Introducing Maven

This chapter covers:

- An overview of Maven
- Maven's objectives
- Maven's principles:
  - Convention over configuration
  - Reuse of build logic
  - Declarative execution
  - Coherent organization of dependencies
  - Maven's benefits

Things should be made as simple as possible, but not any simpler.

- Albert Einstein

1.1. Maven Overview

Maven provides a comprehensive approach to managing software projects. From compilation, to distribution, to documentation, to team collaboration, Maven provides the necessary abstractions that encourage reuse and take much of the work out of project builds.

1.1.1. What is Maven?

Maven is a *project management framework*, but this doesn't tell you much about Maven. It's the most obvious three-word definition of Maven the authors could come up with, but the term *project management framework* is a meaningless abstraction that doesn't do justice to the richness and complexity of Maven. Too often technologists rely on abstract phrases to capture complex topics in three or four words, and with repetition phrases such as *project management* and *enterprise software* start to lose concrete meaning.

When someone wants to know what Maven is, they will usually ask “What exactly is Maven?”, and they expect a short, sound-bite answer. “Well it is a build tool or a scripting framework” Maven is more than three boring, uninspiring words. It is a combination of ideas, standards, and software, and it is impossible to distill the definition of Maven to simply digested sound-bites. Revolutionary ideas are often difficult to convey with words. If you are interested in a fuller, richer definition of Maven read this introduction; it will prime you for the concepts that are to follow. If you are reading this introduction just to find something to tell your manager[1], you can stop reading now and skip to Chapter 2.

If Maven isn't a “project management framework". what is it? Here's one attempt at a description: Maven is a set of standards, a repository format, and a piece of software used to manage and describe projects. It defines a standard life cycle for building, testing, and deploying project artifacts. It provides a framework that enables easy reuse of common build logic for all projects following Maven's standards. The Maven project at the Apache Software Foundation is an open source community which produces software tools that understand a common declarative *Project Object Model* (POM). This book focuses on the core tool produced by the Maven project, Maven 2, a framework that greatly simplifies the process of managing a software project.

You may have been expecting a more straightforward answer. Perhaps you picked up this book because someone told you that Maven is a build tool. Don't worry, Maven can be the build tool you are looking for, and many developers who have approached Maven as another build tool have come away with a finely tuned build system. While you are free to use Maven as “just another build tool”, to view it in such limited terms is akin to saying that a web browser is nothing more than a tool that reads hypertext.

Maven and the technologies related to the Maven project are beginning to have a transformative effect on the Java community.

In addition to solving straightforward, first-order problems such as simplifying builds, documentation, distribution, and the deployment process, Maven also brings with it some compelling second-order benefits.

1 You can tell your manager: “Maven is a declarative project management tool that decreases your overall time to market by effectively leveraging your synergies. It simultaneously reduces your headcount and leads to remarkable operational efficiencies.”

As more and more projects and products adopt Maven as a foundation for project management, it becomes easier to build relationships between projects and to build systems that navigate and report on these relationships. Maven's standard formats enable a sort of "Semantic Web" for programming projects. Maven's standards and central repository have defined a new naming system for projects. Using Maven has made it easier to add external dependencies and publish your own components.

So to answer the original question: Maven is many things to many people. It is a set of standards and an approach as much as it is a piece of software. It is a way of approaching a set of software as a collection of interdependent components which can be described in a common format. It is the next step in the evolution of how individuals and organizations collaborate to create software systems. Once you get up to speed on the fundamentals of Maven, you will wonder how you ever developed without it.

1.1.2. Maven's Origins

Maven was borne of the practical desire to make several projects at the Apache Software Foundation (ASF) work in the same, predictable way. Prior to Maven, every project at the ASF had a different approach to compilation, distribution, and Web site generation. The ASF was effectively a series of isolated islands of innovation. There were some common themes to each separate build, but different communities were creating different build systems of varied complexity and there was no reuse of build logic between projects. The build process for Tomcat was different from the build process used by Struts, the Turbine developers used a different approach to site generation than the developers of Jakarta Commons. Etc.

