Developing and maintaining a mixed Java/Groovy software using Maven

Tiziano Fagni
Network Multimedia Information Systems (NeMIS) laboratory,
ISTI - CNR, Pisa, Italy,
tiziano.fagni@isti.cnr.it

July 6, 2013

Abstract

Developing and maintaining a complex software today is not a trivial task especially if you don't use the proper tools. This is even more true if your software depends on a lot of external libraries/frameworks. In such cases, a special care must be taken in order to proper handle all the third-party dependencies required by the software you are building. Maven is a build automation and project management tool that automatically manages the dependencies of a software in a project and that provide a lot of facilities to support the developers in all phases of software development (building, testing, packaging) and communications (e.g. generating reports, project web site, etc.). In this article, we will show you how Maven can be used effectively to support the application development in mixed projects which use the Java/Groovy languages, two very popular programming languages among the developers.

1 Introduction

Today develop a new software in the Java world is quite easy. The Java language is mature and feature rich with a very complete standard library (known as JDK). If the JDK is not sufficient, there are a lot of open source or closed third-party libraries which can allows the developers to add specific features to their software very easily. As developer, the real challenge is not only write code which does something but ensure that all the code you write and integrate works correctly over time. You must not spend time by hacking the way you compile the source code or asking yourself which is the best way to integrate the testing phase in your project. You should use some tool which standardize in some way these tasks and let you concentrate on just what it does matter. If you try to write applications/libraries with mixed source code (like Java and Groovy), the problems described above are amplified. In these cases, a project management tool able to drive the development process is simply a necessity. Apache Maven seems to be a very good answer to these requirements.

The aim of the article is to show how to use Maven to develop/mantain sofware project written by using both Java and Groovy languages. For these

purposes, in the next sections, we will briefly describe the programming languages Java and Groovy, then we will see in more depth how Maven works, how to configure it and how to use it to mantain the software projects on which you are working.

2 The ingredients of this technological recipe

2.1 The Apache Maven tool

Maven [2] is a powerful build automation tool used primarily for Java projects. Maven serves a similar purpose to the Apache Ant [1] tool, but it is based on different concepts and works in a different manner. Maven in a certain sense is a more complete tool than Ant and it provides for free a lot of features that in Ant are not available or are difficult to achieve. In particular, Maven main benefits are the following:

- Making the build process easy. In particular, building or deploying a software is just a pair of shell commands away and a lot of complications are hidden to the developers.
- Automatic dependencies management. Generally, it is sufficient to declare which are the direct libraries from which your software depends on and Maven automatically downloads all the declared dependencies (in a transitive way!) inside your project. Maven knows where to find all the declared libraries because has access to one ore more global software repositories (this is configurable by the developers) containing all the libraries commonly used or private and specific to certain projects.
- Providing a uniform build system. Maven uses the POM (Project Object Model) to describe completely a project. The POM is an XML file where the developers put all the informations related to the project. These information include the name and version of the software, the set of dependencies of the project, the set of software repositories in use, where to find sources, etc. Given all these info, Maven provides a set of predefined goals (i.e. target command) to build and deploy software. These goals can be customized or enriched by using specific plugins. Each plugin can modify the default behaviour or add new features to the building and deployment process. In general, you can use very similar POM for very similar project, so a POM specific for a certain type of software (e.g. Web apps, Jar library, etc.) can be easily reused multiple times with minor customization in several projects.
- Automatic testing during deployment of the software. During the build process, Maven can be configured to execute all the required tests and to force the stop of building process in case on some failure. This can be very useful to avoid deploying buggy software with unwanted runtime errors and inconsistencies.
- Providing guidelines for best practices development. Maven encourages the developers to follow good and effective rules while developing the software. For example, the default project folders organization already propose standard directories for sources (both the software source and test

source) and resources (both the software resources and test resources) which clearly drives developers to put things in the right places.

