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

6
Primary Metadata

ClearCase offers auxiliary objects to help managing files: **metadata**. The different metadata types are not equally important. One way to trace a boundary is to elect as *primary* the ones appearing in the version extended view paths: **labels** and **branches**.

We do here a first pass at handling these objects, leaving deeper and less critical aspects for later.

- Singling out labels and branches
- Types and instances
- The relative roles of labels and branches
- Labels: floating and fixed
- Baselines and incremental labels
- The delivery process

Metadata in the version extended view

From the version extended perspective, we notice at once that labels and branches share a common namespace: it is not easy a priori to tell them apart. The distinction between the two has to be built on convention. It is customary to follow the example set by ClearCase itself for predefined types: uppercase for labels such as `LATEST` and lowercase for branches such as `main`.

To illustrate this, let us compare the element extended version path and its version tree:

```
$ ct lsvtree
.@@/main
.@@/main/14 (REL1)
.@@/main/38
.@@/main/br1
```

```
.@@/main/br1/6 (L1)
.@@/main/br1/t1
.@@/main/br1/t1/3 (PUB3.5)
.@@/main/br1/t1/4
.@@/main/br1/8
.@@/main/45
.@@/main/br2
.@@/main/br2/2
.@@/main/46
```

The `lsvtree` command without the `-all` option shows only *important* versions: that is, labeled (`/main/14`,`/main/br1/6`, and `/main/br1/t1/3`), last on their branch (`/main/br1/8`, `/main/br1/t1/4`, `/main/46`, and `/main/br2/2`), and versions of which new branches were spawned (`/main/38`, `/main/br1/6`, and `/main/45`).

The element's extended view shows a "projection" of the version tree: on the top level, it shows the first-level branches (`main`) and all the shortcuts, i.e. the labels that give access to a particular version of this element.


```
$ ll .@@
total 8
drwxrwxrwx 2 ann jgroup  223 Feb  4 2008 REL1
drwxrwxrwx 2 ann jgroup  932 Feb 14 2008 L1
drwxrwxrwx 4 ann jgroup    0 Sep  3 2007 main
drwxrwxrwx 2 ann jgroup 1159 Aug 25 2008 PUB3.5
```

The second-level branches (`br1`, `br2`) can be found as sub-directories of the main directory:

```
$ ll .@@/main
total 90
drwxrwxrwx 2 jgroup 2000    0 Sep  3 2007 0
drwxrwxrwx 2 jgroup 2000   52 Sep  3 2007 1
...
drwxrwxrwx 2 jgroup 2000    0 May 18 2009 br2
drwxrwxrwx 3 jgroup 2000    0 Sep 13 2007 br1
drwxrwxrwx 2 jgroup 2000  223 Feb  4 2008 REL1
drwxrwxrwx 3 jgroup 2000 1034 Sep 16 2008 LATEST
```

The other subdirectories are explicit versions: 0, 1, ... 46; and the labeled versions, on *this* branch: `REL1`, `LATEST`.

Note that for directory elements (as in our example above), all the entries (both labels and branches) are directories, whereas for file elements, *label* entries are always files as they refer to a particular version of this *file* element. *Branch* entries are always directories as they contain at least the following versions for each branch: `0`, `LATEST`:

```
$ ll foo.html@@
total 9
dr-xr-xr-x 3 ann 2000     0 Feb 14 2008 main
-r-xr-xr-x 1 ann 2000 10927 Sep  9 2008 L1
$ ll foo.html@@/main
total 5
-r-xr-xr-x 1 ann 2000 0 Feb 14 2008 0
dr-xr-xr-x 3 ann 2000 0 Feb 14 2008 br1
-r-xr-xr-x 1 ann 2000 0 Feb 14 2008 LATEST
```

Types and instances

ClearCase consistently follows a philosophy (born in *static typing*) of distinguishing between declaration/definition on the one hand and use on the other. We shall therefore systematically meet **types** and **instances** as we just did. Note that this is proper to ClearCase, and sometimes confusing to users with other backgrounds. Indeed, verification is always based on the consistency of representations, and therefore requires some degree of duplication. Splitting tasks in two steps, often separated in time and space, allows such verifications. We shall refer to this philosophy later, trying to apply it ourselves where support for it is not built in (see *Use of locking* section).

It is somewhat awkward to consistently talk of **label** and **branch** types, so that we shall, in non-ambiguous contexts, follow the common practice of using the terms *labels* and *branches*. For example, we shall speak of *applying a label*, when more rigorously, we *create* several *instances* of a single *label type*.

