BEA Maven Requirement Documents

BEA Maven Requirements

Dan Fabulich
Senior Software Engineer
BEA Systems, Inc.
Business Interaction Division

This document is long and terse; I apologize for that.

I've divided it up into four sections: one section of Definitions, one section of Use Cases, and two "Requirement Chains".

The section on Definitions is very important, because it attempts to discuss SCM "version numbers" while being totally agnostic as to whether CVS, SVN or P4 is being used under the hood. I've made up a few terms in an effort to remain agnostic; I want to especially discourage anyone from using the term "Revision Number" while discussing this document, because each provider uses that term differently.

After that, I define four uses cases. Each of them is followed by a "Failure Use Case".

Finally, there are two "Requirement Chains". The Requirement Chains are so-called because they discuss multiple requirements that *flow from one another*. If earlier Requirements aren't satisfied, later Requirements are irrelevant. If *only* earlier Requirements are satisfied, later Requirements become more important; in some cases it would even be *harmful* to satisfy earlier Requirements without satisfying later Requirements.

Also, I've been very explicit about the logic that I used to come up with these requirements. That's because I want to make sure we can all see *wh y* we want these things. My claim is that if you accept the top-level Requirements and Assertions, you'll be forced to accept the Derived Requirements below, and you'll also know why they're necessary.

Finally, as a note, these are not the *only* requirements BEA needs from Maven. There's still other stuff not mentioned here ("who depends on me?" reports, VStudio integration, dependency conflict reports) that don't require an elaborate requirements document to explain/describe, and won't necessitate any changes to the Maven core or Continuum. (Some of them are already on the Maven roadmap.) Still, issues like these are very important to us; even if all of the issues in this doc were fixed today, we'd still have quite a bit of work ahead of us before we could actually *us e* Maven for everything we do.

OK, on with the show!

Definitions

SCM - Source Configuration Management. We have to do this so we can keep track of our source files.

SCM Provider - CVS, SVN, Perforce, ClearCase, etc. Each of these is a different provider of SCM. (You could call them vendors, but nobody really "vends" CVS.)

SCM System - One particular SCM machine (or a cluster of machines that behave as if they were one machine) running some SCM software.

Consumer - A Maven project that "consumes" some Producer as a dependency.
Producer - A Maven project that provides code for a Consumer to use.

Maven Version Number - The number that appears in the /project/version tag in some project's pom.xml file.

Checked-in Maven Version Number - The Maven Version Number that appears in the pom.xml file that appears in the SCM system.

Continuum Published Maven Version Number - The Maven Version Number that appears in the pom.xml file that Continuum publishes to the remote repository. (Normally this is the same as the Checked-in Maven Version Number, but these can be different if, say, Continuum changes the Maven Version Number as it is published.)

Suppose you have a directory layout like this:

```
foo
 |
 +- bar
 |   |
 |   +- One.java
 +--baz
     |
     +- Two.java
     +- Three.java
```

Single File SCM Number - A version number that applies only to a particular file under source control. Each file has its own Single File SCM Number. In the example above, One.java has its own Single File SCM Number separate from Two.java's Single File SCM Number. Check-ins that only modify One.java do not change the Single File SCM number of Two.java, and vice-versa.

Global SCM Number - A version number that applies to the entire SCM system at once. There should be only one Global SCM Number at a time for any given SCM system. (In CVS, this version number can simply be the date of the last check-in.)

Directory Specific SCM Number - A version number that applies only to a particular directory in the SCM system. In SCM systems that have Global SCM Numbers, you can find a Branch Specific SCM Number by finding the largest Global SCM Number that affected files in that directory. (In CVS, this version number can simply be the date of the last check-in in the specified directory.)

"Revision" Number (deprecated term) - Means different things across various SCM systems, and should not be used in a technical discussion that intends to remain agnostic to SCM systems. In particular, CVS *only* has Single File SCM Numbers, called "Revision" numbers. Subversion does not have Single File SCM numbers, but only has the Global SCM Number, and calls it the "Revision" number. Perforce has Single File SCM Numbers (called "Revision" numbers, just like CVS,) and it also has Global SCM Number, called a "Changelist" number.