This lack of a common approach to building software meant that every new project tended to copy and paste another project's build system. Ultimately, this copy and paste approach to build reuse reached a critical tipping point at which the amount of work required to maintain this collection of build systems was distracting from the central task of developing high-quality software. In addition, the barrier to entry for a project with a difficult build system was extremely high; projects such as Jakarta Taglibs had (and continue to have) a tough time attracting developer interest because it could take an hour to configure everything in just the right way. Instead of focusing on creating good component libraries or MVC frameworks, developers were building yet another build system.

Maven entered the scene by way of the Turbine project, and it immediately sparked interest as a sort of Rosetta Stone for software project management. Developers within the Turbine project could freely move between subcomponents, knowing clearly how they all worked just by understanding how one of them worked. If developers spent time understanding how one project was built, they would not have to go through this process again when they moved on to the next project. People stopped figuring out creative ways to compile, test, and package, and started focusing on component development.

The same effect extended to testing, generating documentation, generating metrics and reports, and deploying. If you followed the Maven Way, your project gained a build by default. Soon after the creation of Maven other projects, such as Jakarta Commons, the Codehaus community started to adopt Maven 1 as a foundation for project management.

Many people come to Maven familiar with Ant so it's a natural association, but Maven is an entirely different creature from Ant. Maven is not just a build tool, and not just a replacement for Ant.

Whereas Ant provides a toolbox for scripting builds, Maven is about the application of patterns in order to facilitate project management through the reuse of common build strategies. Make files and Ant are alternatives to Maven.

However, if your project currently relies on an existing Ant build script that must be maintained, existing Ant scripts and Make files can be complementary to Maven and used through Maven's plugin architecture. Plugins allow developers to call existing Ant scripts and Make files and incorporate those existing functions into the Maven build life cycle.

1.1.3. What Does Maven Provide?

It provides a comprehensive model that can be applied to all software projects. Maven provides a common project “language”, and the software tool named Maven is just a secondary side-effect of this model. Projects and systems that use this standard declarative approach tend to be more transparent, more reusable, more maintainable, and easier to comprehend.

Maven provides a useful abstraction in the same way an automobile provides an abstraction most of us are familiar with. When you purchase a new car, the car provides a known interface; if you've learned how to drive a Jeep, you can easily drive a Camry. Maven takes a similar approach to software projects: if you can build one Maven project you can build them all, and if you can apply a testing plugin to one project, you can apply it to all projects. You describe your project using Maven's model, and you gain access to expertise and best-practices of an entire industry.

An individual Maven project's structure and contents are declared in a Project Object Model (POM), which forms the basis of the entire Maven system. The key value to developers from Maven is that it takes a declarative approach rather than developers creating the build process themselves, referred to as "building the build". Maven allows developers to declare goals and dependencies and rely on Maven’s default structures and plugin capabilities to build the project. Much of the project management and build orchestration (compile, test, assemble, install) is effectively delegated to the POM and the appropriate plugins. Developers can build any given project without having to understand how the individual plugins work (scripts in the Ant world).

Given the highly inter-dependent nature of projects in open source, Maven’s ability to standardize locations for source files, documentation, and output, to provide a common layout for project documentation, and to retrieve project dependencies from a shared storage area makes the building process much less time consuming, and much more transparent.

Maven provides you with:

- A comprehensive model for software projects
- Tools that interact with this Declarative Model

Organizations and projects that adopt Maven benefit from:

- **Coherence** - Maven allows organizations to standardize on a set of best practices. Because Maven projects adhere to a standard model they are less opaque. The definition of this term from the American Heritage dictionary captures the meaning perfectly: “Marked by an orderly, logical, and aesthetically consistent relation of parts.“
- **Reusability** - Maven is built upon a foundation of reuse. When you adopt Maven you are effectively reusing the best practices of an entire industry.
- **Agility** - Maven lowers the barrier to reuse not only of build logic but of components. When using Maven it is easier to create a component and integrate it into a multi-project build. It is easier for developers to jump between different projects without a steep learning curve that accompanies a custom, home-grown build system.
- **Maintainability** - Organizations that adopt Maven can stop building the build, and start focusing on the application. Maven projects are more maintainable because they have fewer surprises and follow a common model.

Without these characteristics, it is improbable that multiple individuals will work productively together on a project. Without visibility it is unlikely one individual will know what another has accomplished and it is likely that useful code will not be reused. When code is not reused it is very hard to create a maintainable system.

