Software Development and Management

Daniel Drake

July 13, 2020

1 Agile

The values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The Manifesto for Agile Software Development is based on twelve principles:

- Customer satisfaction by early and continuous delivery of valuable software.
- Welcome changing requirements, even in late development.
- Deliver working software frequently (weeks rather than months)
- Close, daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted
- Face-to-face conversation is the best form of communication (co-location)
- Working software is the primary measure of progress
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicity—the art of maximizing the amount of work not done—is essential
- Best architectures, requirements, and designs emerge from self-organizing teams
- Regularly, the team reflects on how to become more effective, and adjusts accordingly

2 Java and Maven

2.1 POM.XML and File Organization

The purpose of the pom.xml file is: To describe the project, configure plugins, and declare dependencies.
When we create a project, our sources go under the **src/main** directory.
As an example, we might put our java classes in: **src/main/java** and our **classpath resources** are placed in **src/main/resources**

The files under the `src/main/java` directories typically follow this pattern:

```
src/main/java/org/url/<groupId>/<artifactId>/javaFilesGoHere
```

This makes the package `org.url.groupId.artifactId` which is referenced in the javaFilesGoHere file.

2.2 Maven coordinates:

- The <ArtifactId> is the project's base directory and the name of the project.

  A unique identifier under groupId that represents a single project.

- The <groupId> tells you which company or group made the project.

  The group, company, team, organization, project, or other group. The convention for group identifiers is that they begin with the reverse domain name of the organization that creates the project. Projects from Sonatype would have a groupId that begins with com.sonatype, and projects in the Apache Software Foundation would have a groupId that starts with org.apache.

- The <packaging> indicates what the project will build.

  The type of project, defaulting to jar, describing the packaged output produced by a project. A project with packaging jar produces a JAR archive; a project with packaging war produces a web application.

- The <version> is literally just the version of the code.

  A specific release of a project. Projects that have been released have a fixed version identifier that refers to a specific version of the project. Projects undergoing active development can use a special identifier that marks a version as a SNAPSHOT.

- The <classifier>

The Maven coordinates uniquely identify a project and are referenced:

```
groupId:artifactId:packaging:version
```
Example:
```
mavenbook:my-app:jar:1.0-SNAPSHOT
```

Maven always executes against an effective POM, a combination of settings from this project's pom.xml, all parent POMs, a super-POM defined within Maven, user-defined settings, and active profiles. All projects ultimately extend the super-POM, which defines a set of sensible default configuration settings. While your project might have a relatively minimal pom.xml, the contents of your project's POM are interpolated with the contents of all parent POMs, user settings, and any active profiles. To see this "effective" POM, run the following command in the simple project's base directory.

```
mvn help:effective-pom
```

When you run this, you should see a much larger POM which exposes the default settings of Maven. This goal can come in handy if you are trying to debug a build and want to see how all of the current project's ancestor POMs are contributing to the effective POM.

When Maven executes a goal, each goal has access to the information defined in a project's POM. When the jar:jar goal needs to create a JAR file, it looks to the POM to find out what the JAR file's name is. When the compiler:compile goal compiles Java source code into bytecode, it looks to the POM to see if there are any parameters for the compile goal. Goals execute in the context of a POM. Goals are actions we wish to take upon a project, and a project is defined by a POM. The POM names the project, provides a set of unique identifiers (coordinates) for a project, and defines the relationships between this project and others through dependencies, parents, and prerequisites. A POM can also customize plugin behavior and supply information about the community and developers involved in a project.

2.3 More Tags and dependencies

The: `<configuration>` tag allows you to set things like: `<source>` and `<target>` which set A and B respectively.
The: `<name>` tag allows you to:
The: `<url>` tag allows you to:
To add dependencies we need to know the `<groupID>`, the `<artifactID>`, and the `<version>`. In addition, you may want to include a `<scope>` tag which does:
We can find the available groupId and artifactId values here, the sonatype repository. With this information we can then add a dependency as follows:

```
<dependencies>
     <dependency>
          <groupId>superSpecialDevGroup</groupId>
          <artifactId>dep_0</artifactId>
          <version>69.420.69</version>
     <\dependency>
     <dependency>
          <groupId>boringNormalDevGroup</groupId>
          <artifactId>dep_1</artifactId>
          <version>1.0.0</version>
          <scope>test</scope>
     <\dependency>
<\dependencies>
```

2.4 Plugin's and Goals

In general, Maven plugin is a collection of goals.
An example of a goal would be creating a jar file, compiling some code, doing unit tests, or generating reports.

When Maven executes a plugin goal, it prints out the plugin identifier and goal identifier to standard output:

```
[INFO] [archetype:generate]
```

In this example, maven is attempting to execute the archetype plugin's generate goal.
A custom plugin can be written in Java, Ant, Groovy, beanshell, Ruby or others.
If a plugin isn't already defined and ready to go for Mavin, it will download them from the central Maven repository.
When the programmer calls: `mvn install`
This is equivalent to calling:

```
mvn resources:resources \
     compiler:compile \
     resources:testResources \
     compiler:testCompile \
     surefire:test \
     jar:jar \
     install:install
```

2.5 Repository

A repository is a collection of project artifacts stored in a directory structure that closely matches a project's Maven coordinates.

Example

The standard for a Maven repository is to store an artifact in the following directory relative to the root of the repository:

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

> Maven downloads artifacts and plugins from a remote repository to your local machine and stores these artifacts in your local Maven repository. Once Maven has downloaded an artifact from the remote Maven repository it never needs to download that artifact again as Maven will always look for the artifact in the local repository before looking elsewhere. On Unix systems, your local Maven repository is available in /.m2/repository.