Example Using the Terms

Let's refer to Single File SCM Numbers by reference to a file name followed by a # and a number. So One.java#4 means One.java's fourth Single File SCM Number. Global SCM Numbers will just be called "G" followed by a number, e.g. G123.

Let's say this happens:

Alice checks in One.java#1: G1
Bob checks in Two.java#1 and Three.java#1: G2
Alice checks in One.java#2: G3
Bob checks in Two.java#2: G4
Alice checks in One.java#3 and Two.java#3: G5
Bob checks in Two.java#4: G6

At this point, "bar"'s Directory Specific SCM Number is 5, because One.java was updated in G1, G3 and G5, and not since then. "baz"'s Directory Specific SCM Number is 6, because Two.java was updated in G6. (Note that Three.java never changed.)

Use Cases

Use Case 1: Do You Have My Fixes?

Alice, a QA engineer, finds and files a bug in the "P" project, whose Maven Version Number is "2.2". The developer Bob attempts to fix the bug in Foo.java. He checks some changes into the SCM system, without modifying pom.xml. The SCM system gives Bob an SCM Number (of some kind) that represents the check-in... let's call it "123".

The continuous build system automatically creates a build containing Bob's changes; the build is a success. Bob marks the bug as "Coded - Please Verify." But Alice reports that the bug is "Not Fixed."

Bob is skeptical, and asks Alice whether she tested with a binary containing check-in "123". Without using a pom.xml file or the SCM system itself, Alice examines the binary's metadata and verifies whether it contains that check-in number.

Failure Use Case: The Maven Version Number of P contains only "2.2", so P's binary metadata doesn't contain any more information than that. Alice has no way to tell whether she has the correct copy of P version "2.2".

Use Case 2: No Need to Retest

Tuesday morning, the automated continuous build system generates a build for project "P". Alice tests this project on Tuesday afternoon.

A day passes, and no one checks-in any changes to P. On Wednesday, someone kicks-off an "on demand" build of P. P builds successfully as it did on Tuesday, and the P binaries are given to Alice to test. Alice examines the binary metadata and discovers that she does not need to retest P.

Failure Use Case: P's builds are identified by "build numbers", which go up even when the source does not change. Alice is unable to tell whether this binary needs to be retested or not.

Use Case 3: Reproducible Automatic Updates

Developer Carol works on the "C" project, which consumes Bob's "P" project. Carol wishes to continously integrate with P, to make sure that she's constantly using the latest version of P.

On Tuesday, the automated continuous build system creates a build of C, which runs a series of automated tests. The build+test process passes; Carol makes no further changes to C.

On Wednesday, Bob checks-in a change to project P, which builds successfully on the automated continuous build system, but it doesn't work correctly with C. The build system automatically updates Carol's C pom.xml file, pointing at the new version of P, and then builds C using the new P.

Even though no other code changes went into C, C automatically gets a new version number, because it is using a newer version of P. The continuous build system therefore automatically builds the new C. This new version of C fails to pass the automated tests. Carol remembers that she hasn't changed C herself, so she looks in the SCM history to see what has changed in C. She sees that the automated build system has modified C to point at a newer version of P.

Carol synchronizes her source code back to the configuration it was in on Tuesday. She runs that build on her machine and finds that it works, just like it did on Tuesday. She syncs to the latest Wednesday version of C and attempts to build that. She finds that the automated tests no longer pass. Realizing what has happened, she opens a bug against Bob. Bob fixes the bug in P and then kicks off a build of P. The automated build system automatically updates C to use the new version of P, and builds C. C now builds successfully.

Finally, later in the development cycle, Carol decides that she doesn't want C to be automatically upgraded to the latest version of P. She makes a change to her pom.xml file to indicate that automatic upgrades are not desired. Now, when a new version of P is released, C remains unaffected.

