Ruminations of a Mavenite
Architecture, Design and Implementation

Bob McWhirter
bob@eng.werken.com
The Werken Company
http://www.werken.com/

February 13, 2003

Abstract

Maven, a project management tool created by Jason van Zyl in early 2002 has seen rapid evolution with an increase in both features and inadequacies. I have worked with Maven for only a few weeks, but is obvious that in order to survive and thrive, certain changes must occur. Weather these changes are evolutionary or revolutionary in terms of the current Maven code-base is still to be decided. Herein, I address various good and bad points that I have observed while using Maven on several projects.

1 Origins, History & Future

1.1 Motivation

Jason van Zyl, attempting to get a hold on the recently decoupled Turbine components, decided to start the Maven project. Turbine had initially been a mostly monolithic project, but over the course of time various components were shed from the core project into other projects such as *jakarta-commons*. As such, to develop on Turbine, a developer must have all dependencies up-to-date and managing these dependent jars was a hassle. Maven is Jason's attempt to provide a consistent mechanism for building multi-project code-bases. Maven was initially announced on February 25 on the *turbine-dev* mailing list.

1.2 The Early Years

The initial development was performed by Jason van Zyl, while documentation was produced by Pete Kazmier. The current incarnation of Maven is basically an add-on to widely-used *jakarta-ant* Java build-system.

Fairly quickly, Maven developed a good-sized base of developers, documenters, testers and users. People immediately saw the beauty behind maven, of isolating logic into an abstract project descriptor instead of directly coding actions within *ant*. Maven grew quickly and now supports many J2EE components, documentation-generator of several types, and advanced inter-project dependency management.

The maven developer community is active on the *#maven* channel of Jon Scott Steven's IRC server *irc.whichever.com*.

1.3 The Future

The general consensus is that Maven could some day be an *ant*-killer, obviating the need for ant. While it may be politically incorrect within the jakarta community, many of the maven developers believe there are fundamental flaws with the design an implementation of *ant*, and hold little hope for improvement in *ant-2*.

2 Features and Capabilities

2.1 Network-Enabled

2.1.1 Shared Dependency Repositories

Maven makes extensive use of the network when managing inter-project dependencies. This is considered one of the most important features of Maven. When building a project, and dependent libraries may be pulled from a common network repository and placed into a local cache. Maven itself manages the classpath issues in order to allow building of the target project. No longer must a user explicitly download dependencies and manually install them into the correct locations for building, testing and deploying.

Maven currently resolves all project dependencies through a local cache which may be populated from multiple other local or remote repositories.

The one main issue with regards to the remote repositories is administration. Currently, the remote repository is simply an HTTP accessible directory. For example, the main maven repository is locate at *http://jakarta.apache.org/turbine/jars/*. Some sort of repository-management scheme needs to be implemented. Currently, the owner of the directory on the server is responsible for maintaining the repository using normal file-system commands.

I propose that a repository management tool be created that will either accept built jars from the responsible projects or provides some mechanism for the repository itself to build releases of jars. In the common case, a project could notify the repository, possibly through email or an HTTP request, that a new version of a project is available. The repository is then responsible for syncing out the correct version of the code from a revision-control system and building an official build.

One issue with the method is that "official" builds will not actually come from the project itself, but rather from the repository itself.

If each project were allowed to submit versions of their deliverables, strict security must be in place. Potentially public-key signatures could be used while submitting deliverables in order to prevent spoofing and the possibility of introducing malicious code. This issue is extreme important with the central repository because any bogus code propagates quickly to many other projects and users. The downside of using signatures is that each project must establish a relationship with each repository in order to verify public key fingerprints, adding to administrative overhead.

As a complete departure, each project could possibly maintain their own repository, under their own control, and simply register the repository itself with a central repository-manager, instead of registering the actual deliverables.

2.2 Extensible Components

2.2.1 Action & Plug-In Extension

Maven, at the core, relies on *elemental actions.* Various elemental actions may be used in concert to produce an *extension*, which is implemented as a run-time plug-in. Elemental actions consist of things such as:

- Compilation of a source tree.
- Documentation generation of a source tree.
- Processing a set of templates and data.
- Compressing a file.

In the original version of Maven, based upon *ant*, these element actions were simply ant's own `Task` objects. Maven simply built the work already done by the ant team. An ant `Task` is simply a parameterizable process. The ant `<javac>` task, for example, takes `srcDir`, `destDir`, `classpath` and other flags and parameters.

Larger-grain *activities* aggregate several elemental actions and parameterize them with information from the *project descriptor.* A project to simply defines the location of the source tree along with a set of external dependencies, and Maven will determine the necessary invocation of `<javac>` to perform compilation correctly.

