# Migrating to Maven

This chapter explains how to migrate (convert) an existing build in Ant, to a build in Maven:

- Splitting existing sources and resources into modular Maven projects
- Taking advantage of Maven's inheritance and multi-project capabilities
- Compiling, testing and building jars with Maven, using both Java 1.4 and Java 5
- Using Ant tasks from within Maven
- Using Maven with your current directory structure

> This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.
>
> *- Morpheus. The Matrix*

# 8.1. Introduction

The purpose of this chapter is to show a migration path from an existing build in Ant to Maven.

The Maven migration example is based on the Spring Framework build, which uses an Ant script. This example will take you through the step-by-step process of migrating Spring to a modularized, component-based, Maven build.

You will learn how to start building with Maven, while still running your existing, Ant-based build system. This will allow you to evaluate Maven's technology, while enabling you to continue with your required work.

You will learn how to use an existing directory structure (though you will not be following the standard, recommended Maven directory structure), how to split your sources into modules or components, how to run Ant tasks from within Maven, and among other things, you will be introduced to the concept of dependencies.

## 8.1.1. Introducing the Spring Framework

The Spring Framework is one of today's most popular Java frameworks. For the purpose of this example, we will focus only on building version 2.0-m1 of Spring, which is the latest version at the time of writing.

The Spring release is composed of several modules, listed in build order:

- spring-core
- spring-beans
- spring-aop
- spring-context
- spring-dao
- spring-jdbc
- spring-support
- spring-web
- spring-webmvc
- spring-remoting
- spring-portlet
- spring-jdo
- spring-hibernate2
- spring-hibernate3
- spring-toplink
- spring-ojb
- spring-mock
- spring-aspects

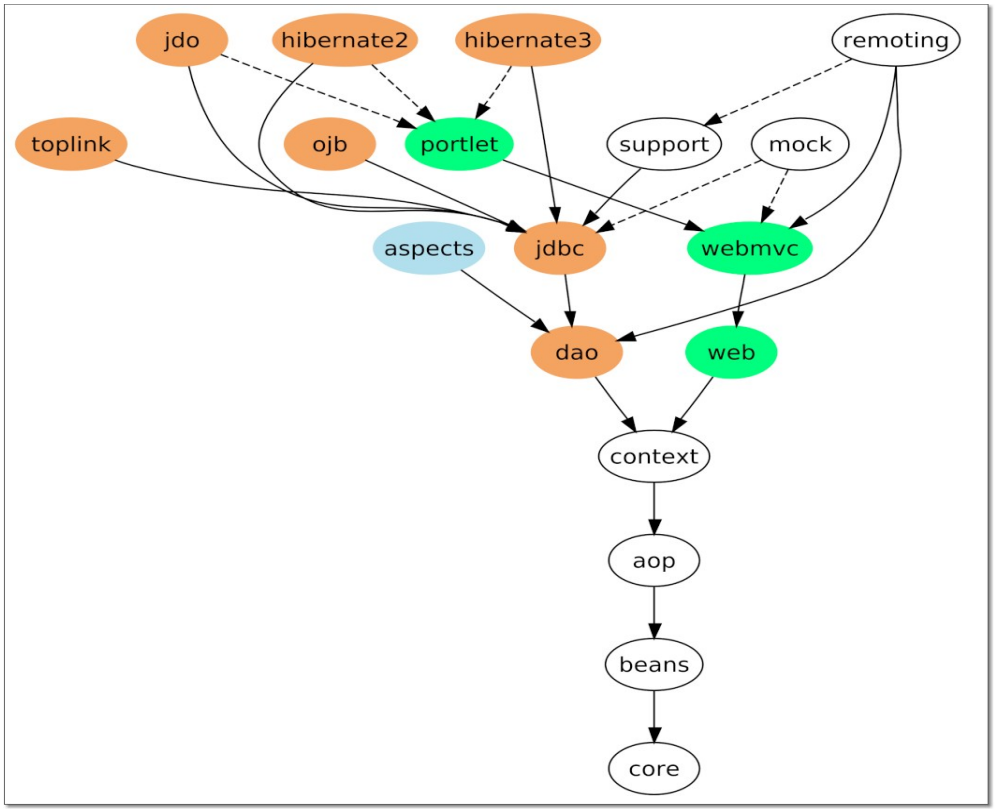



*Figure 8-1: Dependency relationship between Spring modules*

In figure 8-1. you can see graphically the dependencies between the modules. Optional dependencies are indicated by dotted lines.

Each of these modules corresponds, more or less, with the Java package structure, and each produces a JAR. These modules are built with an Ant script from the following source directories:

- `src` and `test`: contain JDK 1.4 compatible source code and JUnit tests respectively
- `tiger/src and tiger/test`: contain additional JDK 1.5 compatible source code and JUnit tests
- `mock`: contains the source code for the spring-mock module
- `aspectj/src and aspectj/test`: contain the source code for the spring-aspects module

Each of the source directories also include classpath resources (XML files, properties files, TLD files, etc.).

For Spring, the Ant script compiles each of these different source directories and then creates a JAR for each module, using inclusions and exclusions that are based on the Java packages of each class. The `src` and `tiger/src` directories are compiled to the same destination as the test and `tiger/test` directories, resulting in JARs that contain both 1.4 and 1.5 classes.

