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

12
Challenges

The unique features of clearmake we have been describing so far, and to which we grant so much value, fall short in at least two extremely significant contexts: *Java* and *MultiSite*. Our goal in this chapter will be to analyze the causes of the problems, to examine the existing solutions, and to propose directions for extensions. In the end, we'll also briefly consider a few other challenges, opened by recent evolutions in software development.

Java

ClearCase was mostly (at least originally) written in C++, and its build model is well suited (with some historical adjustments to cope with templates in two generations of compilers) to development using this language. Java, although already old at the time of its wide success, broke a few significant assumptions of the model.

Problems with the Java build process

The traditional build model is stateless, and therefore easily reproducible: running the same build command in the context of static sources (*leaves* of the dependency tree, seen upwards from the products) produces the same results, but doesn't alter the context. This is not the case anymore with javac. The reason is trivial: javac integrates into the compiler a build tool function. The *compiler* reads the Java source as a build script and uses the information to build a list of dependencies, which it verifies first, using traditional time stamp comparison between the sources and class files produced, and rebuilding missing or out-dated class files. It doesn't, however, perform a thorough recursive analysis, nor attempt to validate jars, for instance.

This behavior is highly problematic from a clearmake point of view, as it results in the list of derived objects produced with a given rule (siblings of the target) being variable from one invocation to the next, and conversely, in a given derived object potentially being produced by several different rules. Both of these effects result in incorrect dependency analysis, and in spurious invalidation of previous build results.

Let's note that since javac performs time stamp comparisons, the default behavior of cleartool to set the timestamp at checkin time is inadequate for Java sources, and results in needlessly invalidating classes produced before checkin: set up a special element type manager defaulting to the `-ptime` (preserve time) checkin option.

The second traditional assumption broken by Java is a practical one: the language has been designed to optimize compilation speed, which means that build time stops being a primary issue. This is of course obtained by using a single target, the Java virtual machine, and at the expense of run-time performance; but history has already clearly validated this choice, in the context of favorable progresses in hardware.

This is obviously not a problem in itself, but it had two clear consequences:

- The winkin behavior of clearmake cannot be *sold* to users anymore on the argument of its saving build time (by side-effect). As we know, the argument of the accuracy of management appeals only to users having experienced its importance, and *reward following investment* is a known recipe for failure in evolution.
- It encourages carelessness among developers: throw away (clean) and start again from scratch.

Of course, the gain in speed is mostly felt in small configurations, and at the beginning of projects: this strategy doesn't scale, as the total build time still depends on the overall size of the component, instead of on this of the increment (the number of modified files). It is however often late to change one's strategy when the slowness becomes noticeable.

.JAVAC support in clearmake

Support for Java was added relatively late to clearmake (with version 2003.06.00), in terms of a `.JAVAC` special target (and a `javaclasses` makefile macro). The idea (to which your authors contributed) was to use the **build audit** to produce a `.dep` file for every class, which would be considered by clearmake in the next invocation, thus giving it a chance to preempt the *javac* dependency analysis. Of course, the dependency tree would only be as good as this of the previous compile phase, but it would get refined at every step, eventually converging towards one which would satisfy even the demanding `catcr -union -check`.

Special care was needed to handle:

- Inner classes (producing several class files per java source, some of them being prefixed with the name of the enclosing class, with a dollar sign as separator—not a friendly choice for UNIX shells).
- Cycles, that is, circular references among a set of classes: a situation which clearmake could only process by considering all the set as a common target with multiple siblings.

This solution should be very satisfactory, from the point of view of ensuring correctness (consistency of the versions used), sharing of objects produced, and thus managing by differences. It should offer scalability of performance, and therefore present a breakeven point after which it would compete favorably with from scratch building strategies.

One might add that a makefile-based system is likely to integrate with systems building components written in other languages (such as C/C++), as well as with performing other tasks than compiling Java code.

Let us demonstrate how the dependency analysis and derived objects reuse are working using the `.JAVAC` target in the makefiles, testing exactly the aspects mentioned above—inner classes and cycles.