Maven's current reliance upon ant is proving to be one of its biggest faults. Ant's build-file language was never intended for the type of programming the Maven developers are using it for. In order to implement the initialization, callbacks and other features of Maven, many ant targets are required, drastically slowing down even a small incremental build. Additionally, the passing of information in various forms to various targets in various build-files is sometimes tenuous at best. Ant seems to basically be making the goals of Maven more difficult to attain than they should be. Several replacement options are described below.

2.2.2 Project-Defined Callbacks

While Maven attempts to always Do The Right Thing, there are times that a project will require the ability to augment and otherwise affect the normal execution of a Maven build. Currently, this is implemented using a callback mechanism. Before and after each Maven activity, the project is given an opportunity to fire a callback routine. This is currently also implemented as a project-define ant target. By implementing project-defined callbacks in the ant build-file language, Maven is accessible to the large body of ant-knowledgeable developers.

3 Architectural & Design Issues

3.1 Elemental actions: Ant, Jelly or Otherwise

Internally, Maven implements the *project descriptor* as a normal graph of JavaBeans known as the *Project Object Model (POM)*. In the current implementation, information from the POM is extracted and made available to the ant process in order to parameterize various elemental actions. Ant itself is currently the driver of Maven. A user must explicitly invoke maven capabilities from his own ant build-file. It is this aspect of Maven being secondary to ant, instead of being in control the creates hardships for the Maven developers.

We propose to, at the bare minimum, invert this relationship so that ant is a slave to Maven. Maven would become a full application, instead of simply a collection of ant tasks. This would allow each project to actually exist without a build.xml file, unless required for implementation of callbacks.

Developers are finding it less-than-friendly to perform the complex logic necessary for maven in ant's XML language. Many feel that XML is not an appropriate scripting language for these tasks. Several participants (including the author) feel that for elemental tasks, a pure JavaBeans interface may be appropriate. While ant has walked partially down their path by implementing the `Task` interface, this interface is not generic and is dependent upon the rest of the ant components. A developer may not simply use the `Javac` task independently of the rest of ant.

I therefore propose that Maven uses independent elemental action JavaBeans that do not rely upon other Maven components, necessarily (Figure 1).

Notice that this class contains absolutely nothing from the Maven project itself. It is imminently re-usable in many places without maintaining complex relationships to other components. At run-time, Maven would simply use reflection to invoke the `execute()` method.

This would be the basic building-block for Maven.

A middle mediator layer would be used to extract the necessary information from the POM to parameterize the elemental actions.

The mediator layer may somehow be implemented using James Strachan's *Jelly* (Figure 2), or may simply be done using properties files and XPaths to walk the POM (Figure 3).

```
import java.io.File;

public class Javac {

    public Javac() {
        ....
    }

    public void setSourceDirectory(File sourceDirectory) {
        ....
    }

    public void setDestinationDirectory(File sourceDirectory) {
        ....
    }

    public void execute() throws Exception {
        ....
    }
}
```

Figure 1: Javac Elemental Action

```
<maven:javac>
    <sourceDirectory>${project.sourceDirectory}</sourceDirectory>
    <destinationDirectory>${project.destinationDirectory}</destinationDirectory>
    <classpath>${project.buildTimeDependencies}</classpath>
    ....
</maven:javac>
```

Figure 2: Jelly Mediator Example

```
sourceDirectory      = /project/sourceDirectory
destinationDirectory = /project/destinationDirectory
classpath            = /project/buildTimeDependencies
....
```

Figure 3: XPath Mediator Example

3.2 Composition and Inheritance

By implementing elemental actions simply as generic JavaBeans, all normal Java constructs for inheritance and composition are available. I personally suggest that Java is the method for constructing both elemental actions and the larger-grained activities. Ant's use of an interpreted XML scripting language is important because every developer on every project was required to write his own build-file. Since Maven makes this mostly unneeded, where only the Maven developers have to create actions, binding ourselves directly to Java does not seem bad. Through the use of aspect-oriented programming (AOP) or post-compilation byte-code processing (possibly as class resolution time), we can add dependency-checking advice before each invocation of an action's `execute()` method.

There *is* still a need for project-defined callbacks, and this is an area where I believe we need many options. Maintaining the familiar ant build-file for project-specific callbacks (a form of extension) is necessary. Though, I propose that the callback mechanism include an abstraction layer so that callbacks may be implemented as Java code, ant targets, jelly scripts, or others.

3.3 XML, JavaBeans and Testing

By implementing elemental actions are mere JavaBeans without dependency upon Maven components, unit-testing will be much easier. We currently have issues with attempting to test the ant-based Maven targets. By converting the targets to JavaBeans, this will be alleviated. If we bind ourselves to an XML scripting language or any sort as the only method for controlling the elemental activities, the testing process becomes much more difficult and cross-cutting.