## 8.2. Where to Begin?

With Maven, the rule of thumb to use is to produce one artifact (JAR, WAR, etc.) per Maven project file. In the Spring example, that means you will need to have a Maven project (a POM) for each of the modules listed above.

To start, you will create a subdirectory called '`m2`' to keep all the necessary Maven changes clearly separated from the current build system. Inside the '`m2`' directory, you will need to create a directory for each of Spring's modules.

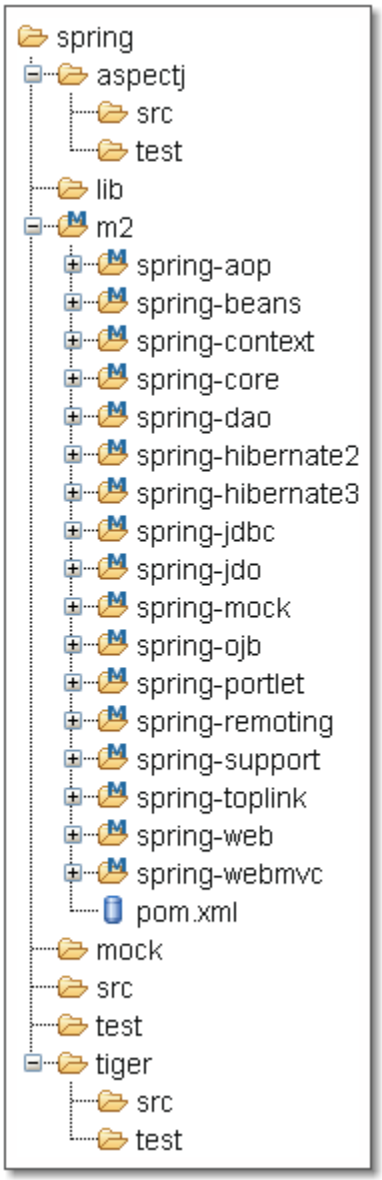



*Figure 8-2: A sample spring module directory*

In the `m2` directory, you will need to create a parent POM. You will use the parent POM to store the common configuration settings that apply to all of the modules. For example, each module will inherit the following values (settings) from the parent POM.

- `groupId`: this setting indicates your area of influence, company, department, project, etc., and it should mimic standard package naming conventions to avoid duplicate values. For this example, you will use `com.mergere.m2book.migrating`, as it is our 'unofficial' example version of Spring; however, the Spring team would use `org.springframework`
- `artifactId`: the setting specifies the name of this module (for example, `spring-parent`)
- `version`: this setting should always represent the next release version number appended with - *`SNAPSHOT`* – that is, the version you are developing in order to release. Recall from previous chapters that during the release process, Maven will convert to the definitive, non-snapshot version for a short period of time, in order to tag the release in your SCM.
- `packaging`: the `jar`, `war`, and `ear` values should be obvious to you (a `pom` value means that this project is used for metadata only)

The other values are not strictly required, primary used for documentation purposes.

```
<groupId>com.mergere.m2book.migrating</groupId>
<artifactId>spring-parent</artifactId>
<version>2.0-m1-SNAPSHOT</version>
<name>Spring parent</name>
<packaging>pom</packaging>
<description>Spring Framework</description>
<inceptionYear>2002</inceptionYear>
<url>http://www.springframework.org</url>
<organization>
 <name>The Spring Framework Project</name>
</organization>
```

In this parent POM we can also add dependencies such as JUnit, which will be used for testing in every module, thereby eliminating the requirement to specify the dependency repeatedly across multiple modules.

```
<dependencies>
 <dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
 </dependency>
</dependencies>
```

As explained previously, in Spring, the main source and test directories are `src` and `test`, respectively. Let's begin with these directories.

Using the following code snippet from Spring's Ant build script, in the `buildmain` target, you can retrieve some of the configuration parameters for the compiler.

```
<javac destdir="${target.classes.dir}" source="1.3" target="1.3" debug="${debug}"
    deprecation="false" optimize="false" failonerror="true">
 <src path="${src.dir}"/>
 <!-- Include Commons Attributes generated Java sources -->
 <src path="${commons.attributes.tempdir.src}"/>
 <classpath refid="all-libs"/>
</javac>
```

As you can see these include the source and target compatibility (**1.3**), deprecation and optimize (**false**), and `failonerror` (**true**) values. These last three properties use Maven's default values, so there is no need for you to add the configuration parameters.

For the debug attribute, Spring's Ant script uses a debug parameter, so to specify the required debug function in Maven, you will need to append `-Dmaven.compiler.debug=false` to the `mvn` command (by default this is set to true). For now, you don't have to worry about the commons-attributes generated sources mentioned in the snippet, as you will learn about that later in this chapter. Recall from Chapter 2, that Maven automatically manages the classpath from its list of dependencies.