When everyone is constantly searching to find all the different bits and pieces that make up a project, there is little chance anyone is going to comprehend the project as a whole. As a result you end up with a lack of shared knowledge along with the commensurate degree of frustration among team members. This is a natural effect when processes don't work the same way for everyone.

1.2. Maven's Principles

According to Christopher Alexander "*patterns help create a shared language for communicating insight and experience about problems and their solutions*". The following principles were inspired by Christopher Alexander's idea of creating a shared language:

- Convention over configuration
- Declarative execution
- Reuse of build logic
- Coherent organization of dependencies

Maven provides a shared language for software development projects. As mentioned earlier Maven creates a sense of structure so that problems can be approached in terms of this structure. Each of these principles allows developers to talk about their projects at a higher level of abstraction allowing more effective communication and allowing team members to get on with the important work of creating value at the application level. This chapter will examine each of these principles in detail and these principles will be demonstrated in action in the following chapter when you create your first Maven project.

1.2.1. Convention Over Configuration

One of the central tenets of Maven is to provide sensible default strategies for the most common tasks so that you don't have to think about these mundane details. This is not to say that you can't override them, but the use of these sensible default strategies is encouraged and should be strayed from only when absolutely necessary.

This notion is known as "convention over configuration" and has been popularized by the Ruby on Rails (ROR) community and specifically encouraged by ROR's creator David Heinemeier Hansson who summarizes the notion as follows:

> "Rails is opinionated software. It eschews placing the old ideals of software in a primary position. One of those ideals is flexibility, the notion that we should try to accommodate as many approaches as possible, that we shouldn't pass judgment on one form of development over another. Well, Rails does, and I believe that's why it works.
>
> With Rails, you trade flexibility at the infrastructure level to gain flexibility at the application level. If you are happy to work along the golden path that I've embedded in Rails, you gain an immense reward in terms of productivity that allows you to do more, sooner, and better at the application level.
>
> One characteristic of opinionated software is the notion of "convention over configuration". If you follow basic conventions, such as classes are singular and tables are plural (a person class relates to a people table), you're rewarded by not having to configure that link. The class automatically knows which table to use for persistence. We have a ton of examples like that, which all add up to make a huge difference in daily use."[2]

2 O'Reilly interview with DHH

David Heinemeier Hansson articulates very well what Maven has aimed to accomplish since its inception (note that David Heinemeier Hansson in no way endorses the use of Maven, he probably doesn't even know what Maven is and wouldn't like it if he did because it's not written in Ruby yet!). Using standard conventions saves time, makes it easier to communicate to others, and allows you to create value in your applications faster with less effort. You shouldn't need to spend a lot of time getting your development infrastructure functioning.

With Maven you slot the various pieces in where it asks and Maven will take care of almost all of the mundane aspects for you. You don't want to spend time fiddling with building, generating documentation, or deploying. All of these things should simply work, and this is what Maven provides for you.

There are three primary conventions that Maven employs to promote a familiar development environment:

- Standard directory layout for projects
- The concept of a single Maven project producing a single output
- Standard naming conventions

Let's elaborate on each of these points in the following sections.

Standard directory layout for projects

The first convention used by Maven is a standard directory layout for project sources, project resources, configuration files, generated output, and documentation. These components are generally referred to as project content.

Maven encourages a common arrangement of project content so that once you are familiar with the standard locations, you will be able to navigate within any Maven project you encounter in the future.

It is a very simple idea but it can save you a lot of time. If this saves you 30 minutes for each new project you look at, even if you only look at a few new projects a year that's time better spent on your application. First time users often complain about Maven forcing you to do things a certain way and the formalization of the directory structure is the source of most of the complaints.

You can override any of Maven's defaults to create a directory layout of your choosing, but, when you do this, you need to ask yourself if the extra configuration that comes with customization is really worth it. If you have no choice in the matter due to organizational policy or integration issues with existing systems you might be forced to use a directory structure that diverges from Maven's defaults.

In this case, you will be able to adapt your project to your customized layout at a cost, increased complexity of your project's POM. If you do have a choice then why not harness the collective knowledge that has built up as a result of using this convention? You will be able to look at other projects and immediately understand the project layout. Follow the standard directory layout, and you will make it easier to communicate about your project. You will see clear examples of the standard directory structure in the next chapter but you can also take a look in Appendix B for a full listing of the standard conventions.