Functions will come in pairs: first make **type** (`mklbtype` and `mkbrtype`, for the kinds of metadata we consider here) for the declaration, and then make **instance** (`mklabel` and `mkbranch`) for the actual use.

Both label and branch types are abstract, element-independent concepts. On the contrary, instances are concrete and bound to a particular ClearCase element (either a file or a directory). A label is *applied to* it, a branch is *created on* it.

```
$ ct mklbtype -c "a concise comment, only if useful" MYLABEL
$ ct ls foo
foo@@/main/4                          Rule: /main/LATEST [-mkbranch br1]
$ ct mklabel MYLABEL foo
$ ct des -fmt "%n %Nl\n" foo@@/MYLABEL
foo@@/main/4 MYLABEL
```

We created a label type `MYLABEL` and applied a label of this type to a version `/main/4` of the element `foo`. Then we showed how to access the labeled version directly by specifying the label:

```
$ ct mkbrtype -nc br1
$ ct catcs
element * CHECKEDOUT
element * .../br1/LATEST
mkbranch br1
element * /main/LATEST
$ ct co -nc foo
Created branch "br1" from "foo" version "/main/4".
Checked out "foo" from version "/main/br1/0".
```

We first created a branch type `br1`, and then created the branch by setting a `mkbranch br1` config spec rule and checking out the element, which resulted in the `br1` branch creation of the `foo` element. The `br1` branch was spawned off the view-selected version `/main/4`. Branches can also be created explicitly by the `ct mkbranch` command (although the implicit branching described above is more common):

```
$ ct mkbranch -nc br1 bar@@/main/br/2
Created branch "br1" from "bar" version "@@/main/br/2".
Checked out "bar.txt" from version "/main/br/br1/0".
```

One more observation one can make in the last example is that the ClearCase branches *cascade*. For example, if we branch off, using type `br1` from a version `/main/2` of the element, this creates the `/main/br1` branch of it; then continuing to branch off using now `br2` from version `/main/br1/1`, we get branch `/main/br1/br2`, and so on. This is why, in the config spec, it is convenient to specify the branch type as a wildcard, such as `.../br2/LATEST`, rather than with its full name as `/main/br2/LATEST`, as the latter may not select all the desired versions (as there can be both elements with branches `/main/br2`, and `/main/br1/br2`, and so on).

Note that one cannot cascade branches indefinitely because of system limitations (of 1024 bytes for a full version extended pathname). Besides, it is not very handy either. One may be interested to take a look at the MGi extension of the ClearCase Wrapper, providing support for `BranchOff: root` rule in the config spec: it forces new branches to be created from the root (usually `/main`) rather than following the cascading mode. It maintains the genealogy with **Merge arrows** (see *Chapter 7, Merging*).

Each vob must have one and only one definition for each branch or label type, which it may inherit from another vob if using global types.

The following exclusion principle applies, by default, to branch and label instances: every element can have only one version carrying a particular label, and only one branch of any given type in its version tree (see the version tree of a current directory in the preceding example).

This may be bypassed at label type creation by using the `-pbranch` option, which allows to use the same types on different branches of an element; we cannot recommend this in general, as this leads to the possibility of ambiguous rules in config specs.

Labels or branches?

There is a great deal of symmetry between the two sets of functions (pertaining to labels and branches), and actually between the concepts. Both may be used in config specs, with very similar effect at first sight. The strategies built upon them are *dual*, to use a metaphor familiar to the electric engineer thinking of currents and tensions when considering a circuitry schema. However, as in the electrical case, this duality eventually reaches some boundaries, and we'll see which. It must be stressed that such analyses are very specific to ClearCase, and bear no validity at the abstract level of conceptual CM; they don't apply to other tools.

We already mentioned in the last chapter the main difference between labels and branches: labels can be aliased whereas branches cannot, or in other words, a given version may bear as many labels as one likes but it sits only on one single branch.

This is the technical aspect. On the functional side, one might think that branches embody *intentions* (the future), whereas labels represent *states* (the past). Now, this division is not always respected in practice, especially in config specs. This is particularly true in the context of UCM, where both labels and branches are treated as implementation details, and buried under the higher-level concepts of *activities*, *streams*, and *projects*. We shall ignore them here, not because we believe the concepts themselves would not be sound (with a reserve concerning the "project" one), but because their implementation in UCM is built upon an unfortunate foundation. This will become obvious in the following, although we won't mention it anymore until *Chapter 13*, *The Recent Years' Development*. As mentioned in *Chapter 2*, *Presentation of ClearCase*, UCM is largely covered in existing literature, and we focus here on issues ignored therein.