In our small example, `Main.java` implements the main class, `FStack.java` implements another independent class, which the `Main` class is using. Finally, the `FStack` class also contains an inner class `Enumerator`, which results after the compilation in a file of name `FStack$Enumerator.class`:

```
# Main.java
public class Main {
   public static void main(String args[]) {
      FStack s = new FStack(2);
      s.push("foo");
      s.push("bar");
   }
};

# FStack.java
public class FStack {
   Object array[];
   int top = 0;
   FStack(int fixedSizeLimit) {
      array = new Object[fixedSizeLimit];
   }
   public void push(Object item) {
      array[top++] = item;
   }
```

```
      public boolean isEmpty() {
         return top == 0;
      }
      public class Enumerator implements java.util.Enumeration {
         int count = top;
         public boolean hasMoreElements() {
            return count > 0;
         }
         public Object nextElement() {
            return array[--count];
         }
      }
      public java.util.Enumeration elements() {
         return new Enumerator();
      }
   }
```

We create a tiny `Makefile` making use of the `.JAVAC` target. Note that we do not have to describe any dependencies manually; we just mention the main target `Main.class`, leaving the rest to the javac and the ClearCase Java build auditing:

```
# Makefile
.JAVAC:
.SUFFIXES: .java .class
.java.class:
        rm -f $@
        $(JAVAC) $(JFLAGS) $<
all: /vob/jbuild/Main.class
```

The first run of the clearmake does not look very spectacular: it just executes the javac compiler, submitting the `Main.java` source to it, and all the three class files (`FStack.class`, `FStack$Enumerator.class`, and `Main.class`) get generated. The same would have been produced if we used the "default" Makefile (the same, but without the `.JAVAC` target):

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note though that one thing looks different from the default Makefile execution: our "`.JAVAC`" `Makefile` produces the following dependency (`.dep`) files:

```
$ ll *.dep
-rw-r--r--  1 joe jgroup 654 Oct 19 14:45 FStack.class.dep
-rw-r--r--  1 joe jgroup 514 Oct 19 14:45 Main.class.dep
```

But their contents are somewhat puzzling at the moment:

```
$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has not been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=true />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                                  classes: -->
<target name=/vob/jbuild/Main.class path=/vob/jbuild/Main.class />
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ###############
                                    FStack$Enumerator.class inner=true />

$ cat Main.class.dep
<!-- Main.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/Main.class conservative=false />
<mysource path=/vob/jbuild/Main.java />
<!-- Target /vob/jbuild/Main.class depends upon the following ###########
                                                                  classes: -->
<target name=/vob/jbuild/FStack.class path=/vob/jbuild/ #################
                                       FStack.class precotarget=false />
```

So, it looks as if the `FStack` class was depending on the `Main` class, and the other way around as well. But that's what one can only figure out after a single javac execution—The `Main` class was produced and, in order to compile it, two more classes were needed: `FStack` and `FStack$Enumerator`.

But we can do better. Let's try the second subsequent clearmake execution, without any real changes (for our purpose: in a real work scenario, a new build would of course be motivated by a need to test some changes). It does not yield `all is up to date`, as one would expect when using the default Makefile, but instead it does something interesting:

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note that it does not even execute the default script, but rather some other one (`/usr/bin/javac /vob/jbuild/FStack.java`). Where did it come from? Actually from the `FStack.class.dep` dependency file mentioned above. And what about the dependency files themselves?-They have somewhat changed:

```
$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=false />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                          classes: -->
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ###############
                                    FStack$Enumerator.class inner=true />

$ cat Main.class.dep
<!-- Main.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/Main.class conservative=false />
<mysource path=/vob/jbuild/Main.java />
<!-- Target /vob/jbuild/Main.class depends upon the following ###########
                                                            classes: -->
<target name=/vob/jbuild/FStack.class path=/vob/jbuild/ #################
                                       FStack.class precotarget=false />
```

And now this looks right! The `FStack` class depends on `FStack$Enumerator`, but it does not depend on the `Main` class, and this is noted in the modified `FStack.class.dep`. The `Main` class, on the other hand, does depend on `FStack`, and that is stated correctly in `Main.class.dep`.

Now, if we try to run clearmake once again, it yields `'all' is up to date`:

```
$ clearmake -f Makefile
'all' is up to date.
```

But this time it means that all the dependencies have been analyzed and recorded in the `dep` files.