At this point, your build section will look like this:

```
<build>
 <sourceDirectory>../../src</sourceDirectory>
 <testSourceDirectory>../../test</testSourceDirectory>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <source>1.3</source>
    <target>1.3</target>
   </configuration>
  </plugin>
 </plugins>
</build>
```

The other configuration that will be shared is related to the JUnit tests. From the `tests` target in the Ant script:

```
<junit forkmode="perBatch" printsummary="yes" haltonfailure="yes"
haltonerror="yes">

 <jvmarg line="-Djava.awt.headless=true -XX:MaxPermSize=128m -Xmx128m"/>

 <!-- Must go first to ensure any jndi.properties files etc take precedence -->
 <classpath location="${target.testclasses.dir}"/>
 <classpath location="${target.mockclasses.dir}"/>
 <classpath location="${target.classes.dir}"/>

 <!-- Need files loaded as resources -->
 <classpath location="${test.dir}"/>

 <classpath refid="all-libs"/>

 <formatter type="plain" usefile="false"/>
 <formatter type="xml"/>

 <batchtest fork="yes" todir="${reports.dir}">
  <fileset dir="${target.testclasses.dir}" includes="${test.includes}"
excludes="${test.excludes}"/>
 </batchtest>
</junit>
```

You can extract some configuration information from the previous code:

- `forkMode`=”`perBatch`” matches with Maven's `forkMode` parameter with a value of once, since the concept of a batch for testing does not exist.
- You will not need any `printsummary`, `haltonfailure` and `haltonerror` settings, as Maven prints the test summary and stops for any test error or failure, by default.
- `formatter` elements are not required as Maven generates both plain `text` and `xml` reports.
- The nested element `jvmarg` is mapped to the configuration parameter `argLine`
- As previously noted, `classpath` is automatically managed by Maven from the list of dependencies.
- Maven sets the reports destination directory (`todir`) to `target/surefire-reports`, by default, and this doesn't need to be changed.
- You will need to specify the value of the properties `test.includes` and `test.excludes` from the nested `fileset`; this value is read from the `project.properties` file loaded from the Ant script (refer to the code snippet below for details).
- Maven uses the default value from the compiler plugin, so you will not need to locate the test classes directory (`dir`).

```
# Wildcards to be matched by JUnit tests.
# Convention is that our JUnit test classes have XXXTests-style names.
test.includes=**/*Tests.class
#
# Wildcards to exclude among JUnit tests.
# Second exclude needs to be used for JDK 1.3, due to Hibernate 3.1
# being compiled with target JDK 1.4.
test.excludes=**/Abstract*
#test.excludes=**/Abstract* org/springframework/orm/hibernate3/**
```

The includes and excludes referenced above, translate directly into the include/exclude elements of the POM's plugin configuration.

> Since Maven requires JDK 1.4 to run you do not need to exclude `hibernate3` tests. Note that it is possible to use another lower JVM to run tests if you wish – refer to the Surefire plugin reference documentation for more information.

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-plugin</artifactId>
 <configuration>
  <forkMode>once</forkMode>
  <childDelegation>false</childDelegation>
  <argLine>
   -Djava.awt.headless=true -XX:MaxPermSize=128m -Xmx128m
  </argLine>
  <includes>
   <include>**/*Tests.class</include>
  </includes>
  <excludes>
   <exclude>**/Abstract*</exclude>
  </excludes>
 </configuration>
</plugin>
```

The `childDelegation` option is required to prevent conflicts when running under Java 5 between the XML parser provided by the JDK and the one included in the dependencies in some modules, mandatory when building in JDK 1.4. It makes tests run using the standard `classloader` delegation instead of the default Maven isolated `classloader`. When building only on Java 5 you could remove that option and the XML parser (`Xerces`) and APIs (`xml-apis`) dependencies.

Spring's Ant build script also makes use of the commons-attributes compiler in its `compileattr` and `compiletestattr` targets, which are processed prior to the compilation. The commons-attributes compiler processes javadoc style annotations – it was created before Java supported annotations in the core language on JDK 1.5 - and generates sources from them that have to be compiled with the normal Java compiler.

From `compileattr`:

```
<!-- Compile to a temp directory: Commons Attributes will place Java Source here.
-->
<attribute-compiler destdir="${commons.attributes.tempdir.src}">
 <!--
 Only the PathMap attribute in the org.springframework.web.servlet.handler.metadata
 package currently needs to be shipped with an attribute, to support indexing.
 -->
 <fileset dir="${src.dir}" includes="**/metadata/*.java"/>
</attribute-compiler>
```

From `compiletestattr`:

```
<!-- Compile to a temp directory: Commons Attributes will place Java Source here.
-->
<attribute-compiler destdir="${commons.attributes.tempdir.test}">
 <fileset dir="${test.dir}" includes="org/springframework/aop/**/*.java"/>
 <fileset dir="${test.dir}" includes="org/springframework/jmx/**/*.java"/>
</attribute-compiler>
```

In Maven, this same function can be accomplished by adding the commons-attributes plugin to the build section in the POM. Maven handles the source and destination directories automatically, so you will only need to add the inclusions for the main source and test source compilation.

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>commons-attributes-maven-plugin</artifactId>
 <executions>
  <execution>
   <configuration>
    <includes>
     <include>**/metadata/*.java</include>
    </includes>
    <testIncludes>
     <include>org/springframework/aop/**/*.java</include>
     <include>org/springframework/jmx/**/*.java</include>
    </testIncludes>
   </configuration>
   <goals>
    <goal>compile</goal>
    <goal>test-compile</goal>
    </goals>
  </execution>
 </executions>
</plugin>
```

Later in this chapter you will need to modify these test configurations.

## 8.3. Creating POM files

Now that you have the basic configuration shared by all modules (project information, compiler configuration, JUnit test configuration, etc.), you need to create the POM files for each of Spring's modules. In each subdirectory, you will need to create a POM that extends the parent POM.