One primary output per project

The second convention used by Maven is the concept of a Maven project producing one primary output. To illustrate this point consider a set of sources for a client/server-based application that contained the client code, the server code, and some shared utility code.

You could produce a single JAR file which contained all the compiled classes, but Maven would encourage you to have three separate projects: a project for the client portion of the application, a project for the server portion of the application, and a project for the shared utility code. In this scenario the code contained in each separate project has a different role to play, or concern, and they should be separated.

The *separation of concerns* (SoC) principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability.

If you have placed all the sources together in a single project. the boundaries between our three separate concerns can easily become blurred and a simple desire to reuse the utility code could prove to be difficult. Having the utility code in a separate project, in a separate JAR file, is much easier to reuse. Maven pushes you to think clearly about the separation of concerns when setting up your projects because modularity leads to reuse.

Standard naming conventions

The third convention, a set of conventions really, is a standard naming convention for directories and a standard naming convention for the primary output of a project. The conventions provide clarity and immediate comprehension. This is important if there are multiple projects involved, because the naming convention keeps them separate in a logical, easy to see manner.

This is illustrated in the *Coherent Organization of Dependencies* section, later in this chapter.

A simple example of the naming convention might be `commons-logging-1.2.jar`. It is immediately obvious that this is version 1.2 of Commons Logging. If the JAR were named `commons-logging.jar` you would not really have any idea what version of Commons Logging you were dealing with and in a lot of cases not even the manifest in the JAR gives any indication.

The intent behind the naming conventions employed by Maven is that you can tell exactly what you are looking at by, well, looking at it. It doesn't make much sense to exclude pertinent information when you have it at hand to use.

Systems that cannot cope with information rich artifacts like `commons-logging-1.2.jar` are inherently flawed because there will come a time when something is misplaced and you'll track down a `ClassNotFound` exception which resulted from the wrong version of a JAR file being used. It's happened to all of us and it doesn't have to.

1.2.2. Reuse of Build Logic

As you have already learned, Maven encourages a separation of concerns because it promotes reuse. Maven puts this principle into practice by encapsulating build logic into coherent modules called *plugins*. Maven can be thought of as a framework which coordinates the execution of plugins in a well defined way.

In Maven there is a plugin for compiling source code, a plugin for running tests, a plugin for creating JARs, a plugin for creating Javadocs, and many others. Even from this short list of examples you can see that a plugin in Maven has a very specific role to play in the grand scheme of things. One important concept to keep in mind is that everything accomplished in Maven is the result of a plugin executing. Plugins are the key building blocks for everything in Maven.

The execution of Maven's plugins is coordinated by Maven's build life cycle in a declarative fashion with inputs from Maven's Project Object Model (POM), specifically from the plugin configurations contained in the POM.

1.2.3. Declarative Execution

Everything in Maven is driven in a declarative fashion using *Maven's Project Object Model* (POM) and specifically the plugin configurations contained in the POM.

Maven's project object model (POM)

Maven is project-centric by design, and the POM is Maven's description of a single project. Without POM, Maven is useless - the POM is Maven's currency. It is the POM that drives execution in Maven and this approach can be described as model driven execution.

The POM below is an example of what you could use to build and test a project. The POM is an XML document and looks like the following (very) simplified example:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This POM will allow you to compile, test, and generate basic documentation. You, being the observant reader, will ask how this is possible using a 15 line file. It's a very good and essential question in fact, and the answer lies in Maven's implicit use of its Super POM.

Maven's Super POM carries with it all the default conventions that Maven encourages, and is the analog of the Java language's `java.lang.Object` class.

In Java, all objects have the implicit parent of `java.lang.Object`. Likewise, in Maven all POMs have an implicit parent which is Maven's Super POM. The Super POM can be rather intimidating at first glance so if you wish to find out more about it you can refer to Appendix B. The key feature to remember is the Super POM contains important default information so you don't have to repeat this information in the POMs you create.

The POM contains every important piece of information about your project. The POM shown above is a very simple POM, but still displays the key elements every POM contains.