Failure Use Case 1: Carol is using a generic or "soft" version number in her C pom.xml, instead of a hard literal version number. Carol's code does not change when there's a newer version of P, so the automated build system does not build a new version. Later, when Carol changes C, she finds that the build is failing, but not due to any change of her own. Carol rolls back her changes and finds that the build still doesn't work. After investigation, she finds that P was to blame, so Bob releases a new version of P. Then, C's build magically starts working again.

Failure Use Case 2: On Wednesday, C is using a newer version of P than Tuesday, but C has not changed. Alice looks at C, observes that it hasn't changed, and decides not to retest it, not realizing that there are now bugs in that version of C that weren't there on Tuesday.

Use Case 4: Windows Library Support

Bob, a developer, owns a project P that builds a Windows library file called "Z.dll"; Z.dll is consumed by another library Y.dll owned by developer Carol, which is consumed by another library X.dll, which is consumed by an executable called "Foo.exe".

Carol declares in her Y pom.xml file that she depends on Z.dll version 1.0. Later, Bob releases Z.dll version 1.1. (In this scenario, Carol is not auto-updating as in the previous case.)

Dave, a customer, calls complaining of a critical problem in Foo.exe. Eventually, the problem is transferred to Carol. Carol reproduces the problem on a test machine, and suspects that the problem may be remedied by replacing Z.dll with version 1.1. Without using Maven, Carol copies Z.dll version 1.1 directly from the remote repository and copies it into her installation directory. She finds that this fixes the problem.

Carol checks-in an update to Y's pom.xml and kicks off builds and tests. Carol finds that no regressions are introduced, so she sends Z.dll version 1.1 to Dave as a critical update, without sending a new X.dll or Foo.exe, since these files were not changed. Dave copies only Z.dll into the installation directory, and finds that the problem is fixed.

*Failure Use Case 1: *The Windows library in the Maven repository is called "Z-1.1.dll", but Y.dll declares in its linking metadata that it depends on "Z-1.0.dll". Carol kicks off a new build of Y using the new Z. Y now has a new version number, so she has to kick off a build of X to consume the new Y. But now X has a new version number, so she has to rebuild Foo.exe as well to use the new X. When she's done, she finds that this does fix the problem, but now Dave must replace not only Z.dll but also Y.dll, X.dll and Foo.exe to benefit from the fix.

Failure Use Case 2: The Windows library in the Maven repository is called "Z-1.1.dll", even though it was built as "Z.dll". Carol copies this into her installation directory, but it doesn't work, and she doesn't know why.

Use Case 5: Single-Sourced "Jumpable" Builds

At BEA, we have a facility called "J2C" which we can use to convert (or "jump") Java code into C#, which we then compile into .NET assembly DLLs. Jumpable code will thus have two sets of dependencies: one set of "Java" dependencies, and another set of ".NET" dependencies.

Normally all of the sources files are Java, though there may be some source files that are "non-jumpable", in which case you'll have three sets of source: the jumpable Java, the non-jumpable Java, and the native C#.

Bob, a developer, maintains a jumpable project, Producer P. Carol is a developer on another jumpable project, Consumer C. When Bob runs builds of P, the build automatically runs J2C to convert his Java code into C#. It then builds the Java JAR output, as well as the C# DLL output. It also runs unit tests against both the Java and the C#.

Bob then checks in his code and publishes it. When published, the C# and the Java code appear in the repository as a single project with multiple artifacts. Both of these artifacts appear in the same directory in the remote repository. They necessarily have the exact same version numbers (because they both descend from the same source code in SCM).

Carol's Consumer C depends on P in both of its forms: the Java build of C depends on the Java part of P; the .NET build of C depends on the .NET build of P. Carol declares this in her pom.xml file; she only has to declare the version number of P once, because there is just one project and just one version number. Carol's build then generates Java and .NET artifacts.