The following is the POM for the spring-core module. This module is the best to begin with because all of the other modules depend on it.

```
<parent>
 <groupId>com.mergere.m2book.migrating</groupId>
 <artifactId>spring-parent</artifactId>
 <version>2.0-m1-SNAPSHOT</version>
</parent>
<artifactId>spring-core</artifactId>
<name>Spring core</name>
```

Again, you won't need to specify the version or `groupId` elements of the current module, as those values are inherited from parent POM, which centralizes and maintains information common to the project.

## 8.4. Compiling

In this section, you will start to compile the main Spring source; tests will be dealt with later in the chapter. To begin, review the following code snippet from Spring's Ant script, where `spring-core` JAR is created:

```
<jar jarfile="${dist.dir}/modules/spring-core.jar">
  <fileset dir="${target.classes.dir}">
    <include name="org/springframework/core/**"/>
    <include name="org/springframework/util/**"/>
  </fileset>
  <manifest>
    <attribute name="Implementation-Title" value="${spring-title}"/>
  <attribute name="Implementation-Version" value="${spring-version}"/>
    <attribute name="Spring-Version" value="${spring-version}"/>
  </manifest>
</jar>
```

From the previous code snippet, you can determine which classes are included in the JAR and what attributes are written into the JAR's manifest. Maven will automatically set manifest attributes such as name, version, description, and organization name to the values in the POM. While manifest entries can also be customized with additional configuration to the JAR plugin, in this case the defaults are sufficient. However, you will need to tell Maven to pick the correct classes and resources from the `core` and `util` packages.

For the resources, you will need to add a `resources` element in the build section, setting the files you want to include (by default Maven will pick everything from the resource directory). As you saw before, since the sources and resources are in the same directory in the current Spring build, you will need to exclude the `*.java` files from the resources, or they will get included in the JAR.

For the classes, you will need to configure the compiler plugin to include only those in the `core` and `util` packages, because as with resources, Maven will by default compile everything from the source directory, which is inherited from the parent POM.

```
<build>
 <resources>
  <resource>
   <directory>../../src</directory>
   <includes>
     <include>org/springframework/core/**</include>
     <include>org/springframework/util/**</include>
   </includes>
   <excludes>
     <exclude>**/*.java</exclude>
   </excludes>
  </resource>
 </resources>
 <plugins>
 <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <includes>
     <include>org/springframework/core/**</include>
     <include>org/springframework/util/**</include>
    </includes>
   </configuration>
  </plugin>
 </plugins>
</build>
```

To compile your Spring build, you can now run `mvn compile`. You will see a long list of compilation failures, beginning with the following:

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[30,34] package org.apache.commons.logging does not exist

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[31,34] package org.apache.commons.logging does not exist

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\core\io\support\PathMatchingResourcePatternResol
ver.java:[107,24] cannot find symbol
symbol : class Log
location: class
org.springframework.core.io.support.PathMatchingResourcePatternResolver

C:\dev\m2book\code\migrating\spring\m2\spring-
core\..\..\src\org\springframework\util\xml\SimpleSaxErrorHandler.java:[19,34]
package org.apache.commons.logging does not exist
```

These are typical compiler messages, caused by the required classes not being on the classpath.

From the previous output, you now know that you need the Apache Commons Logging library (`commons-logging`) to be added to the dependencies section in the POM. But, what `groupid`, `artifactId` and version should we use?

For the `groupId` and `artifactId`, you need to check the central repository at ibiblio. Typically, the convention is to use a `groupId` that mirrors the package name, changing dots to slashes. For example, `commons-logging` `groupId` would become `org.apache.commons.logging`, located in the `org/apache/commons/logging` directory in the repository.

However, for historical reasons some `groupId` values don't follow this convention and use only the name of the project. In the case of commons-logging, the actual `groupId` is commons-logging.

Regarding the `artifactId`, it's usually the JAR name without a version (in this case commons-logging). If you check the repository, you will find all the available versions of commons-logging under http://www.ibiblio.org/maven2/commons-logging/commons-logging/.

As an alternative, you can search the repository using Google. Specify `site:www.ibiblio.org/maven2` commons logging, and then choose from the search results, the option that is closest to what is required by your project.

With regard the version, you will find that there is documentation for all of Spring's dependencies in `readme.txt` in the lib directory of the Spring source. You can use this as a reference to determine the versions of each of the dependencies. However, you have to be careful as the documentation may contain mistakes and/or inaccuracies. For instance, during the process of migrating Spring to Maven, we discovered that the commons-beanutils version stated in the documentation is wrong and that some required dependencies are missing from the documentation.