- `project` - This is the top-level element in all Maven `pom.xml` files.
- `modelVersion` - This required element indicates the version of the object model this POM is using. The version of the model itself changes very infrequently, but it is mandatory in order to ensure stability when Maven introduces new features or other model changes.
- `groupId` - This element indicates the unique identifier of the organization or group that created the project. The `groupId` is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example `org.apache.maven.plugins` is the designated `groupId` for all Maven plugins.
- `artifactId` - This element indicates the unique base name of the primary artifact being generated by this project. A typical artifact produced by Maven would have the form `<artifactId>-<version>.<extension>` (for example, `myapp-1.0.jar`). Additional artifacts such as source bundles also use the `artifactId` as part of their file name.
- `packaging` - This element indicates the package type to be used by this artifact (JAR, WAR, EAR, etc.). This not only means that the artifact produced is a JAR, WAR, or EAR, but also indicates a specific life cycle to use as part of the build process. The life cycle is a topic dealt with later in the chapter. For now, just keep in mind that the selected packaging of a project plays a part in customizing the build life cycle. The default value for the packaging element is jar so you do not have to specify this in most cases.
- `version` - This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development.
- `name` - This element indicates the display name used for the project. This is often used in Maven's generated documentation, and during the build process for your project, or other projects that use it as a dependency.
- `url` - This element indicates where the project's site can be found.
- `description` - This element provides a basic description of your project.

For a complete reference of the elements available for use in the POM please refer to the POM reference at http://maven.apache.org/maven-model/maven.html.

Maven's build life cycle

Software projects generally follow a similar, well-trodden path: preparation, compilation, testing, packaging, installation. The path that Maven moves along to accommodate an infinite variety of projects is called *the build life cycle*. In Maven, the build life cycle consists of a series of phases where each phase can perform one or more actions, or goals, related to that phase. For example, the compile phase invokes a certain set of goals to compile a set of classes.

In Maven you do day-to-day work by invoking particular phases in this standard build life cycle. For example, you tell Maven that you want to compile, or test, or package, or install. The actions you need to have performed are stated at a high level and Maven deals with the details behind the scenes. It is important to note that each phase in the life cycle will be executed up to and including the phase you specify. If you tell Maven to `compile`, the `validate, initialize, generate-sources, process-sources, generate-resources`, and `compile` phases will execute.

The standard build life cycle consists of many phases and these can be thought of as extension points. When you need to add some functionality to the build life cycle you do so with a plugin. Maven plugins provide reusable build logic that can be slotted into the standard build life cycle. Any time you need to customize the way your project builds you either employ the use of an existing plugin or create a custom plugin for the task at hand. See Chapter 2.7 Using Maven Plugins for an example of customizing the Maven build.

1.2.4. Coherent Organization of Dependencies

We are now going to delve into how Maven resolves dependencies and discuss the intimately connected concepts of dependencies, artifacts, and repositories. If you recall, our example POM has a single dependency listed, which is for `Junit`:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This POM states that your project has a dependency on JUnit, which is straightforward, but you may be asking yourself “Where does that dependency come from?” and “Where is the JAR?” The answers to those questions are not readily apparent without some explanation of how Maven dependencies, artifacts and repositories work. In “Maven-speak” an artifact is a specific piece of software.

In Java, the most common artifact is a JAR file, but a Java artifact could also be a WAR, SAR, or EAR file. A dependency is a reference to a specific artifact that resides in a repository. In order for Maven to attempt to satisfy a dependency, Maven needs to know what repository to look in as well as the dependency's coordinates. A dependency is uniquely identified by the following identifiers: `groupId`, `artifactId` and `version`.

At a basic level, we can describe the process of dependency management as Maven reaching out into the world, grabbing a dependency, and providing this dependency to your software project. There is more going on behind the scenes, but the key concept is that Maven dependencies are declarative.

In the POM you are not specifically telling Maven where the dependencies are physically, you are simply telling Maven what a specific project expects.

Maven takes the dependency coordinates you provide in a POM, and it supplies these coordinates to its own internal dependency mechanisms. With Maven, you stop focusing on a collection of JAR files; instead you deal with logical dependencies. Your project doesn't require `junit-3.8.1.jar`, it depends on version 3.8.1 of the `junit` artifact produced by the `junit` group. Dependency Management is one of the most easily understood and powerful features in Maven.

When a dependency is declared, Maven tries to satisfy that dependency by looking in all of the remote repositories that are available, within the context of your project, for artifacts that match the dependency request. If a matching artifact is located, Maven transports it from that remote repository to your local repository for general use.