Let's try to rebuild. Removing either `FStack.class` or `FStack$Enumerator.class` causes its rebuild, which triggers in turn a rebuild of the dependent `Main.class`:

```
$ rm FStack$Enumerator.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

Note that with the default makefile (which does not have any dependencies recorded manually), such a removal would not invalidate the build and `'all' is up to date` would have been yielded.

And even an unconditional rebuild goes differently than the first time clearmake execution. The main target gets built last, and before it, the other independent targets get executed:

```
$ clearmake -f Makefile -u
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ rm Main.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java
```

And what about the dependency files? Have they changed? Not this time. As long as we do not change the code, the dependencies remain the same and the `dep` files are not affected by the rebuild.

Using a second view, we can now demonstrate the winkin:

```
$ ct setview view2
$ clearmake -f Makefile
Wink in derived object "Main.class.dep"
Wink in derived object "FStack.class.dep"
Wink in derived object "FStack$Enumerator.class"
Wink in derived object "FStack.class"
Wink in derived object "Main.class"
```

Let's now introduce an artificial circular dependency between the `FStack` and `Main` classes, for example by adding the following line to the `FStack` class constructor:

```
Main m = new Main();
```

We see that after a couple of clearmake executions, the new dependencies get resolved and recorded in `FStack.class.dep`:

```
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java

  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
'all' is up to date.

$ cat FStack.class.dep
<!-- FStack.class.dep generated by clearmake, DO NOT EDIT. -->
<version value=1 />
<!-- (A build of this target has been directly audited.) -->
<mytarget name=/vob/jbuild/FStack.class conservative=false />
<mysource path=/vob/jbuild/FStack.java />
<!-- Target /vob/jbuild/FStack.class depends upon the following #########
                                                              classes: -->
<cotarget name=/vob/jbuild/FStack.class path=/vob/jbuild/ ###############
                                      FStack$Enumerator.class inner=true />
<target name=/vob/jbuild/Main.class path=/vob/jbuild/ ###################
                                        Main.class precotarget=true />
<cotarget name=/vob/jbuild/Main.class path=/vob/jbuild/ #################
                                               Main.class inner=false />
```

Now, `FStack` depends on `Main`, and `Main` depends on `FStack`, and invalidating either of them would cause a rebuild of the other:

```
$ rm FStack.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
  rm -f /vob/jbuild/Main.class
  /usr/bin/javac  /vob/jbuild/Main.java

$ clearmake -f Makefile
'all' is up to date.
$ rm Main.class
$ clearmake -f Makefile
  rm -f /vob/jbuild/FStack.class
  /usr/bin/javac  /vob/jbuild/FStack.java
```

```
rm -f /vob/jbuild/Main.class
/usr/bin/javac  /vob/jbuild/Main.java
```

This scheme supports some level of sophistication such as maintaining one single makefile for several source directories, and exporting the classes to a build hierarchy. One may use for this purpose the `javaclasses` macro, including while using a distinct deployment directory:

```
%.class : %.java
    rm - f $@
    $(JAVAC) $(JFLAGS) -d $(BUILD) $<

COM = org/wonderland
SROOT = $(SRC)/$(COM)
CROOT = $(BUILD)/$(COM)
CLASSES = $(javaclasses $(CROOT)/dir1, $(SROOT)/dir1) \
    $(javaclasses $(CROOT)/dir2, $(SROOT)/dir2) \
    $(javaclasses $(CROOT)/dir3, $(SROOT)/dir3)