Failure Use Case 1: Bob creates a build that has multiple artifacts, but only one of them is allowed to be "primary"; he must declare the packaging (filetype) of the primary artifact explicitly. He selects the Java artifact to be primary, so P's packaging is "jar". Carol then can't add the secondary .NET artifact to her .NET classpath, because her declaration of dependency on P means that she only gets the "primary" artifact, whose packaging is "jar".

Failure Use Case 2: Bob must maintain two projects, "p-java" and "p-dotnet". Bob arranges his artifacts like this:

```
bobstuff
 |
 +- p-java
 |   |
 |   +- src/One.java
 |   +- pom.xml
 +- p-dotnet
     |
     +- pom.xml
```

The "p-dotnet" pom.xml declares that its source directory is "../p-java/src".

Bob checks in a change to One.java. This automatically changes the Directory Specific SCM Number of "p-java", but it doesn't change the Directory Specific SCM Number of "p-dotnet". Therefore "p-dotnet"'s version number isn't kept up to date with "p-java"'s version number. The build system therefore knows to automatically kick off a build of p-java, but forgets to build p-dotnet, because p-dotnet apparently hasn't changed.

Failure Use Case 3: Exactly as above in Failure Use Case 2, but somehow, the build system is smart enough to know to rebuild p-dotnet. But p-dotnet's Directory Specific SCM Number hasn't changed, so its version number doesn't change. Therefore Alice, a QA Engineer, can't tell whether or not to retest p-dotnet.

Failure Use Case 4: As above in Failure Use Case 2. An automated system automatically updates p-dotnet after Bob checks in changes to p-java. Therefore p-dotnet is automatically rebuilt with a new version number, but this version number is different from the version number of p-java. Carol must declare her dependency on Bob's project twice, with two different version numbers, even though the source is the same, doubling the reporting workload, and introducing new opportunities for errors.

Requirement Chain 1: Version Numbers and Automated Check-ins

Here comes a bunch of logic. Simple "Requirements" are requirements that I think everyone already accepts. Simple "Assertions" are statements that I think everyone already knows.

"Derived Assertions" are Assertions that follow logically from other Assertions. "Derived Requirements" are Requirements that follow logically from other Requirements or Assertions.

Consider a Consumer build project called "C" and a Producer build project called "P".

1) Requirement: P's version number must always change if its source has changed, and should not change if its source has not changed.
  1a) Assertion: "Build numbers" generated by the build engine (Continuum) increase even if the source has not changed.
  1b) Assertion: "Global SCM Numbers" increase even if the source of P has not changed.
  1c) Derived Requirement: On account of 1a and 1b, official (Continuum) builds should be identified by their Directory Specific SCM Number.
  1d) Derived Requirement: On account of 1c, Continuum should publish P's built artifacts using P's Directory Specific SCM Number (appending it to P's Checked-in Maven Version Number) when it does official builds.
2) Requirement: C's official build must be reproducible using the source of C and the build repository.
  2a) Clarification: In particular, if on Tuesday C's build succeeds but on Wednesday C's build fails, it must be because there was some change *in C* that caused the failure.
  2b) Derived Requirement: On account of 2a, I should be able to synchronize the source of C back to Tuesday's source and rebuild that; that C build should build successfully if it built successfully on Tuesday.
  2c) Derived Requirement: On account of 2a and 2b, C must always declare in C's source code that it depends on a *specific* "hard" version of P, and never use a generic or "soft" term like "LATEST" or "SNAPSHOT" in an official build.