Maven has two types of repositories: local and remote. Maven usually interacts with your local repository, but when a declared dependency is not present in your local repository Maven searches all the remote repositories it has access to in an attempt to find what's missing. Read the following sections for specific details about where Maven searches for these dependencies.

Local Maven repository

When you install and run Maven for the first time your local repository will be created and populated with artifacts as a result of dependency requests. By default, Maven creates your local repository in `~/.m2/repository`. You must have a local repository in order for Maven to work. The following folder structure shows the layout of a local Maven repository that has a few locally installed dependency artifacts such as `junit-3.8.1.jar`:

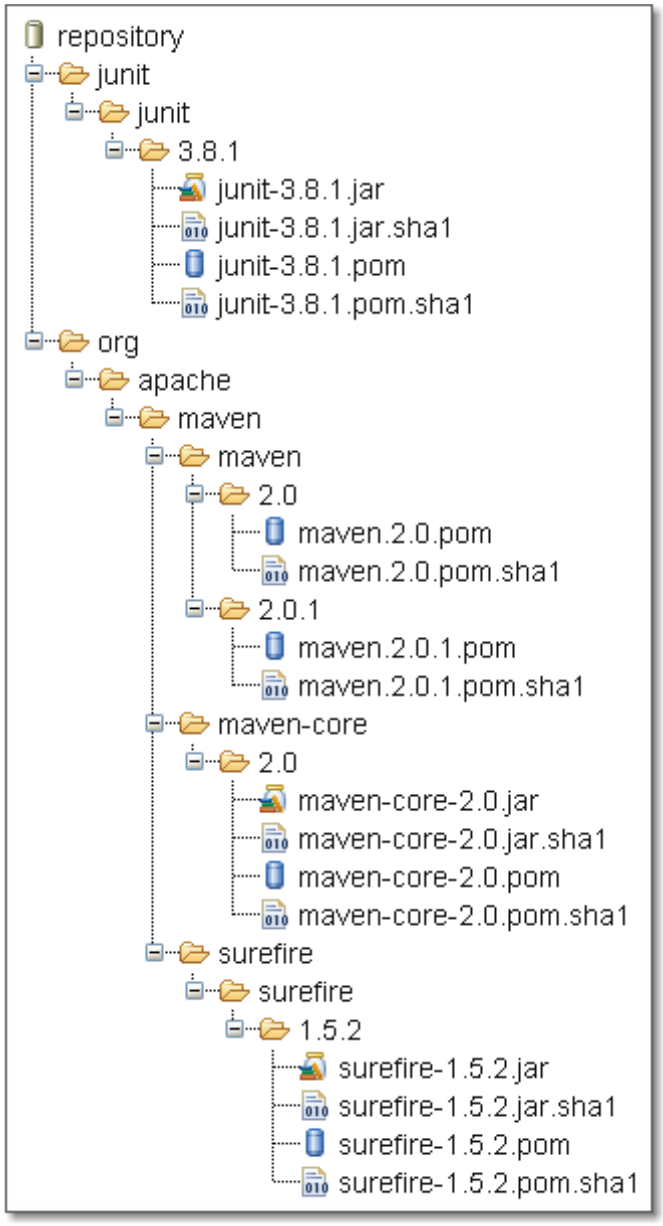

Figure 1-1: Artifact movement from remote to local repository

Take a closer look at one of the artifacts that appeared in your local repository, so you understand how the layout works. In theory a repository is just an abstract storage mechanism, but in practice the repository is a directory structure in your file system. We'll stick with our JUnit example and examine the `junit-3.8.1.jar` artifact that landed in your local repository.