3.4 Dependencies

Dependencies between projects is accounted for through the repository mechanism described above. Dependencies between elemental actions and larger-grained activities must also be accounted for. Each activity may also include version specifiers. It may be possible to simply utilize the normal inter-project dependency repository for the Maven-extension dependencies also. This could be accomplished by tagging each project dependency as a build-time or run-time dependency, possibly. More is discussed below in sections 4.1 and 4.2.

4 Implementation Issues

4.1 Dependencies

While the notion of dependencies seems simple enough, there are many complexities involved, several of which have been directly experienced by the jakarta-ant project team.

4.1.1 Maven Dependencies

Maven itself, as a project, has dependencies. When building Maven, it is no different than building any other project. The complexity comes during the execution of Maven, and maintain the correct classpaths. The classpath issue is discussed below in section 4.2.

4.1.2 Extension Dependencies

Since Maven can load, at run-time, various extensions to provide more capabilities, the dependencies (and possibly conflicts) between these extensions must also be managed at run-time. Luckily, this is simply considered a subset of the afore-mentioned maven dependencies when considering classpaths. Unlike the run-time dependency mechanism, below, this requires a run-time loading of dependent jars into Maven's own process-space.

4.1.3 Build-Time Dependencies

Build-time dependencies can be considered a sub-set of the extension dependencies above. There may exist an extension that provides an interface to *javacc* or *antlr* through Maven. In addition to the dependency on those elemental activities, those activities themselves required the javacc or antlr tools to be available within Maven's process-space.

4.1.4 Run-Time/Deploy-Time Dependencies

Run-time/deploy-time dependencies are the simplest dependencies to manage. These are *not* required to be present within Maven's own process-space. Instead, the list of them must be provided to activities and elemental actions executed within Maven. For example, during compilation, Maven is not required to have the dependent libraries within its process space, but it must provide to the Java compiler a list of locations for the dependencies. The compiler uses this list internally.

4.2 The Many Classpaths

4.2.1 Maven's Classpath

Maven's own classpath is comprised of the following:

- *Core Maven Components.* This includes maven.jar and all other libraries that maven itself requires in order to operate.
- *Extension Components.* This includes extensions that bundled into jar files and deployed within Maven itself.
- *Build-time Components.* This includes libraries that the extensions themselves rely upon.

4.2.2 Run-Time Dependency Classpath

The run-time dependency classpath is not actually loaded within Maven's process-space, but is simply a list of dependencies on the local drive that is made available to activities and actions.

4.2.3 Classpath interactions

By far, the most interesting classpath interaction involves *junit*. A project may have a source-tree of unit tests written against junit 3.6. Therefore, junit-3.6.jar is described as a run-time dependency, as it is required in order to actually execute the unit-tests. Maven's own `junit` action, though, may be compiled against junit 3.7. Here we have a definite version mismatch. Somehow, we must find a method for aligning run-time dependencies with extension and build-time dependencies. Otherwise, users of Maven are inherently forced to use the version of junit (or other libraries that exhibit similar usage idioms) that Maven itself depends upon. This would be sub-optimal.

4.2.4 Classloaders

We should achieve complete isolation, where possible, between Maven's own classpath and the classpath provided to elemental action. For example, while maven uses log4j 1.2.3 internally, it should not ever be presented to an elemental action in the dependency classpath. There also exists the possibility of version clashes between Maven extensions, and if possible these should be handled gracefully. A classloader sandboxing mechanism may be necessarily even with Maven itself, to possibly isolate each extension component's classpath from another. Management of the classloader hierarchy will be an important, if not exciting, task.

5 Bottom Line

To sum up, I feel that the current incarnation of Maven is indeed a wonderful prototype of a beautiful concept. I feel that we should take a hard look at massively reworking Maven to support further future development. If we do not, I fear that Maven will be lost in a quagmire of hacks.

The following things should be well-design and cleanly implemented, the next time around:

- *Maven is an Applications.* Maven should be a full-fledged application, and not simply a collection of ant tasks. Maven should not be subservient.
- *Repository Management.* Consider either a server component for the repository to allow each project to update its own libraries, or a distributed model, like DNS, where registration of a repository allows delegation of management responsibilities.

- *Full-Fledge Language.* Instead of working primarily in an interpreted XML-based language such as ant build-files or Jelly, the primary language for implementing Maven actions and activities should be normal Java. Avoid anything Maven-specific, and allow the Maven mediator layer parameterize the JavaBean actions.

- *Robust Dependency Management.* Being able to manage the many times of dependencies between Maven, extensions, build-time and run-time is the utmost importance.

- *Correct ClassLoader Sand-boxing.* Hand-in-hand with robust dependency management is correct handling of classloaders within Maven, to avoid collisions or incorrect behaviour.

I heartily applaud all of the developers who have spent considerable time bringing Maven to where it is today. I would like to see the same amount of progress continue into the future and suggest a framework that will hopefully make this realizable. If a re-engineering of Maven does not occur, I feel that in the near future, we will hit a wall where every added feature creates more problems than it solves.