What needs to be stressed is that metadata can and should be used to implement communications between the members of the development team, in an objective way, hence supported by the tools. Interpretation remains necessary, but is pushed forward, or upwards if one wants to retain the "high-level" metaphor. This may and should be enforced by conventions. We shall review here, *bottom-up*, the constraints that drive the making of sensible conventions.

Parallel development

Derived objects are shared transparently under ClearCase. This happens through build avoidance, winkin, and DO identification. However, nothing of this is spread via replication through MultiSite! The alternative while using MultiSite is thus to be down to the rudimentary level of version control, or to do something to raise it back to the full ClearCase power.

Applying labels serves the latter purpose: config records are not replicated, but labels applied using them are. The sharing of derived objects is not transparent, but it is reasonably easy to reproduce them on another site, or to compare the bill of materials of objects produced locally with this of objects produced remotely.

Also, every developer needs a workspace under her own control, both protected from interferences from others and easy to update with the latest relevant changes. We saw in Chapter 5 that MultiSite considerations direct to work in branches, which the developer should create herself on a need basis. We will defer to the next chapter, on merging, the details on how to update them, but will address in the following section, the questions of delivering her work thus releasing the protection acquired while creating the branches, and also specifying clearly to her collaborators, the changes from which they might be willing to update their own environment.

Config specs

The place of choice to express semantics in such a way that the tools will obey them is the config specs. This fights the common practice of *generating* them, using once again, *higher-level* considerations.

One concern will be to avoid the pitfall (already mentioned in Chapter 2) of growing the complexity of config specs to a point when their generation might meet a demand from users.

To be convenient, config specs should be both simple (and concise) and stable.

We already mentioned (in Chapter 2) the first requirement: few generic rules so that the user can have a clear idea of which of them applies to any given element. This ought to be governed by simple considerations, ideally either of two: her own changes on one hand, or the common baseline applying to her situation on the other—maybe in addition, some intermediate level changes, shared with close collaborators, on top of the aforementioned baseline.

The second requirement comes from the fact that config specs are not themselves versioned. Since they are not managed, they shouldn't change. Optimally again, the user shouldn't have to modify her config spec in the following two most common scenarios (otherwise, this would be redundant and distracting):

- As the common baseline (again, specific to her situation) would be updated
- As she would herself deliver her work

Getting back to the example of the *useful* config spec from the Chapter 2:

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

Here, the fourth line (`element * LABEL`) represents the common baseline, that is, the set of versions labeled with `LABEL`. In case this baseline gets updated (e.g. the label `LABEL` is moved to different versions after a bug fix), the changes will be reflected instantly in the user's view, and the new versions will be selected.

The user's own changes are done in the branch named `branch`, and selected here with the second line (`element * .../branch/LATEST`) specifying that the user wants to select her own changes first. The third line (`mkbranch branch`) is about driving checkouts to branch off the versions carrying `LABEL` labels, or from `/main/0` on elements where there are no such labels.

The user can designate (they are in fact already public, even if not yet delivered) her changes by applying her own user-specific label, say `ANN_BUGFIX2.1`, to the full set of versions she wants to bind together. This label can be used for verification or integration purposes either by herself or by others. Note that this does not have to affect the user's config spec.

Of course, she would want to switch to a different config spec if her situation would change, e.g. to subscribe to a different baseline. In this case it would be enough to change the fourth line in the above config spec to `element * NEWLABEL`, where `NEWLABEL` refers to a different set of versions.

Also, while debugging a difficult issue, she might want extra stability, and thus to be protected even from normal updates. She might do this by switching to a special config spec, which could be the following:

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

Here `MYLABEL` is the user's own baseline label, which she is sure will not be modified by anyone except herself.

A more complex config spec for a similar purpose (e.g. debugging one's own changes, perhaps with the intention of discarding the temporary changes put in place for the time of debugging) could look like this:

```
element * CHECKEDOUT
element * .../tempbranch/LATEST
mkbranch tempbranch
element * ANN_BUGFIX2.1
element * MYLABEL
element * /main/0
```

Here on top of foundation baseline `MYLABEL`, the own published changes are selected (labeled with `ANN_BUGFIX2.1` label) and branched-off to a different own branch `tempbranch`.

Floating and fixed labels

The issue of stability and volatility is thus essential. Managing changes is managing their propagation, and keeping stable the environment used to manage it.

A simple convention concerning labels conveniently addresses this need: label application may use the `-replace` flag, in which case the labels may be moved from existing versions to new ones. Labels using this feature should clearly be distinguished from others: they are customarily called **floating**, whereas the others, the stable ones, are referred to as **fixed**. More generally, fixed labels should thus offer a guarantee of stability. Floating labels too offer one essential element of stability: their name can be seen as a handle to up-to-date, and therefore evolving, reality.