Above you can see the directory structure that is created when the JUnit dependency is resolved. On the next page is the general pattern used to create the repository layout:

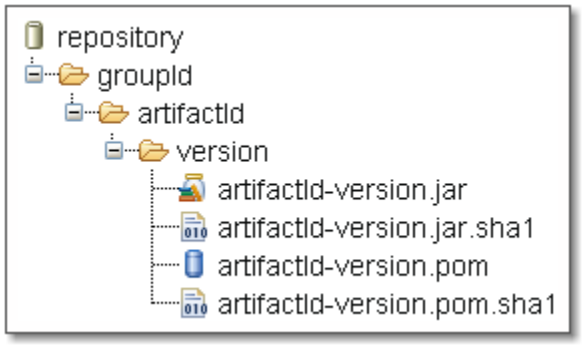

Figure 1-2: General pattern for the repository layout

If the `groupId` is a fully qualified domain name (something Maven encourages) such as `z.y.x` then you would end up with a directory structure like the following:

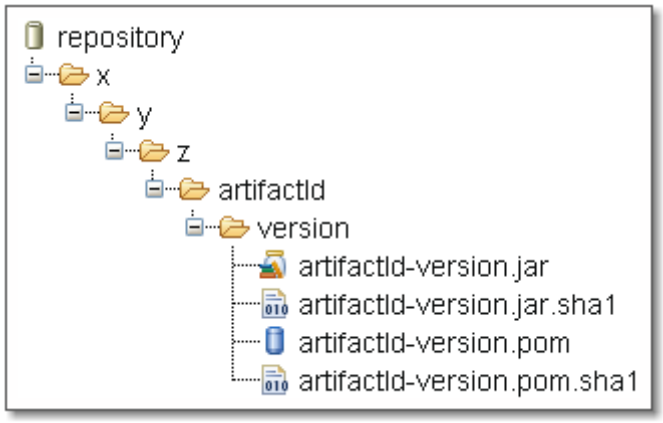

Figure 1-3: Sample directory structure

In the first directory listing you can see that Maven artifacts themselves are stored in a directory structure that corresponds to Maven's `groupId` of `org.apache.maven`.

Locating dependency artifacts

When satisfying dependencies, Maven attempts to locate a dependency's artifact using the following process. First, Maven will generate a path to the artifact in your local repository; for example, Maven will attempt to find the artifact with a `groupId` of "`junit`", `artifactId` of "`junit`", and a version of "`3.8.1`" in `~/.m2/repository/junit/junit/3.8.1/junit-3.8.1.jar`. If this file is not present, it will be fetched from a remote repository.

By default, Maven will attempt to fetch an artifact from the central Maven repository at http://www.ibiblio.org/maven2. If your project's POM contains more than one remote repository, Maven will attempt to download an artifact from each remote repository in the order defined in your POM. Once the dependency is satisfied, the artifact is downloaded and installed in your local repository.

From this point forward, every project with a POM that references the same dependency will use this single copy installed in your local repository. In other words, you don't store a copy of `junit-3.8.1.jar` for each project that needs it; all project referencing this dependency share a single copy of this JAR.

Your local repository is one-stop-shopping for all artifacts that you need regardless of how many projects you are working on. Before Maven, the common pattern in most project was to store JAR files in a project's subdirectory. If you were coding a web application, you would check in the 10-20 JAR files your project relies upon into a lib directory, and you would add these dependencies to your classpath.

While this approach works for a few projects, it doesn't scale easily to support an application with a great number of small components. With Maven, if your project has ten web applications which all depend on version 1.2.6 of the Spring Framework, there is no need to store the various spring JAR files in your project. Each project relies upon a specific artifact via the dependencies listed in a POM, and it is a trivial process to upgrade all of your ten web applications to Spring 2.0 by change your dependency declarations.

Instead of adding the Spring 2.0 JARs to every project, you simply changed some configuration. Storing artifacts in your SCM along with your project may seem appealing but it is incompatible with the concept of small, modular project arrangements. Dependencies are not your project's code, and they don't deserve to be versioned in an SCM. Declare your dependencies and let Maven take care of details like compilation and testing classpaths.

1.3. Maven's Benefits

A successful technology takes away a burden, rather than imposing one. You don't have to worry about whether or not it's going to work; you don't have to jump through hoops trying to get it to work; it should rarely, if ever, be a part of your thought process. Like the engine in your car or the processor in your laptop, a useful technology just works, in the background, shielding you from complexity and allowing you to focus on your specific task.

Maven provides such a technology for project management, and, in doing so, it simplifies the process of development. To summarize, Maven is a set of standards, Maven is a repository, Maven is a framework, and Maven is software. Maven is also a vibrant, active open-source community that produces software focused on project management. Using Maven is more than just downloading another JAR file and a set of scripts, it is the adoption of processes that allow you to take your software development to the next level.