2.2 The Java language

Java [6] was proposed by Sun in 1995 at time when Internet started to have growing popularity among people. According to several sources [3, 5], today is one of the most popular general purpose programming languages and it is used to develop the most varied types of software which run in a multitude of different devices and operating systems. The great success of Java is due to several factors. Among these, the most important are:

- Executable portability. The Java slogan is "Write once, runs everywhere": it basically means that in principle you write the application, compile it just one time and runs it on every system where a compatible Java Virtual Machine (JVM) is available.
- Easy syntax and object-oriented programming. The syntax is very easy to learn and it is similar to the syntax found in C and C++, which were very popular before the introduction of Java. Moreover, the language is completely object-oriented with a logical model based on simple single inheritance and extensive use of interfaces to define common behaviour for objects.
- Automatic memory management. Java allocates automatically the memory required by the code that is running and deallocates it when it is no more referenced by the software (by providing an internal garbage collector). In this way and differently from languages like C or C++, the programmer is free from correctly managing the memory and can concentrating on writing better code.
- Make distributed computing easy. Java is designed to make distributed computing easy with the networking capability that is inherently integrated into it.
- Security and robustness. Java was one of the first languages that considered security as a first citizen, providing concept like sandbox to allow the usage of resources in a controlled environment or to only allow the execution of software cryptographically signed. The robustness is another important aspect handled by Java language which provides a robust mechanism of exception handling.
- Standard platform library. Java provides for free a big standard library which covers a lot of specificity occurring in normal development process: string processing, I/O, security, collections management of data, etc.

Here it is a simple and revised classic "HelloWorld" in Java:

Listing 1: A revised classic "HelloWorld" in Java

```
import java.lang.*;

public class Hello {
  private String name = "Noname";
```

```
  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  public void sayHello() {
    System.out.println("Hello world, "+
      getName()+"!");
  }

  public static void main(String[] args) {
    Hello h = new Hello();
    hello.setName("Tiziano");
    hello.sayHello();
  }
}
```

2.3 The Groovy language

Groovy is an object-oriented programming language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as a scripting language for the Java Platform, it is dynamically compiled to Java Virtual Machine (JVM) bytecode, and interoperates perfectly with other Java code and compiled JVM libraries. Groovy uses a Java-like curly-bracket syntax. Most Java code is also syntactically valid Groovy. The main features of the language are the following:

- The code is compact and clean. The syntax is very similar to Java but Groovy adds some "syntactic sugar" to make things more readable and expressive. Moreover the syntax is by far less pedantic than Java, giving the developers the opportunity to write much less code and to be generally more productive.
- The language is both statically and dynamically typed. A lot of typical characteristics of a scripting language are included in Groovy: native syntax for arrays, dictionaries and regular expressions, expressions embedded inside strings, safe navigational operator, simple and coherent manipulation of natural hierarchical structures (like HTML file, XML file, Swing UI, etc.), operator overloading, and several paradigms coming from functional world like closures, curry functions and lazy evaluation.
- Groovy offers through GroovyBeans technology a faster and cleaner way to add JavaBeans support to classes. The language automatically creates the necessary accessor and mutator methods corresponding to a declared property in a Groovy class.
- Groovy offers great support for metaprogramming and DSLs through the use of ExpandoMetaClass, Extension Modules (only in Groovy 2), Categories and DelegatingMetaClass.

- Groovy uses the powerful Java standard library but adds to to the classes of the JDK a lot of additional helper methods, which simplify a lot the life of programmers. For example, a Groovy program that print and numbers each line of a textual file is as simple as:

```
def cont = 1
new File("/path/to/filename").eachLine{
  println "${cont++}. ${it}"
}
```

- Being a language for JVM, Groovy is 100% compatible with code written in Java and in general with all languages that target the JVM platform. This essentially means that you can freely mix code coming from both languages in a transparent way, giving the developers all the freedom and the flexibility to choose the proper language for the task to be resolved.

For comparison with Java, here it is the same "HelloWorld" programs as before now written in Groovy:

Listing 2: A revised classic "HelloWorld" in Groovy

```
class Hello {
  String name = "Noname"
  def sayHello() {
    println "Hello world, ${getName()}!"
  }
}

def h = new Hello() {name = "Tiziano"}
h.sayHello()
```

3 How Maven works: main concepts

Maven handles projects through the definition of a proper Project Object Model (usually the file "pom.xml"), a configuration file which describes exactly what to do when building or deploying a software. Maven can be invoked through the command `mvn` from the main folder of project (the folder containing the POM file, pom.xml).

Maven is strongly based on "convention over configuration" [4], a software design paradigm which seeks to decrease the number of decisions that developers need to make, gaining simplicity, but not necessarily losing flexibility. This property allows the developers to write a minimal number of directives in POM file and Maven will still be able to manage adequately your project. This is possible because Maven, for each directive not specified by developers, will assume reasonable default values. Here you can see the minimum POM file that you can write:

Listing 3: The most simple Maven POM file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>project_name</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Excluding information contained in the tags "project" and "modelVersion" which describe the specific version of POM to use, it is sufficient that you specify the "domain" name of your project (the tag "groupId"), the specific name of project (the tag "artifactId") and the current version of project (the tag "version"). Even with this minimal set of information, Maven is able to build a Jar distribution of your classes by performing correctly tasks like cleaning, compilation, packaging and installation. For example, cleaning and compiling the source code of a project can be easily done by issuing the command `mvn clean compile` from a command prompt.