Floating labels make it possible for config specs to be stable. Consider the following config spec:

```
element * CHECKEDOUT
element * .../mybranch/LATEST
mkbranch mybranch
element * TOOLS
```

Here, `TOOLS` is a floating label. It is actually applied so that it points to the current version of the particular tool(s) in use. For example, one may have several versions of gcc compiler stored in ClearCase: version 4.4.1 labeled with `GCC_02` label and version 4.1.2, labeled as `GCC_01` (both of these are fixed labels). When the gcc version 4.1.2 was selected for the development purposes, the label `TOOLS` was applied exactly to the same element versions as those carrying `GCC_01` labels. When the newer, 4.4.1 gcc compiler version needs to be taken into use, the floating label `TOOLS` will be moved so that it applies exactly to the same element versions as those labeled with `GCC_02`. The developers' config spec will not change.

It is often a good idea to conventionally bind a floating label to a family of fixed ones. This is most naturally achieved by sharing a common prefix. In the example above, we might have a `GCC` floating label, designating the current recommended version of the Gnu compiler as well as a `TOOLS_1.27` fixed label, contributing to keep track of the position of `TOOLS` at a given point in time, across a consistent set of tools.

Note also that we choose to keep part of the name (the running number, `_01` and `_02` in the example) of the label type free from any predefined meaning (such as `_4.1.2`). The idea is that it is difficult to foresee the future, and that later one might have to import a new, different release of the same version of gcc (for a different platform, or itself using another tool, or whatever). It is much better to put detailed information in the comments of the type than in its name: updating this information if the need arises will be easier than changing the name.

It is important to note that the floating label based config specs we have been promoting here cannot be used to reproduce a precise event (for example a certain build or test run): they are not meant for this purpose. As we saw in *Chapter 3*, *Build Auditing and Avoidance*, an event should be recorded in a derived object hierarchy. The top of this hierarchy may be used to apply labels, and we are naturally speaking here of fixed labels (at least at first: in any case, we intend to easily reconstruct the full baselines which the config records constituted). These labels will be replicated to the other sites, and thus make the event reproducible across MultiSite boundaries.

Here are some hints on how to apply fixed labels.

Suppose, we would like to publish all the element versions that are needed in order to produce a certain result, specified as a build target (this may be a fix, enhancement, whole component or system, etc).

To apply a fixed label:

- Check that there are no issues with the config record (see again Chapter 3):

```
$ ct catcr -check -union build.tag
```

- Determine the vobs involved (`build.tag` is the top derived object produced by the build, refer to Chapter 3):

```
$ VOBS=$(ct catcr -flat -type d -s build.tag | grep \\.@@ | \
 perl -pe 's:/\.@@.*$::' | sort -u)
```

- Create fixed label type in every vob:

```
$ FIXED="TOOLS1.1"
$ for v in $VOBS; \
 do ct mklbtype -vob $v -c "$COMMENT" $FIXED; done
```

  Note that one may prefer to use global types, even without admin vobs—create a global type in the vob where the derived object resides, use `cptype` to create local copies in the other vobs, and explicitly create the `GlobalDefinition` hyperlinks

- Apply the label using the `build.tag` config record:

```
$ ct mklabel -con build.tag $FIXED
```

- Then in order to "select" a certain version of a tool, which bears a fixed label, we would apply a floating label on top of this fixed label as follows:

```
$ FIXED="TOOLS1.1"
$ FLOATING="TOOLS"
$ cleartool find . \
-version "lbtype($FIXED) && ! lbtype($FLOATING)" -print | \
xargs cleartool mklabel -rep $FLOATING
```

- We also need to remove a floating label from versions that do not bear the fixed label (this would be needed when changing the selected tool version):

```
$ cleartool find . \
 -element "lbtype_sub($FLOATING) && ! lbtype_sub($FIXED)" \
 -print | xargs cleartool rmlabel $FLOATING
```

Baselines and incremental labels

A fixed label can designate a **baseline** when it is **full**, that is, applied to all the elements included in a particular software configuration. The label can also be **incremental** (a.k.a. partial or sparse), i.e. applied only to the elements that have changed comparing to a certain baseline. Note that *baseline* is a keyword in UCM, and we are not speaking of it here. A baseline is a software configuration used as a reference.

Let's state for clarity that, for us, a baseline is full by definition. We take the association: "incremental baseline" as an oxymoron. A baseline may be represented by a full label, or by an aggregation of incremental ones. It is anyway a full baseline (and this is a pleonasm).