```
<dependencies>
 <dependency>
  <groupId>commons-logging</groupId>
   <artifactId>commons-logging</artifactId>
  <version>1.0.4</version>
 </dependency>
</dependencies>
```

Usually you will convert your own project, so you will have first hand knowledge about the dependencies and versions used. When needed, there are some other options to try to determine the appropriate versions for the dependencies included in your build:

- Check if the JAR has the version in the file name
- Open the JAR file and look in the manifest file `META-INF/MANIFEST.MF`
- For advanced users, search the ibiblio repository through Google by calculating the MD5 checksum of the JAR file with a program such as md5sum, and then search in Google pre-pending `site:www.ibiblio.org/maven2` to the query. For example, for the `hibernate3.jar` provided with Spring under lib/hibernate, you could search with: site:www.ibiblio.org/maven2 78d5c38f1415efc64f7498f828d8069a

  The search will return: www.ibiblio.org/maven2/org/hibernate/hibernate/3.1/hibernate-3.1.jar.md5

You can see that the last directory is the version (3.1), the previous directory is the `artifactId` (hibernate) and the other directories compose the `groupId`, with the slashes changed to dots (`org.hibernate`)

---

An easier way to search for dependencies, using a web interface, is under development (refer to the Maven Repository Manager project for details).

While adding dependencies can be the most painful part of migrating to Maven, explicit dependency management is one of the biggest benefits of Maven once you have invested the effort upfront. So, although you could simply follow the same behavior used in Ant (by adding all the dependencies in the parent POM so that, through inheritance, all sub-modules would use the same dependencies), we strongly encourage and recommend that you invest the time at the outset of your migration, to make explicit the dependencies and interrelationships of your projects. Doing so will result in cleaner, component-oriented, modular projects that are easier to maintain in the long term.

---

Running again `mvn` compile and repeating the process previously outlined for commons-logging, you will notice that you also need Apache Commons Collections (aka commons-collections) and `log4j`.

```
<dependency>
 <groupId>commons-collections</groupId>
 <artifactId>commons-collections</artifactId>
 <version>3.1</version>
</dependency>
<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.9</version>
 <optional>true</optional>
</dependency>
```

Notice that `log4j` is marked as optional. Optional dependencies are not included transitively, so log4j will not be included in other projects that depend on this. This is because in other projects, you may decide to use another log implementation, and it is just for the convenience of the users. Using the optional tag does not affect the current project.

Now, run `mvn compile` again - this time all of the sources for spring-core will compile.

## 8.5. Testing

Now you're ready to compile and run the tests. For the first step, you will repeat the previous procedure for the main classes, setting which test resources to use, and setting the JUnit test sources to compile. After compiling the tests, we will cover how to run the tests.

### 8.5.1. Compiling Tests

Setting the test resources is identical to setting the main resources, with the exception of changing the location from which the element name and directory are pulled. In addition, you will need to add the log4j.properties file required for logging configuration.

```
<testResources>
 <testResource>
  <directory>../../test</directory>
  <includes>
    <include>log4j.properties</include>
    <include>org/springframework/core/**</include>
    <include>org/springframework/util/**</include>
  </includes>
  <excludes>
    <exclude>**/*.java</exclude>
  </excludes>
 </testResource>
</testResources>
```

Setting the test sources for compilation follows the same procedure, as well. Inside the maven-compiler-plugin configuration, you will need to add the testIncludes element.

```
<testIncludes>
 <include>org/springframework/core/**</include>
 <include>org/springframework/util/**</include>
</testIncludes>
```

You may also want to check the Log4JConfigurerTests.java class for any hard codes links to properties files and change them accordingly.

Now, if you try to compile the test classes by running mvn test-compile, as before, you will get compilation errors, but this time there is a special case where the compiler complains because some of the classes from the org.springframework.mock, org.springframework.web and org.springframework.beans packages are missing. It may appear initially that spring-core depends on spring-mock, spring-web and spring-beans modules, but if you try to compile those other modules, you will see that their main classes, not tests, depend on classes from spring-core. As a result, we cannot add a dependency from spring-core without creating a circular dependency. In other words, if spring-core depends on spring-beans and spring-beans depends on spring-core, which one do we build first? Impossible to know.

So, the key here is to understand that some of the test classes are not actually unit tests for spring-core, but rather require other modules to be present. Therefore, it makes sense to exclude all the test classes that reference other modules from this one and include them elsewhere.

To exclude test classes in Maven, add the testExcludes element to the compiler configuration as follows.

```
<testExcludes>
 <exclude>org/springframework/util/comparator/ComparatorTests.java</exclude>
 <exclude>org/springframework/util/ClassUtilsTests.java</exclude>
 <exclude>org/springframework/util/ObjectUtilsTests.java</exclude>
 <exclude>org/springframework/util/ReflectionUtilsTests.java</exclude>
 <exclude>org/springframework/util/SerializationTestUtils.java</exclude>
 <exclude>org/springframework/core/io/ResourceTests.java</exclude>
</testExcludes>
```

Now, when you run mvn test-compile, you will see the following error:

```
package javax.servlet does not exist
```

This means that the following dependency must be added to the POM, in order to compile the tests:

```
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>servlet-api</artifactId>
 <version>2.4</version>
 <scope>test</scope>
</dependency>
```

The scope is set to test, as this is not needed for the main sources.

If you run mvn test-compile again you will have a successful build, as all the test classes compile correctly now.

## 8.5.2. Running Tests

Running the tests in Maven, simply requires running `mvn test`. However, when you run this command, you will get the following error report:

```
Results :
[surefire] Tests run: 113, Failures: 1, Errors: 1

[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] There are test failures.
[INFO] ------------------------------------------------------------------------
```

Upon closer examination of the report output, you will find the following:

```
[surefire] Running
org.springframework.core.io.support.PathMatchingResourcePatternResolverTests
[surefire] Tests run: 5, Failures: 1, Errors: 1, Time elapsed: 0.015 sec <<<<<<<<
FAILURE !!
```

This output means that this test has logged a JUnit failure and error. To debug the problem, you will need to check the test logs under target/surefire-reports, for the test class that is failing `org.springframework.core.io.support.PathMatchingResourcePatternResolverTests.txt`. Within this file, there is a section for each failed test called `stacktrace`.

The first section starts with `java.io.FileNotFoundException: class path resource [org/aopalliance/]` cannot be resolved to URL because it does not exist.

This indicates that there is something missing in the classpath that is required to run the tests. The `org.aopalliance` package is inside the `aopallience` JAR, so to resolve the problem add the following to your POM

```
<dependency>
 <groupId>aopalliance</groupId>
 <artifactId>aopalliance</artifactId>
 <version>1.0</version>
 <scope>test</scope>
</dependency>
```

Now run `mvn test` again. You will get the following wonderful report:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

The last step in migrating this module (spring-core) from Ant to Maven, is to run `mvn install` to make the resulting JAR available to other projects in your local Maven repository. This command can be used instead most of the time, as it will process all of the previous phases of the build life cycle (generate sources, compile, compile tests, run tests, etc.)

## 8.6. Other Modules

Now that you have one module working it is time to move on to the other modules. If you follow the order of the modules described at the beginning of the chapter you will be fine, otherwise you will find that the main classes from some of the modules reference classes from modules that have not yet been built. See figure 8-1 to get the overall picture of the interdependencies between the Spring modules.

### 8.6.1. Avoiding Duplication

As soon as you begin migrating the second module, you will find that you are repeating yourself. For instance, you will be adding the Surefire plugin configuration settings repeatedly for each module that you convert. To avoid duplication, move these configuration settings to the parent POM instead. That way, each of the modules will be able to inherit the required Surefire configuration.

In the same way, instead of repeatedly adding the same dependency version information to each module, use the parent POM's `dependencyManagement` section to specify this information once, and remove the versions from the individual modules (see Chapter 3 for more information).

Using the parent POM to centralize this information makes it possible to upgrade a dependency version across all sub-projects from a single location.

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>commons-logging</groupId>
   <artifactId>commons-logging</artifactId>
   <version>1.0.4</version>
  </dependency>
</dependencyManagement>
```

The following are some variables that may also be helpful to reduce duplication:

- `${project.version}`: version of the current POM being built
- `${project.groupId}`: `groupId` of the current POM being built

For example, you can refer to `spring-core` from `spring-beans` with the following, since they have the same `groupId` and `version`:

```
<dependency>
 <groupId>${project.groupId}</groupId>
 <artifactId>spring-core</artifactId>
 <version>${project.version}</version>
</dependency>
```

## 8.6.2. Referring to Test Classes from Other Modules

If you have tests from one component that refer to tests from other modules, there is a procedure you can use. Although it is typically not recommended, in this case it is necessary to avoid refactoring the test source code. First, make sure that when you run `mvn install`, that a JAR that contains the test classes is also installed in the repository:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <executions>
  <execution>
   <goals>
    <goal>test-jar</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

Once that JAR is installed, you can use it as a dependency for other components, by specifying the `test-jar` type. However, be sure to put that JAR in the test scope as follows:

```
<dependency>
 <groupId>${project.groupId}</groupId>
 <artifactId>spring-beans</artifactId>
 <version>${project.version}</version>
 <type>test-jar</type>
 <scope>test</scope>
</dependency>
```

A final note on referring to test classes from other modules: if you have all of Spring's mock classes inside the same module, this can cause previously-described cyclic dependencies problem. To eliminate this problem, you can split Spring's mock classes into `spring-context-mock`, with only those classes related to `spring-context` module, and `spring-web-mock`, with only those classes related to `spring-web`. Generally with Maven, it's easier to deal with small modules, particularly in light of transitive dependencies.

## 8.6.3. Building Java 5 Classes

Some of Spring's modules include Java 5 classes from the tiger folder. As the compiler plugin was earlier configured to compile with Java 1.3 compatibility, how can the Java 1.5 sources be added? To do this with Maven, you need to create a new module with only Java 5 classes instead of adding them to the same module and mixing classes with different requirements. So, you will need to create a new `spring-beans-tiger` module.

Consider that if you include some classes compiled for Java 1.3 and some compiled for Java 5 in the same JAR, any users, attempting to use one of the Java 5 classes under Java 1.3 or 1.4, would experience runtime errors. By splitting them into different modules, users will know that if they depend on the module composed of Java 5 classes, they will need to run them under Java 5.

As with the other modules that have been covered, the Java 5 modules will share a common configuration for the compiler. The best way to split them is to create a tiger folder with the Java 5 parent POM, and then a directory for each one of the individual tiger modules, as follows:

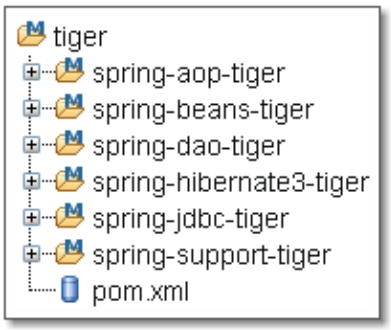
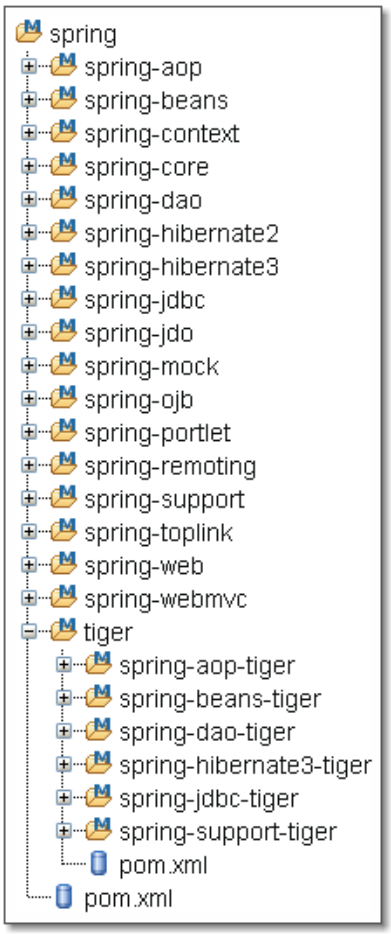



*Figure 8-3: A tiger module directory*




The final directory structure should appear as follows:

*Figure 8-4: The final directory structure*

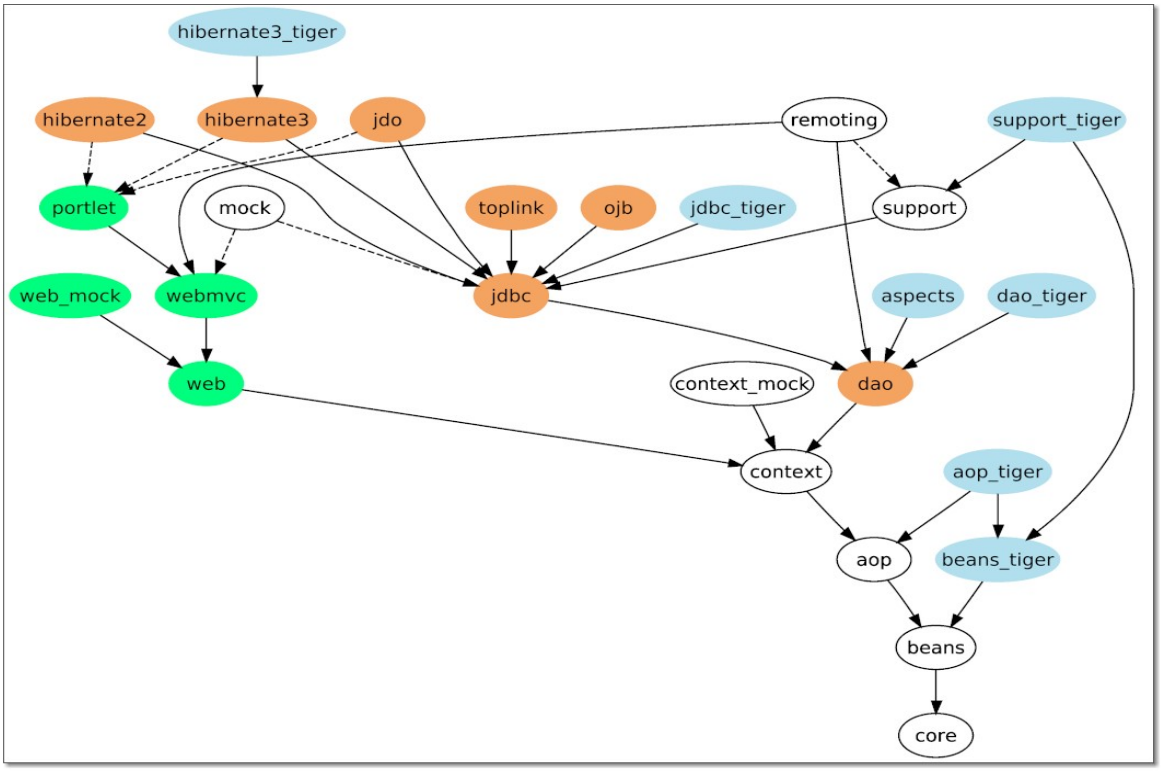



*Figure 8-5: Dependency relationship, with all modules*

In the tiger POM, you will need to add a module entry for each of the directories, as well as build sections with source folders and compiler options:

```
<modules>
 <module>spring-beans-tiger</module>
 <module>spring-aop-tiger</module>
 <module>spring-dao-tiger</module>
 <module>spring-jdbc-tiger</module>
 <module>spring-support-tiger</module>
 <module>spring-hibernate3-tiger</module>
 <module>spring-aspects</module>
</modules>
<build>
 <sourceDirectory>../../../tiger/src</sourceDirectory>
 <testSourceDirectory>../../../tiger/test</testSourceDirectory>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
    <source>1.5</source>
    <target>1.5</target>
   </configuration>
  </plugin>
 </plugins>
</build>
```

In the parent POM, you just need a new module entry for the tiger folder, but to still be able to build the other modules when using Java 1.4 you will add that module in a profile that will be triggered only when using 1.5 JDK.

```
<profiles>
 <profile>
  <id>jdk1.5</id>
  <activation>
   <jdk>1.5</jdk>
  </activation>
  <modules>
   <module>tiger</module>
  </modules>
 </profile>
</profiles>
```

## 8.6.4. Using Ant Tasks From Inside Maven

In certain migration cases, you may find that Maven does not have a plugin for a particular task or an Ant target is so small that it may not be worth creating a new plugin. In this case, Maven can call Ant tasks directly from a POM using the `maven-antrun-plugin`.