3.1 The dependency management

Maven is able to manage the dependencies between projects. You can easily specify a dependency on your project by another project (in Maven terms called "artifacts") by simply declaring it in the POM file. For example, you can declare the dependency of your software from version 1.2.17 of artifact *log4j* in this way:

Listing 4: Declaring a dependency from a specific artifact.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...

  <dependencies>
    ...
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
    ...
  </dependencies>
</project>
```

The dependencies management works also for your project. This is why it is mandatory to declare in the project POM the triple <groupID, artifactID, version> which uniquely identifies your artifact among all the other declared artifacts. In that way, every external project which depends on your project's artifact can easily use it.
An open question is where Maven can find all available artifacts (such which can be referenced through the dependency mechanism). Maven can make use of the so called software repositories. They are repositories containing all artifacts and related metadata and are organized in such way that it is easy for the tool to discover and retrieve the requested information. The repository can be local (local

to the machine) or remote (located on a remote machine.) The local repository works like a cache for most referenced artifacts by all local projects and as a private repository for artifacts generated by private local projects. Maven will use one local repository for each user on a specific machine (normally located under `.m2` folder in the user home directory). The remote repository generally can have public or limited access by the users and can be targeted toward a specific type of software (e.g. public repository for open source software like Maven Central Repository[1], restricted repository for company specific software, etc.). Generally Maven can use an unlimited number of remote repositories in a given project.
Maven resolves the requested dependency in a project by trying to find the requested artifact first in the local user repository and then, if it can not satisfy it, in the list of configured remote repositories. If the found artifact is located remotely, Maven downloads a copy of the artifact on the local repository, and then it makes the artifact available to the build process of the project. Maven will reserve a special treatment to artifacts declared with tag `<version>` containing the suffix `SNAPSHOT`. They are "work-in-progress" code. Maven breaks components into "releases" and "snapshots". When it stores release artifacts on a software repository, it makes use only of the triple <groupID, artifactID, version> to uniquely identify a specific version of the software. When it stores snapshots of a software, Maven uses the triple <groupID, artifactID, version> plus build time to uniquely identify a software version. In this way, successive snapshots build of a project with the same version value (e.g. 1.0-SNAPSHOT) generate different build versions and Maven will correctly use the last generated version of the software when required.

3.2 The build lifecycle

Maven is based around the central concept of a *build lifecycle*. What this means is that the process for building and distributing a particular artifact is clearly defined. There are 3 main built-in build lifecycle defined:

- **clean**: it handles the cleaning process in a project.
- **default**: it handles the building, testing and deployment of artifacts in a project.
- **site**: it handles the documentation process of a project.

Every build lifecycle is composed by an ordered set of *phases* which essentially specifies the logical steps to be performed in a specific build lifecycle. As an example, the *default* build lifecycle is composed by the following ordered phases (as reported by original Maven documentation):

[1] This is the default remote repository configured for a new project and it contains all the more popular Java libraries released under open source license.

validate Validate the project is correct and all necessary information is available.

compile Compile the source code of the project.

test Test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.

package Take the compiled code and package it in its distributable format, such as a JAR.

integration-test Process and deploy the package if necessary into an environment where integration tests can be run.

verify Run any checks to verify the package is valid and meets quality criteria.

install Install the package into the local repository, for use as a dependency in other projects locally.

deploy Done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

You can invoke directly some specific phase and Maven will ensure that before executing the requested phase it will have already executed the previous phases declared in the build lifecycle. So giving the command `mvn package`, Maven will execute the goals `validate` $\longrightarrow$ `compile` $\longrightarrow$ `test` $\longrightarrow$ `package` in the specified order.

Every phase is described by using an ordered set of *goals* which specify in a more granular way how to realize of the considered phase. Every goal can generally be linked to zero, one or more phases. It depends on the genericity of considered goal. If the goal is very generic it could be used for very different purposes and therefore in several different phases. If a goal is not linked to any specific phase, it can be invoked directly through Maven like any other phase.

Maven is a generic project management tool that is not tied to a specific type of software project, nor to a specific language such as Java. By default, it can be used to generate Jar libraries, Web applications (.war archives), enterprise applications (.ear archives), and, thanks to its plugin architecture, extended by developers to be able to generate a lot of other types of custom artifacts.
Maven has two ways to associate the specific goals to phases of a certain build lifecycle. The first way is to specify in the POM file the type of artifact that you want to build through the tag `<packaging>`. The default possible values are `jar`, `war`, `ear` and `pom`. Specify a particular value for this tag implies associate a specific mapping of goals with the set of available phases. For example, to build a Jar library you must declare:

Listing 5: Declaring the type of main artifact to build.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <packaging>jar</packaging>
  ...
</project>
```