For the purpose of selecting a baseline, a floating label may be applied on top of either a full fixed label, or an aggregation of incremental ones.

For example, one may have a baseline denoted by the fixed label `REL_1.00`, which has been applied to all the versions included into the software configuration of the release 1 of the software product. Then, a floating label `REL` would be applied to all the versions already having the label `REL_1.00`, designating the currently selected release.

In case of a bug fix, the changed code base may be labeled as `REL_1.01`, including only the versions changed for this particular fix (incremental label).

Proceeding further with bug fixing activities, another incremental label, `REL_1.02`, can be created, and so on.

In order to aggregate the incremental labels, one could use either of two options. The first is to set the following config spec:

```
element * REL_1.02
element * REL_1.01
element * REL_1.00
```

The other option is to move the label `REL`, first to all the versions labeled as `REL_1.01`, and then further, to all the versions labeled as `REL_1.02`.

Then, the integration view config spec would contain a single line:

```
element * REL
```

The alternative to using incremental labels would be to make a full fixed label recursively applied to the new code base. This would create a new baseline, `REL_1.03`, including all the fixes on top of the original release 1.

Recursively applying labels is however an expensive, time-consuming, operation. Moving a floating over a subset of changed versions out of a large configuration is often far lighter: fewer database write operations are involved. Such a baseline, resulting from an original application of floating labels over a full configuration, followed with moving some of the labels over successive change sets, will be equivalent to one obtained by applying fixed labels over the last full configuration (neglecting the case of elements removed from the configuration at some stage): as the baseline defined in our example by the `REL` floating label, which would be equivalent to the baseline defined by the `REL_1.03` fixed label.

This notion of baseline equivalence may be exploited to achieve a synthesis of the two valuable properties: the relative cheapness of application of the floating labels, and the possibility to use fixed labels as a record. One needs to consider a list of sparse (incremental) labels, applied at every stage to the change sets. It is easy to build a config spec with rules using the incremental labels as fall-back of each other (as the config spec containing `REL_1.02`, `REL_1.01` and `REL_1.00` rules in our example above). Such a config spec would be equivalent to the one line config spec based on the floating label alone (`REL`). Starting from the type which is on the top of the list at any given date in the past, one might reproduce any of the successive baselines the floating labels have embodied over time.

Support for this strategy is implemented in our CPAN module, ***ClearCase::Wrapper::MGi***, where the issue of removing elements from the baseline is addressed.

One might object that this is a typical example of config spec generation, and we must admit it is, but this use is exceptional, and only meant to allow as a rule to use a stable and simple config spec with the guarantee that reproducibility is not jeopardized by the gain in efficiency and convenience.

One limitation of this scheme is that this generation of equivalent config specs works only for one single floating type (which may be global, and thus span across multiple vobs): there is no simple way to interleave the rule stacks that would result from emulating several types. We do not see this limitation as constraining in practice: it only forces to consolidate one's floating labels into a single one, representing the whole baseline.

Branches and branch types

It is sometimes difficult for users to understand that branches may actually be themselves objects, distinct from the branch types. A way to convince one of this fact is to use the `chtype` and `rename` operations, and to notice how they apply respectively to branch instances and to branch types.

Using `chtype` (or `rename`) may be an option to modify the way in which a given config spec will select a version of an element: this will happen if either the initial or the new type (name) is matched in a rule:

```
$ ct ls foo bar zoo
foo@@ [no version selected]
bar@@/main/m/3                Rule: .../m/LATEST
zoo@@ [no version selected]

$ ct catcs
element * CHECKEDOUT
element * .../m/LATEST

$ ct lsvtree foo
foo@@/main/m1
foo@@/main/m1/0
foo@@/main/m1/1

$ ct lsvtree zoo
zoo@@/main/m1
zoo@@/main/m1/0
zoo@@/main/m1/1
zoo@@/main/m1/2

$ ct chtype m foo@@/m1
Changed type of branch "foo@@/m1" to "m".

$ ct ls foo bar zoo
foo@@/main/m/1                Rule: .../m/LATEST
bar@@/main/m/3                Rule: .../m/LATEST
zoo@@ [no version selected]
```

As one can see in this preceding example, it is only the *branch* `m1` of the `foo` element, which has been renamed to `m`, not the `m1` branch type itself; and the other element's (`zoo`) branch `m1` remained unaffected.