3) Assertion: Updating C manually every time you want to use a slightly newer version of P is very tedious and error-prone.
  3a) Derived Requirement: On account of 2c and 3, it should be possible (but not required) to automatically update and check-in C to make it use a newer version of P. (These automatic updates must change C's source code to ensure that C is reproducible.)
4) Assertion: If you change C's source code during the build of C, then its Directory Specific SCM Number changes during its own build.
  4a) Assertion: Therefore, Directory Specific SCM Numbers are *vague* if check-ins happen during the build: you may need to consider the difference between the Directory Specific SCM Number when you *began* the build and the Directory Specific SCM Number when you *finished* the build.
  4b) Assertion: Since all check-ins modify the *latest* code in the repository, if C's build checks-in to C, it is impossible to rebuild old versions of C without clobbering new versions of C.
  4c) Derived Requirement: On account of 4a and 4b, C should never check-in to itself during its own build.
5) Assertion: A builder of C *must* poll the remote repository to see if there's a newer version of P; the remote repository can't/won't send a just-in-time message to clients informing them when there is new code available.
  5a) Clarification: In other words, it can't be P's job to update all consumers of P. A consumer C must check on all of its dependency declarations to see if any of them need to be updated.
  5b) Derived Requirement: On account of 4c and 5, C should (be able to) poll the remote repository for newer versions of its dependencies and check-in an updated version of its POM immediately *before* C builds (in a *pre-build* step).
6) Requirement: Single-sourced "jumpable" builds (and other builds with multiple artifacts of various filetypes) must assign the same version number to all of the derived filetypes.
6a) Assertion: If you try creating multiple POMs in multiple "sibling" directories pointing to the same source files (as in Use Case 4/Failure Use Case 2), not all the various POMs' Directory Specific SCM Numbers will be changed when code is checked-in.
6b) Derived Requirement: On account of 6 and 6a, a single POM needs to be able to generate multiple packaging types (perhaps via profiles).

Final Derived Requirement Chain 1

1d) Continuum should publish P's Directory Specific SCM Number (appending it to P's Checked-in Maven Version Number) when it does official builds.

5b) Consumer C should (be able to) poll the remote repository for newer versions of its dependencies and check-in an updated version of its POM immediately *before* C builds (in a *pre-build* step).

6b) A single POM needs to be able to generate multiple packaging types (perhaps via profiles).

Notice that these requirements depend on each other. There's little point in coding requirements 5b/6b without coding requirement 1d. Furthermore, it's actually *worse* to code requirement 1d without coding requirements 5b/6b... it would be better to leave off 1a entirely than to code it without requirements 5b/6b.

Furthermore, note that if requirement 3a (automatic updating) is coded as *part* of the build (against requirement 5b, which explicitly requires that updates happen *pre-build*), you'll actually violate reproducibility, again making the problem worse, not better.

Requirement Chain 2: Final Names and Windows Libraries

6) Assertion: Windows has no symlink feature and no automated LD-style re-linker.
  6a) Assertion: Therefore, Windows libraries (DLL files) must have the same name at build-time and at run-time; they will not work if they are renamed after they are built (unless they are re-renamed back to their build-time names).
7) Requirement: It should be possible to test a Windows Consumer library Consumer.dll with a newer version of the Producer library Producer.dll without rebuilding Consumer.dll.
  7a) Assertion: Under the current Maven naming standard, Windows libraries (like all other files) include version numbers in their names.
  7b) Derived Assertion: On account of 6a and 7a, under the current Maven naming standard, requirement 7 is not fulfilled.
  7c) Derived Requirement: On account of 7, Windows libraries should not include version numbers in their names, either at build-time or at run-time.
8) Assertion: In Maven, you can use the /project/build/finalname tag in pom.xml to make files use a specified name at build-time. If you use <finalname>, the file will be renamed to include the version number in the default repository layout.
  8a) Derived Assertion: On account of 8 and 6a, Windows libraries will not work at run-time if users copy them directly from the Maven default repository layout without renaming them.
9) Assertion: As it currently stands, Maven plug-ins that use Windows libraries need to rename them back to their "finalname" before using them. Each Maven plug-in that can touch a Windows library must be coded to handle this special case.
10) Requirement: If a file has been built using Maven, it should be possible to tell what version it is without access to any other file.
  10a) Derived requirement: On account of 10, if possible, Maven should add version numbers to built file names.
  10b) Assertion: Windows libraries and Java JAR files can contain metadata containing version information. JARs contain metadata in a meta-inf/manifest.mf file zipped into the JAR; Windows libraries embed VersionInfo metadata that you can see by right-clicking on the file and viewing the file properties.
  10c) Derived requirement: On account of 10, if possible, Maven should burn version numbers into files that it builds, in Windows file meta data or in Java "manifest.mf" files, or by some other appropriate standard mechanism.
  10d) Derived requirement: On account of 10, if a file's name does not include its version number, its version number should *at least* be somehow burned into the file itself.
11) Assertion: End-users use the <finalname> tag to indicate that it is important that the file have a certain name at runtime; they assert that this requirement is more important than the requirement 10a.
  11a) On account of 6a and 11, Windows libraries should have their "finalname" set to their build-time name.
12) Requirement: It should be easy to consume files built with Maven just by copying them out of the remote repository, even if the consumers aren't using Maven.