The second way is through the use of 3rd-party plugins which provide new goals for the project. You can specify in the POM file to use some goals of a certain plugin. The plugin knows the phase it must link to, so it can customize the build process. For example, to generate an additional Jar artifact containing the source code of the project, you can add:

Listing 6: Customize build process through the use of plugins.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
    <build>
      <plugins>
        <!-- Used to pack the source code -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.2.1</version>
        <executions>
          <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
    </build>
  ...
</project>
```

Here we are instructing Maven to execute the goal *jar* of plugin identified by triple *<org.apache.maven.plugins, maven-source-plugin, 2.2.1>* in the *verify* phase of *default* build lifecycle.

3.3 The default source structure of the project

Until now, we have seen how the Maven build process is structured and works. In Listing 3, we showed the minimal POM file that you can write and which can be used successfully by Maven to do a lot of things. To be able to perform correctly the build process, Maven assumes that your project is structured in a certain way. In particular, it targets by default to a Java project (later we

will see how to extend this also to Groovy files). The default Maven project directory is as follows:

`src/main/java`	Application/Library sources
`src/main/resources`	Application/Library resources
`src/main/filters`	Resource filter files
`src/main/assembly`	Assembly descriptors
`src/main/config`	Configuration files
`src/main/scripts`	Application/Library scripts
`src/main/webapp`	Web application sources
`src/test/java`	Test sources
`src/test/resources`	Test resources
`src/test/filters`	Test resource filter files
`src/site`	Site
`target/`	The output build directory
`LICENSE.txt`	Project's license
`NOTICE.txt`	Notices and attributions required by libraries that the project depends on
`README.txt`	Project's readme
`pom.xml`	The Project Object Model file

The `src` folder contains all the source material for building the project. In particular, the Java source files must be in subfolder `src/main/java` and the application/library resource files must be stored in subfolder `src/main/resources`. Note that Maven separates the source folders of the application/library (all that is inside the folder `src/main/`) from the source folder of test code for the application/library (all that is inside the folder `src/test`).
If you follow the previous structure for your project, it is not necessary to specify anything in your POM about where to find the sources (this all the magic of "convention over configuration"). You can obviously also put your source files in other directories but in that case you must indicate to Maven through the POM the new folder containing the source code. To add new source folders, simply add this to your POM file specifying the folder to add (in this example `src/main/another_folder`):

Listing 7: How to add several source directories to your project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
  <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>build-helper-maven-plugin</artifactId>
          <executions>
              <execution>
                  <phase>generate-sources</phase>
                  <goals><goal>add-source</goal></goals>
```

```
                    <configuration>
                        <sources>
                            <source>src/main/another_folder
                            </source>
                        </sources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
      </plugins>
  </build>
  ...
</project>
```

3.4 Integration of Groovy into Maven build process

In this section we'll see how to integrate the compilation phase of Groovy files of the project into global Maven build process. To process correctly Groovy files, the POM file used by Maven need to be extended by a custom plugin which adds the feature "compile Groovy files" to the tool. This can be done by customizing the POM in this way:

Listing 8: How to add several source directories to your project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
  <build>
      <plugins>
      <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <!-- 2.8.0-01 and later require
          maven-compiler-plugin 3.0 or higher -->
          <version>3.0</version>
          <configuration>
              <compilerId>groovy-eclipse-compiler</compilerId>
              <!-- set verbose to be true if you
              want lots of uninteresting messages -->
              <!-- <verbose>true</verbose> -->
          </configuration>
          <dependencies>
              <dependency>
                <groupId>org.codehaus.groovy</groupId>
                <artifactId>groovy-eclipse-compiler</artifactId>
                <version>2.8.0-01</version>
              </dependency>

              <!-- for 2.8.0-01 and later you must
              have an explicit dependency on
              groovy-eclipse-batch -->
            <dependency>
                <groupId>org.codehaus.groovy</groupId>
```

```
                <artifactId>groovy-eclipse-batch</artifactId>
                <version>2.1.5-03</version>

                <!-- or choose a different compiler version -->
                <!-- <version>1.8.6-01</version> -->
                <!-- <version>1.7.10-06</version> -->
            </dependency>
          </dependencies>
        </plugin>
      </plugins>
    </build>
    ...