On the contrary, when we use the `rename` command, the change affects all the branches:

```
$ ct ls foo bar zoo
foo@@ [no version selected]
bar@@/main/m/3                   Rule: .../m/LATEST
zoo@@ [no version selected]
$ ct rename brtype:m1 m
cleartool: Error: Name "m" already exists.
cleartool: Error: Unable to rename branch type from "m1" to "m".
```

```
$ ct chtype m1 bar@@/m
Changed type of branch "bar@@/m" to "m1".

$ ct rmtype -f -rmall brtype:m
Removed branch type "m".

$ ct rename brtype:m1 m
Renamed branch type "m1" to "m".

$ ct ls foo bar zoo
foo@@/main/m/1                  Rule: .../m/LATEST
bar@@/main/m/3                  Rule: .../m/LATEST
zoo@@/main/m/2                  Rule: .../m/LATEST
```

Note that we needed to remove the existing `m` type first (to make the rename succeed), which removed all the existing branches of this type. This is why we preserved the branch `m` of the bar element by *chtyping* it to `m1`.

We can now see that all the elements having branches of type `m1` were affected.

Let's note a couple of peculiarities pertaining to branches:

- Locking branch types affects the branches, so that the versions cannot be labeled anymore. In our opinion, this pretty much defeats the purpose of locking branch types at all
- The `multitool chmaster` command, when applied to a branch type, will also affect existing branches of this type. This is convenient but has one well-founded yet non-intuitive restriction: only if their mastership has not been changed explicitly (this applies also to `main` branches of elements created with the `-master` option)
- One can remove an element's branch being any one of the branch creator, the element owner or the vob owner, or root. The branch creator can remove only his own branch, provided it has no subbranches created by other users, and if no version on this branch carries a locked label. The element owner can remove any branch, unless some version on the branch is protected by a locked label:

```
$ ct des -fmt "%[owner]p\n" foo
sam

$ ct lsvtree foo
foo@@/main
foo@@/main/0
foo@@/main/aa
foo@@/main/aa/0
foo@@/main/aa/bb
foo@@/main/aa/bb/0
```

```
$ ct des foo@@/main/aa
branch "foo@@/main/aa"
  created 2010-08-01T08:59:59+02:00 by Name=mary

$ ct des foo@@/main/aa/bb
branch "foo@@/main/aa/bb"
  created 2010-08-01T09:03:38+02:00 by Name=joe


$ sudo -u joe cleartool mklabel J foo@@/main/aa/bb/0
Created label "J" on "foo" version "/main/aa/bb/0".
$ sudo -u joe cleartool lock lbtype:J
Locked label type "J".

$ sudo -u joe cleartool rmbranch -f foo@@/main/aa/bb
cleartool: Error: Lock on label type "J" prevents operation ######
                                                    "remove version".
cleartool: Error: Only VOB owner or privileged user may remove ###
              a version labeled with an instance of a locked type.
cleartool: Error: Unable to remove branch "foo@@/main/aa/bb".

$  sudo -u sam cleartool rmbranch -f foo@@/main/aa/bb
cleartool: Error: Lock on label type "J" prevents operation ######
                                                    "remove version".
cleartool: Error: Only VOB owner or privileged user may remove ###
              a version labeled with an instance of a locked type.

$ sudo -u joe cleartool unlock lbtype:J
Unlocked label type "J".

$ sudo -u sam cleartool rmbranch –f foo@@/main/aa/bb
Removed branch "foo@@/main/aa/bb".
```

Delivery

We believe to follow the common use in distinguishing *delivery* from *release*, by considering that the latter involves an additional *packaging* step, implying further extraction and installation. A similar procedure may be necessary if some level of testing in mandated to take place in an environment where the SCM system is not available.

The developer works in her branches, checks in and out her code as she needs to produce save-points, builds and tests her results. At some point, she has something to offer to others. Her delivery will result in updating a public baseline. It is essential that:

- What the user publishes matches exactly what she just tested
- The procedure is trivial and cheap, which guarantees she will use it often and with small increments
- The procedure may apply in cascade to various degrees of integration
- Different developers do not block each other and do not waste time in synchronization
- Concurrent publications do not obliterate each other (losing with one contribution what the previous brought in)
- Derived objects built prior to the delivery are not obsoleted by it. On the contrary, they are offered for sharing, thus avoiding the race condition that otherwise takes place for the creation of the first DOs matching the new delivery. Such derived objects also remain valid for config record comparison, in case a problem is found and one needs to assess why it wasn't detected by previous testing
- The procedure is reversible, so that in case of error, the delivery can be withdrawn (with no loss of information) and analyzed by the developer, in preparation for a new fixed delivery.

These requirements plead for an in-place delivery, with no modification of data: only a change of status. This is typically what labeling offers.

