mostly in the way they use source and cleartext containers: the former are used for storing the data, and the latter for presenting it (this of the selected version only) to the user or her tools. Do the two need to differ at all? Can multiple versions share a common source container? And what are the implications in terms of performance and storage?

Binary types

There are two basic native binary types: `file` and `compressed_file`. There is also one special type: `binary_delta_file`.

The `file` type uses the `whole_copy` type manager. Elements of this type actually have no *cleartext containers*: every version is stored integrally in a new source container.

Note from the following output that both container paths are identical, and from the source pool (`s/sdft`):

```
$ ct mkelem -nc -elt file file
$ mkfile 1k file
$ ct ci -nc file
$ ct dump file | grep cont=
source cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-3774301fbe9311df913100156004455c-6g"
clrtxt cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-3774301fbe9311df913100156004455c-6g"
$ ct co -nc file
$ ct ci -nc file
$ ct dump file | grep cont=
source cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-52e43027be9311df913100156004455c-qq"
clrtxt cont="/vob/foo.vbs/s/sdft/32/3e/####################################
                                    2-52e43027be9311df913100156004455c-qq"
```

The `compressed_file` type uses `z_whole_copy`. The source containers are compressed, and thus uncompressed cleartext containers are generated on access:

```
$ ct mkelem -nc -elt compressed_file cfile
$ mkfile 1k cfile
$ ct ci -nc cfile
$ wc -l cfile
     0 cfile
$ ct dump cfile | grep cont=
source cont="/vob/foo.vbs/s/sdft/10/42/####################################
                                    3-1ec43037be9411df913100156004455c-1m"
clrtxt cont="/vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c"
$ ls -l /vob/foo.vbs/s/sdft/10/42/3-1ec43037be9411df913100156004455c-1m \
 /vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c
-r--r--r-- 1 vobadm adm 1024 Sep 12 18:30 ################################
               /vob/foo.vbs/c/cdft/30/15/1ec43037be9411df913100156004455c
-r--r--r-- 1 vobadm adm 36 Sep 12 18:28 ##################################
         /vob/foo.vbs/s/sdft/10/42/3-1ec43037be9411df913100156004455c-1m
$ ct co -nc cfile
$ ct ci -nc cfile
$ ct dump cfile | grep cont=
```

```
source cont="/vob/foo.vbs/s/sdft/10/42/#################################
                                    3-8c643047be9411df913100156004455c-9n"
clrtxt cont="/vob/foo.vbs/c/cdft/20/2/8c643047be9411df913100156004455c"
```

One interesting conclusion is that compressing files will actually first increase the space consumption. Space will be saved after enough versions will have been created, assuming only few of them remain accessed (and the others' cleartext containers get scrubbed). The break-even will depend on the compression factor.

The last binary type (`binary_delta_file`) is actually not a binary type, but a text one, only lacking a predictable record delimiter. It is the supertype used by the various HTML and XML types mentioned earlier.

Text type

By far the most interesting element type, for purposes of version control, is `text-file`.

A basic understanding of the format of the source containers is an investment which may pay back in some cases of catastrophe recovery.

There is one single source container for all the branches and all the versions.

Creating a new version actually involves creating a new updated source container and removing the old one. The information in the container is however stable.

The container starts with a header, beginning with `^S` and ending in `^E`. A 2-digit id is assigned to branches `^B` and versions `^V` bottom-up (that is `/main` == `0 0` last).

Here is an example:

```
^S db1 4
^V 72bda4c9.fbe111dc.8c74.00:01:84:16:9f:47 4 1 47eb692e
^V 6c2da4c5.fbe111dc.8c74.00:01:84:16:9f:47 4 0 47eb692e
^B 6c2da4c1.fbe111dc.8c74.00:01:84:16:9f:47 4 2 1
^V 7c7075ce.fb1911dc.8b8a.00:15:60:04:45:5c 3 1 47ea1a4a
...
^B e71bad18.f20c11dc.8ef5.00:01:84:16:9f:47 0 0 0
^E e71bad14.f20c11dc.8ef5.00:01:84:16:9f:47
```

The first digit of the id identifies the branch. For branches, the second digit identifies the parent branch, for versions, the version id.

```
0 0 /main
1 0 /main/mg-001
2 1 /main/mg-001/mg-002
4 2 /main/mg-001/mg-002/mg50
3 1 /main/mg-001/mg
```

The last field on the line is a time stamp (*time2date*) for versions.

The header is followed with annotations of two kinds: insertions `^I` or deletions `^D`, with the id and the number of lines:

```
^I 1 1 613
package ClearCase::SyncTree;

^D 4 1 1
^D 3 1 1
^D 2 1 1
^D 1 7 1
^D 1 4 1
^D 1 2 1
$VERSION = '0.47';
^I 1 2 1
$VERSION = '0.48';
^I 1 4 1
...
```

Here is a possible recovery scenario: an administrator misunderstood a user request and removed a critical version (`rmver` or `rmbranch`).

A previous version of the source container is found in a filer snapshot. It is possible to identify the container by looking at the time stamps and grepping some older data.

The challenge: produce from there a cleartext container corresponding to the lost version(s), and check it back in again.

By comparing with the current source container, it is possible to identify the oids of the missing version and of the branch object.

```
^V 2594991b.490d11dd.9673.00:15:60:04:45:5c 17 2 486dfe83
...
^B bd588b72.237145da.8ecf.6a:2a:92:7a:ac:e1 17 15 1
```

In practice, one could interpret `17` as the branch, and thus `17 2` as version `2` on it, checking that this one had been removed.

So, from this container, one can run the `construct_version` method from the command line, and produce a `/tmp/clrtxt.out` file using the version oid found in the source container header:

```
$  srccon=/vob/foo.vbs/.snapshot/vobs_snapshot.1/s/sdft/1e/d/0-a14e41314e
7111dd925a00156004455c-qp
$ cd /opt/rational/clearcase/sun5/lib/mgrs/text_file_delta
$ ./construct_version $srccon /tmp/clrtxt.out \
 2594991b.490d11dd.9673.00:15:60:04:45:5c
```

Summary

This completes our review of metadata. Contrast the power and the beauty of element types, to the superficiality of triggers. Know to use attributes and hyperlinks, and avoid abusing comments.

The next chapter will continue with low-level concerns, close to the ClearCase implementation.