    <dependencies>
      ...
      <dependency>
        <groupId>org.codehaus.groovy</groupId>
        <artifactId>groovy-all</artifactId>
        <version>2.1.5</version>
      </dependency>
      ...
    </dependencies>
</project>
```

We extend the behaviour of standard component *maven-compiler-plugin* to use the *groovy-eclipse-compiler* plugin when one of its standard goals is called. The groovy-eclipse-compiler manages the task of compile the Groovy source files. Coherently with Java, it searches for Groovy files in specific directories: `src/main/groovy` for source code of the application/library, `src/test/groovy` for test source code of the application/library. In order for this process to work properly, you must also include in the section *dependencies* of your POM the dependency from Groovy artifact (in this case the 2.1.5, the current last available version of the language runtime).

4 Useful customizations of the Project Object Model

4.1 Using multiple software repositories

As discussed in section 3.1, Maven can use multiple software repositories to resolve dependencies. Let's see how to change the POM to make use of several software repositories:

Listing 9: Using multiple software repositories.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
  <repositories>
    <repository>
      <id>my-internal-site-1</id>
```

```
      <url>http://myserver1/repo</url>
    </repository>
    <repository>
      <id>my-internal-site-2</id>
      <url>http://myserver2/repo</url>
    </repository>
  </repositories>
  ...
</project>
```

After being declared, Maven will access the repositories in the specified order when need to resolve an artifact. Sometimes it is necessary to access to a restricted repository (for example because you need to access to a proprietary artifact). In that case, you can provide the required credentials inside the Maven settings file (the file `<home_user_directory>/.m2/settings.xml`) as follows:

Listing 10: How to specify credentials to a specific repository

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>my-internal-site-1</id>
      <username>my_login</username>
      <password>my_password</password>
    </server>
  </servers>
  ...
</settings>
```

In this case, we are assuming that the server `http://myserver1/repo` has needs of the declared credentials (user: my_login and password: my_password) to allow access to artifacts. Note that the link among the information declared in both files is made through the `id` tag (the value must be the same in both files).

4.2 Using multiple profiles

Another useful thing you can do in a project is to define different profiles for specific contexts. The classic example is when you exploit this feature to generate two profiles, one for snapshot builds and the other for release builds. Inside each build profile, you can define where deploy the generated artifacts and redefine the value of certain properties (like the name of the artifact to be generated) or create new properties specific for the considered profile. Such scenario can be made by declaring:

Listing 11: Using multiple software repositories.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...

  <profiles>
    <profile>
      <id>devel</id>
      <!-- Devel profile is active by default -->
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <artifact.name>artifact-devel</artifact.name>
      </properties>

      <distributionManagement>
        <repository>
          <id>my-internal-site-1</id>
          <url>
            http://http://myserver1/repositories/snapshots
          </url>
        </repository>
      </distributionManagement>
    </profile>
    <profile>
      <id>release</id>
      <properties>
        <artifact.name>artifact-release</artifact.name>
      </properties>
      <distributionManagement>
        <repository>
          <id>my-internal-site-1</id>
          <url>
            http://myserver1/repositories/releases
          </url>
        </repository>
      </distributionManagement>
    </profile>
  </profiles>
  ...
</project>
```

Here we define the profiles *devel* and *release*. On each profile we redefine the variable *artifact.name* to be coherent with the specific profile we are considering. More importantly, each profile defines the repository where an artifact built with that profile should be deployed. For example, in profile *devel*, activated by default if not differently specified, we assume that the artifact will be deployed on url `http://http://myserver1/repositories/snapshots`. Eventually, if the server has restricted access, we can specify the corresponding credentials on the Maven settings file as explained in section 4.1.

4.3 Building a bundle of main artifact

Sometimes it is useful to generate an artifact containing the union of several artifacts used in the project. For example, assume that the main artifact of a project should be a library Jar file. By default, Maven will build an artifact containing all your compiled code but it will exclude from the artifact generated all the code coming from artifacts from which your code depends on. This default setting is reasonable because Maven thinks that you want to distribute your software (the Jar library) through the Maven's software repositories and so easily included as dependency in other Maven projects. Sometimes we simply want to generate a secondary artifact containing all compiled code (called *bundle*, it will contain your own code plus code of external dependencies) because you want to be able to immediate use the artifact (e.g. call a Java/Groovy program located in the artifact).
To generate a bundle of main artifact, you should update your POM file in this way:

Listing 12: Generating a bundle of our software.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...