Note how the model of delivering by merging to integration branches fails to achieve these requirements. Merges produce new versions, which invalidates the existing derived objects. The process may attempt to compensate for this loss by requiring a build to take place as part of the delivery, but this hugely raises its cost (for example in time) and introduces new intermediate points of failure. Reversibility at this point may only be achieved if the build uses checked out versions, by unchecking them out. But this loses the information needed to analyse the cause of the failure. In any case, the identity of the delivered changes cannot be guaranteed by the system since distinct versions were produced. Once the initial strategic error is made, there are only bad solutions: fixing any one aspect unescapably creates a new problem.
One problem we must acknowledge is that delivering by labeling doesn't solve the problem of offering derived objects as part of the baseline update over MultiSite. This is an issue for which we do not have an elegant and simple solution. The best we can offer it the recommendation to stick to local labels in usual config specs. Thus follow a remote delivery by first producing a build using it, and then updating the local baseline to use this build.

Ironically, in spite of the similarities between the label and branch concepts mentioned earlier, people tend for historical reasons to think of integration as merging everything to a single branch. Using this branch type in the config spec is even considered safe and clear. Label-based config specs on the contrary are seldom used for integration purposes and often neglected. The obvious duality between labels and branches is often lost: users are surprised when reminded that labels are sufficient to select versions, independently from their position in the version tree. Select a baseline with integration labels, and you don't need integration branches: it is irrelevant where the labels are found in the version trees.

One of the above requirements, though, requires special attention: ensuring continuity with other developers' contributions. This implies that the delivery labels are not moved blindly from the versions to which they are currently applied, but only from direct ancestors of the new versions. In practice, this requires what UCM terms a *rebase* operation, and we shall name, in order to avoid confusion with the `rebase` *command* (which we cannot use outside of UCM), a **home merge**. We shall deal with this in our next chapter.

Archiving

As mentioned earlier, the config spec of the developer authoring the delivery shouldn't have to be modified. Again, by delivery here we mean **publishing in-place** *by labeling*. For example, the user has labeled her deliveries with fixed `ANN_BUGFIX_1` and `ANN_BUGFIX_2` labels, and also applied a floating "integration" label named `ANN_BUGFIX`.

Until the delivery, the new versions were selected either with branch-based rules in the developer's config spec (`mybranch` in the next example), or with previous delivery labels (`REL`). There might also be pre-delivery labels. This would be the case if team work was integrated in steps. For example, the developer could be using the following config spec (building on one already described in the beginning of this chapter):

```
element * CHECKEDOUT
element * .../mybranch/LATEST
mkbranch mybranch
element * ANN_BUGFIX
element * REL
element * /main/0
```

The use of a personal label is possible, on top of the team pre-delivery one, and even useful to keep track of correspondences between different elements, and to communicate between different views the user may have.

It is at any rate essential that the developer starts using the delivery herself, and doesn't stay caught selecting her own development versions. With the in-place delivery, there is no difference at first sight, but there might soon arise some, as other developers start delivering further improvements.

Our solution is to *archive*, as a part of the delivery, one's branches and floating labels, referenced in the user's config spec, and to do this via renaming.

The branch type `mybranch` can be archived when `ANN_BUGFIX` label application is complete and needs to be tested properly:

```
$ ct rename brtype:mybranch mybranch-001
$ ct mkbrtype -nc mybranch
```

The label type `ANN_BUGFIX` can be archived when user's published changes have been accepted and the floating label REL has been moved over it:

```
$ ct rename lbtype:ANN_BUGFIX ANN_BUGFIX-001
$ ct lock -obs lbtype:ANN_BUGFIX-001
$ ct mklbtype -nc ANN_BUGFIX
```

This way the user's config spec need not change after the delivery: the `mybranch` and `ANN_BUGFIX` (and possible pre-delivery) types are again free for development purposes. Contrast this to keeping the names unchanged and modifying the config spec instead. In this latter and common case:

- All the config specs that use the types need to be updated synchronously
- There is no support for backup and history of these changes

This is something that may be scripted (and again, is already supported by our CPAN module ***ClearCase::Wrapper::MGi***).

The binding between the branch type and the label type may be interpreted as the creation of a higher-level concept, not far from *activity* or *stream*: the stream completes at the delivery.

Rollback

The possibility to rollback one's changes is offered by the fact that the delivery was exclusively a labeling (considering the eventual home merge as part of pre-delivery development, and indeed, this one might legitimately have been followed with build and test recording). As such, it is trivially reversible.