12a) Derived Requirement: Therefore, files built with Maven should not spend *any* time under a name that makes them unusable.
12b) Derived Requirement: On account of 11 and 12, files built with Maven that have a <finalname> tag should always have that name, even in the default repository layout, because this is more important than requirement 10a.
12c) Derived Requirement: On account of 6a, 11a and 12, Windows libraries built with Maven should always have the same name they had at build time, even in the default repository layout, because this is more important than requirement 10a.
13) Requirement: Maven plug-ins that handle files with special naming requirements must always use those files under a name that works
13a) Derived Requirement: Therefore, the Maven Core should always provide working filenames to plugins.
13b) Derived Requirement: On account of 11 and 13a, when Maven uses files that have the <finalname> tag, the Maven Core should provide files to plug-ins under the specified finalname.
13c) Derived Requirement: On account of 6a, 11a and 13a, when Maven uses Windows libraries, the Maven Core should provide those library files to plugins under the same name under which they were built.

Final Derived Requirement Chain 2

7c) Windows libraries should not include version numbers in their names, either at build-time or at run-time.

11a) Windows libraries should have their "finalname" set to their build-time name.

12b) Files built with Maven that have a <finalname> tag should always have that name, even in the default repository layout.
12c) Windows libraries built with Maven should always have the same name they had at build time, even in the default repository layout.

13b) When Maven uses files that have the <finalname> tag, the Maven Core should provide files to plug-ins under the specified finalname.
13c) When Maven uses Windows libraries, the Maven Core should provide those library files to plugins under the same name under which they were built.

Clarifications

The problems I associate with Windows Libraries apply both to native win32 Windows libraries, as well as Windows COM libraries, as well as Microsoft .NET assemblies. None of these files can be renamed after build time and be expected to work. (Note that all of these files, each of which are very different types of files, have the .DLL extension.)

I am aware that our proposed changes involve a change to the Maven Core. I believe that this change is for the better and that everyone would benefit from it.

I believe that there may be some temptation to avoid changing the Maven Core and push the work of auto-re-renaming finally-named files onto the plug-ins. I believe that it would be a mistake to force the plug-ins to deal with this, because it inappropriately distributes code that should be written once in the Maven Core.

I also suspect there must be some temptation to ignore the point about <finalname>, and just add this special case code to Windows library plug-ins as a special case. But Windows development is not and cannot be a "special case". Most developers are Windows developers. **Windows libraries must stand as first-class citizens in the Maven universe**.
It's especially important that the Maven repository not just be a "jar repository," in which you could also (sort-of) put Windows DLLs. It shouldn't even be a library repository. It should be an artifact repository, which includes shell scripts, settings files, installers, and the whole menagerie of stuff that everyone needs to use their software.

Finally, note that the problem of including version numbers in library names is *bad* now, but it would become *much worse* if Requirement Chain 1 were satisfied without satisfying the Requirements in Requirement Chain 2, because it would require changing linking metadata much more frequently.

Thanks!