For example, with the Spring migration, you need to use the Ant task in the `spring-remoting` module to use the RMI compiler.

From Ant, this is:

```
<rmic base="${target.classes.dir}"
  classname="org.springframework.remoting.rmi.RmiInvocationWrapper"/>

<rmic base="${target.classes.dir}"
  classname="org.springframework.remoting.rmi.RmiInvocationWrapper" iiop="true">
  <classpath refid="all-libs"/>
</rmic>
```

To include this in Maven build, add:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
   <execution>
    <phase>process-classes</phase>
    <configuration>
     <tasks>
      <echo>Running rmic</echo>
      <rmic base="${project.build.directory}/classes"
            classname="org.springframework.remoting.rmi.RmiInvocationWrapper"/>
      <rmic base="${project.build.directory}/classes"
            classname="org.springframework.remoting.rmi.RmiInvocationWrapper"
            iiop="true">
       <classpath refid="maven.compile.classpath"/>
      </rmic>
     </tasks>
    </configuration>
    <goals>
     <goal>run</goal>
    </goals>
   </execution>
  </executions>
  <dependencies>
   <dependency>
    <groupId>com.sun</groupId>
    <artifactId>tools</artifactId>
    <scope>system</scope>
    <version>1.4</version>
    <systemPath>${java.home}/../lib/tools.jar</systemPath>
   </dependency>
  </dependencies>
 </plugin>
```

As shown in the code snippet above, there are some references available already, such as `${project.build.directory}` and `maven.compile.classpath`, which is a classpath reference constructed from all of the dependencies in the compile scope or lower. There are also references for anything that was added to the plugin's dependencies section, which applies to that plugin only, such as the reference to the `tools.jar` above, which is bundled with the JDK, and required by the RMI task.

To complete the configuration, you will need to determine when Maven should run the Ant task. In this case, the `rmic` task, will take the compiled classes and generate the `rmi` skeleton, stub and tie classes from them. So, the most appropriate phase in which to run this Ant task is in the process-classes phase.

### 8.6.5. Non-redistributable Jars

You will find that some of the modules in the Spring build depend on JARs that are not available in the Maven central repository. For example, Sun's Activation Framework and JavaMail are not redistributable from the repository due to constraints in their licenses. You may need to download them yourself from the Sun site or get them from the lib directory in the example code for this chapter. You can then install them in your local repository with the following command.

```
mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id>
-DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging>
```

For instance, to install JavaMail:

```
mvn install:install-file -Dfile=mail.jar -DgroupId=javax.mail
-DartifactId=mail -Dversion=1.3.2 -Dpackaging=jar
```

You will only need to do this process once for all of your projects or you may use a corporate repository to share them across your organization. For more information on dealing with this issue, see http://maven.apache.org/guides/mini/guide-coping-with-sun-jars.html.

### 8.6.6. Some Special Cases

In addition to the procedures outlined previously for migrating Spring to Maven, there are two additional, special cases that must be handled. These issues were shared with the Spring developer community and are listed below:

- Moving one test class, `NamespaceHandlerUtilsTests`, from test directory to `tiger/test_` directory – as it depends on a tiger class
- Fixing toplink tests that don't compile against the oracle toplink jar (Spring developers use a different version than the official one from Oracle)

In this example it is necessary to comment out two unit tests, which used relative paths in `Log4JConfigurerTests` class, as these test cases will not work in both Maven and Ant. Using classpath resources is recommended over using file system resources.

> There is some additional configuration required for some modules, such as `spring-aspects`, which uses AspectJ for weaving the classes. These can be viewed in the example code.

## 8.7. Restructuring the Code

If you do decide to use Maven for your project, it it highly recommended that you go through the restructuring process to take advantage of the many timesaving and simplifying conventions within Maven.

For example, for the `spring-core` module, you would move all Java files under `org/springframework/core` and `org/springframework/util` from the original `src` folder to the module's folder `src/main/java`.

All of the other files under those two packages would go to `src/main/resources`. The same for tests, these would move from the original test folder to `src/test/java` and `src/test/resources` respectively for Java sources and other files - just remember not to move the excluded tests (`ComparatorTests`, `ClassUtilsTests`, `ObjectUtilsTests`, `ReflectionUtilsTests`, `SerializationTestUtils` and `ResourceTests`).

By adopting Maven's standard directory structure, you can simplify the POM significantly, reducing its size by two-thirds!

## 8.8. Summary

By following and completing this chapter, you will be able to take an existing Ant-based build, split it into modular components (if needed), compile and test the code, create JARs, and install those JARs in your local repository using Maven. At the same time, you will be able to keep your current build working. Once you decide to switch completely to Maven, you will be able to take advantage of the benefits of adopting Maven's standard directory structure. By doing this, you would eliminate the need to include and exclude sources and resources “by hand” in the POM files as shown in this chapter.

Once you have spent this initial setup time Maven, you can realize Maven' other benefits - advantages such as built-in project documentation generation, reports, and quality metrics.

Finally, in addition to the improvements to your build life cycle, Maven can eliminate the requirement of storing jars in a source code management system. In the case of the Spring example, as Maven downloads everything it needs and shares it across all your Maven projects automatically - you can delete that 80 MB lib folder.

Now that you have seen how to do this for Spring, you can apply similar concepts to your own Ant based build.