Of course, the situation gets slightly more complex as soon as further deliveries have happened on top of the one, which its author wishes to roll back. We'll see in the next chapter (because reconstructing consistent versions will indeed involve some merging) that removing an intermediate step gets facilitated by the fact that it maps to a distinct set of branches.

Use of locking

Locking is often thought as a final operation, meant to conclude changes, making further modifications impossible.

We see this as highly paradoxical: locking is a management operation and management is needed to allow change. Avoiding changes is an act of control, and thus, quite the opposite.

So, in fact, the more one needs to change things, the more one wants to lock the label types. If there was no need for any change, no locking would be needed.

After all the labels have been applied as needed, it is advisable to lock both fixed and floating label types.

Label types should be locked in order to prevent accidental label removal, moving, and application to wrong elements. It is important here to avoid confusing safety with security: these tools are proper to fight error, not fraud. Misusing them to counter mythical attackers easily compromises safety by artificially growing complexity, making fixes cumbersome, and alienating developers of their responsibilities.

When the floating label needs to be moved, its type must be unlocked first, the labels applied, and the type locked again.

Locking to manage change is a way to be faithful to the *static typing* philosophy stated at the beginning of this chapter: unlocking a floating label before moving it parallels creating the type before applying it the first time, and therefore restores the first step required to allow for consistency checking.

In addition, locking provides an event recorded in the history of the label type, and thus with a time stamp and information about the user and the context. This event marks the end of the label application.

To lock a label type, one can use the plain `cleartool lock` command:

```
$ ct lock lbtype:L1
Locked label type "L1".

$ ct lstype -kind lbtype | grep L1
--07-06T21:15 ann label type "L1" (locked)

$ ct des -fmt "%[locked]p\n" lbtype:L1
locked
```

One can also use the `cleartool lock -obsolete` command, to lock it and mark as *obsolete*:

```
$ ct lock -obs lbtype:L2
Locked label type "L2".
```

Obsolete label types are not listed by default by the `cleartool lstype` command, unless `-obsolete` flag is specified, which makes `lstype` queries slightly more efficient and less verbose:

```
$ ct lstype -kind lbtype | grep L2
$ ct lstype -obs -kind lbtype | grep L2
--07-07T22:34 ann label type "L2" (obsolete)

$ ct des -fmt "%[locked]p\n" lbtype:L2
obsolete
```

To obsolete the already locked label, one can specify the `-rep` and `-obs` flags to the lock command:

```
$ ct lock -rep -obs lbtype:L1
Locked label type "L1".

$ ct des -fmt "%[locked]p\n" lbtype:L1
obsolete
```

To unlock the label type (also obsolete one), use the `unlock` command:

```
$ ct unlock lbtype:L1
Unlocked label type "L1".

$ ct des -fmt "%[locked]p\n" lbtype:L1
unlocked
```

Locking an object is possible only for its owner (or to an administrator). This may easily become a nuisance, and may thus requires scripting (the use of `sudo`, `suid` scripts, `ssh`, or `runas` on Windows), for example, to authorize group members to lock/unlock each others' types.

Obsoleting types, under ClearCase, has yet another distinct and loosely related use: obsolete locks are the only ones replicated, as the effects of locking are otherwise redundant with these of mastership.

Let's note that one cannot change the mastership of locked objects, which is often a good thing as such: it is better to leave these objects locked (and thus mastered in their original replica), than to unlock them, `chmaster` them, to lock them back in the new replica (losing part of the original history for no clear benefit). We may further note that may be surprisingly the mastership of objects locked obsolete may be changed.

Types as handles for information

A last note on types is that they often constitute convenient objects to which to attach information to be shared between files.

This is trivially true of comments, but also of attributes and hyperlinks.

We'll give examples of such uses in the *Chapter 9*, *Secondary Metadata*.

Summary—wrapping up of recommended conventions

We announced that our review of constraints would give rise to recommendations of conventions, and it did: we made a good deal of suggestions that go against some traditional *best practices*. Let's mention that we do *not* believe these recommendations would only be suitable for specific teams, environments, or levels of expertise. On the contrary, they are designed to be scalable, generic, and consistent, which doesn't mean they would constitute a silver bullet or cannot be further enhanced. Here is a small summary, and it covers the essential aspects we examined in this chapter:

- Checkout exclusively in (private) branches, of one single and stable type
- Use branch rules in your config spec of only this type
- Apply labels in pairs: a full floating label and an incremental fixed one
- Use only the floating label in your development config specs
- Deliver exclusively by applying labels thus in place, and archive your development branches away
- This way, keep your config spec stable
- Prior to delivering, *rebase* where the baseline has moved after you branched off it
- Lock your labels and unlock the floating ones before moving them