all: $(CLASSES)
```

Note however that this macro is only mechanically computed from the contents of the source directories and cannot reliably be used to name all the classes collected in a *jar*: it would miss the inner classes.

Also, it may in fact catch files that would actually not be needed. It is thus not obvious whether using it is actually better than relying as in our first case, on naming one (or some) main targets and relying upon the dependency analysis to find the others!

The use of a deployment directory (used with the `-d` option of javac) makes it easy to create a jar from what was actually produced.

This `.JAVAC` support seems quite usable, even if we lack a real size experience. All the infancy problems that were reported at some stage have been satisfactorily addressed.

Unfortunately, this solution has not been widely used in practice due to the following reasons:

- It came too late.
- It performs comparably poorly in small configurations, because it invokes javac once per target; javac starts a Java virtual machine and the cost of initializing and finalizing it overshadows the build cost proper: it is much more efficient to invoke javac once for a large set of files.
- It departs from the main stream practices, and appeals to competences not typically shared by Java developers (makefiles).

These are however to some extent only speculations, based upon assumptions more than on measurements. The performance penalty alluded to above may actually be overstated...

The third bullet leads us naturally to the next sections: both *ant* and *maven* come with off-the-shelf support for producing the other kinds of deliverables (*jar*, *war*, *ear*... and so on) common to web development, as well as for deploying them to the various application servers (*WebSphere*, *tomcat*,... and so on) and the various repositories available, under *Eclipse* or *OSGi*. Nothing inherently impossible to do with makefiles, only no Open Source project or consortium maintains and offers such a support.

Ant and XML

The *main stream* tool for building Java code is *ant*, although it is getting challenged by *maven*. Neither is actually deeply concerned with the issues of consistency of the build products: they typically delegate the dependency traversal, at the local level, as well as the compilation to javac tasks in the Java virtual machine, and rely for the rest on performing a *clean*. Their goal is to supplement javac in generating sources from *idl*, handling multiple components and their inter-dependencies, packaging, deploying, and so on.

The efficiency of ant comes from the reason we described earlier: ant is itself written in Java—invoking it results in only one initialization of the java virtual machine. In theory, one could intercept the dependency analysis between build jobs. There is even a mechanism which may be used for this purpose: *listeners*.

After promising a *clearant* for a few years, ClearCase came with an `ant_ccase` man page describing how to audit Ant builds, using `CCAudit.jar`. No winkin is supported so far:

```
$ ant -listener CCAudits ...
```

This is of course a way to produce config records that might be analyzed with `catcr -union -check`. Will the evidence produced be sufficient to convince users to switch back from their *ant* build system to a clearmake / `.JAVAC` one? We recommend our readers to test their builds: we are used to raising surprise among users! The recipe is straightforward and non-invasive:

```
$ export ###################################################################
       CLASSPATH=/opt/rational/clearcase/java/lib/CCAudits.jar:$CLASSPATH
$ export ###################################################################
    LD_LIBRARY_PATH=/opt/rational/clearcase/linux_x86/shlib:$LIBRARY_PATH
$ ant -listener CCAudits -buildfile build.xml
```

In most of the cases, the cause of the surprise is incomplete cleaning (for example, of generated Java sources). Not auditing one's builds moves some responsibility back from the tool to the developers, which is error-prone. Remember that barring auditing, all the dependency analysis must be specific to the tool chain.

But we should note that the dependencies produced are not (at all) fine grained: basically all of the derived objects share one common (bulky) config record, as with the first run the `.JAVAC` clearmake build mentioned above. But unfortunately, in Ant build auditing there is no way to improve it.

We can again only speculate about the reasons why *clearant* didn't make it as promised (i.e. fully supporting *winkin*). One might conjecture that it wasn't so easy to beat the results achieved with clearmake and the `.JAVAC` mechanism.

In other words, the existing support is actually better than commonly thought, and originally anticipated.

Audited Objects

The resolution of our problems may come from a recent Open Source product: ***Audited Objects***. This offers build auditing without ClearCase, that is, using user level system call interception and an SQL database.

The huge interest there is that one may hope to transfer some functionality (competence, concerns) from ClearCase, to e.g. *git*.

MultiSite

We have already seen a few of the most obvious limitations of *ClearCase MultiSite*: since it doesn't replicate config records, i.e. the information about the successful production of derived objects, using well identified resources, MultiSite doesn't really allow to distribute a development environment. While crossing site boundaries, one loses the most interesting part of the information. The interactions between developers on the same site are much richer than the ones across the MultiSite boundary. This may feel acceptable in the context of a top-down process of work division and assignment of tasks, but it would be a limitation for a bottom-up process of managing peer contributions, which is closer to this of Open Source development.

We showed how to work around some of these limitations by using labels, but this obviously requires user attention, and manual action.

We also mentioned the problems of clashes between the names of types created in different replicas, within the windows of opportunity left open by the delays in synchronization. Resolving these problems in a satisfactory way would require structuring the namespace of types, or in fact allow to *version* types (group the types in sets, considering them as representatives thereof).

Some recent products bring interesting perspectives on these issues. Here are a few.

Maven, and Buckminster

We already mentioned *maven*. The most interesting conceptual novelty in this product, in spite of its reputation for efficient template processing, may be even more strongly demonstrated in *Buckminster*. Both systems identify certain build products in a way independent from the site in which they were originally produced—they allow one to alternatively download binaries *or* produce them locally: what matters is the result.

Of course these systems use crude naming conventions as the primary means of identification (which checksums in addition), and ClearCase is far more robust and flexible in this respect.

Mercurial and git

Distributed management has known a recent favor with *Mercurial* and *git* (and some other less widely used: *svk*, *Gnu arch* to name a few). These offer an alternative to the centralized model of *subversion* (and CVS) and to a paradox this model leads to—it resorts to *committers* applying patches, that is, to a process completely outside subversion itself!

All these products support a *pull* (*get*) instead of the *push* model of ClearCase MultiSite. Once again, this means that information about interesting versions to pull has to be conveyed by means external to the SCM system: users must get convinced to make their pulling efforts by other means than evidence supported by the system.

These considerations should remind us that ClearCase was a forerunner in this category as well (*distributed* management), and that it is still a technological leader for those who want to use it.

However, here as well as for the continuity of management by auditing, some obvious limitations would need to be lifted for it to support tens of thousands of replicas. All replicas cannot be flatly exposed: some structure is badly needed.

In this respect, one must admit that both Mercurial and git are far superior to ClearCase: they easily support hundreds of sites, which may or may not be shared among communities of users. ClearCase developers came too hastily with the short-sighted concept of snapshot views, when laptops flooded their customer base. They missed the opportunity to extend MultiSite towards facing the challenge.

Perspectives in Software Engineering

Eclipse and OSGI

The *eclipse* repository stores multiple versions of the same jars, and offers some management of dependencies between user applications. This allows to propagate changes gradually, instead of upgrading all the applications depending on a common resource synchronously; it even allows to leave some applications as such, if they do not benefit from the changes.

This displays a radical novelty: the customer environment, under which one delivers, is not *homogeneous* and *consistent* anymore—delivery happens *under* SCM!

To be honest, this had always existed in UNIX, with shared libraries (see *Chapter 8, Tools Maintenance*). Using symbolic links, one could decouple the current (latest) version of a shared library, for linking purposes, from the ones actually bound to existing applications, which could thus use different versions and did not need to be updated every time a new version of the shared library was published.

Now, the support in eclipse clearly shows an increase in such uses.

One challenge may be to simulate this under ClearCase in the same view, for example, with hard links and pattern-based config spec rules (which we mentioned already, while speaking of *config specs* in *Chapter 2, Presentation of ClearCase*). But one could take a wider perspective and consider that different views offer a much more flexible and robust support, and that the need for an SCM system is in fact much wider than might have been anticipated. One might want to see the eclipse repository as a first case of *delivering under SCM*, into an SCM system itself deployed in the customer environment. What is missing there is a *standard* or rather an open specification, to protect customers from being coupled to one single vendor.

The only existing attempt to standardize versioning interfaces is *WebDAV*, the extension of the HTTP protocol, geared towards authoring, that is, towards the other end of the configuration space (or dependency tree).

Virtual machines

ClearCase virtual file system, with its dynamic views, was a foregoer, but has its own limitations: some resources cannot be versioned (such as sockets, or any interfaces on which one and only one protocol is being used). This problem was first met with versioning shared libraries used at boot time.

An interesting trend is the use of virtual machines to split the resources of one host, protecting the users of a slice from what happens in another. Or other way around: cloud computing, federating multiple physical hosts to offer a virtual environment.

ClearCase has already certified certain virtual machines as platforms, but isn't the real challenge the opposite? Isn't there a need to run multiple virtual machines under some kind of software configuration management?

The challenge here would thus be to extend dynamic views to the status of full-blown virtual machines.

Conclusion

This chapter ended up posing more questions than it offered answers.

What it should convince us of is that there is still a need to develop ClearCase, and not to drop the experience specifically gained with this wonderful tool, being far from the main streams of tradition. That one's investment in ClearCase is not lost, even if it doesn't always seem directly usable in the contexts of its competitors.