    <build>
      <plugins>
        <!-- Used to build a bundle containing the
        code of software and all
        its library dependencies -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <shadedArtifactAttached>true</shadedArtifactAttached>
          <finalName>
            software_name-bundle
          </finalName>
          <createDependencyReducedPom>
            false
          </createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      </plugins>
    </build>
```

```
  ...
</project>
```

In the example, the bundle name will have the prefix *software_name-bundle*.

4.4 Generating a secondary artifact containing the source code

Sometimes we want to make available the source code of our software as a Jar artifact (e.g. useful for debugging purposes when our library code is used by other software). To do this, you should update your POM file as follows:

Listing 13: Generating a secondary artifact containing the source code of our software.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...

    <build>
      <plugins>
        ...
        <!-- Used to pack the source code -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.2.1</version>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      ...
      </plugins>
    </build>

  ...
</project>
```

5 A complete template Maven project using Java/-Groovy code

5.1 The POM of the template project

Following we show the complete template of Maven's POM which can be used to maintain projects based on code written in the programming languages Java and Groovy:

Listing 14: The complete Maven POM template.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.company</groupId>
  <artifactId>software_name</artifactId>

  <!-- ********** CHANGE THESE PROPERTIES WHEN UPDATE
    TO A NEW SOFTWARE VERSION. IMPORTANT: IN DEVEL LEAVE
    "-SNAPSHOT" AT THE END OF VERSION. MAVEN WILL THREAT
    IT DIFFERENTLY! ***** -->
  <version>0.1.0</version>


  <packaging>jar</packaging>

  <name>software_name</name>
  <url>http://www.software_name.com</url>

  <properties>
    <project.build.sourceEncoding>
      UTF-8
    </project.build.sourceEncoding>
    <project.reporting.outputEncoding>
      UTF-8
    </project.reporting.outputEncoding>
  </properties>

  <!-- Put all your software repositories. -->
  <repositories>
    <repository>
      <id>my-internal-site-1</id>
      <url>http://myserver1/repo</url>
    </repository>
    <!--
    <repository>
      <id>my-internal-site-2</id>
      <url>http://myserver2/repo</url>
    </repository>
    -->
  </repositories>
```

```
<!-- Profiles configuration -->
<!-- Specify to Maven which profile to use when
   building the library. If you want to build
   a devel snapshot specify 'mvn package -P devel',
   if you want to build a release specify
   'mvn package -P release'. If you not specify the
  -P parameter, the profile devel will be used. -->
<profiles>
  <profile>
    <id>devel</id>
    <!-- Devel profile is active by default -->
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <software.build.generateversion>
        ${project.version}
      </software.build.generateversion>
      <software.build.finalname>
        ${project.name}-dev
      </software.build.finalname>
    </properties>

    <distributionManagement>
      <repository>
        <id>my-internal-site-1</id>
        <url>http://myserver1/repositories/snapshots</url>
      </repository>
    </distributionManagement>
  </profile>
  <profile>
    <id>release</id>
    <properties>
      <software.build.generateversion>
        ${project.version}
      </software.build.generateversion>
      <software.build.finalname>
        ${project.name}
      </software.build.finalname>
    </properties>
    <distributionManagement>
      <repository>
        <id>my-internal-site-1</id>
        <url>http://myserver1/repositories/releases</url>
      </repository>
    </distributionManagement>
  </profile>
</profiles>

<build>
  <finalName>${software.build.finalname}</finalName>
```

```
<plugins>
  <!-- Used to build a bundle containing the code
  of the software and all its dependencies -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.0</version>
    <configuration>
      <shadedArtifactAttached>
        true
      </shadedArtifactAttached>
      <finalName>
        ${software.build.finalname}-bundle
      </finalName>
      <createDependencyReducedPom>
        false
      </createDependencyReducedPom>
    </configuration>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>shade</goal>
        </goals>
      </execution>
    </executions>
  </plugin>

  <!-- Used to pack the source code -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
      <execution>
        <id>attach-sources</id>
        <goals>
          <goal>jar</goal>
        </goals>
      </execution>
    </executions>
  </plugin>

  <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <!-- 2.8.0-01 and later require
      maven-compiler-plugin 3.0 or higher -->
      <version>3.0</version>
      <configuration>
          <compilerId>
            groovy-eclipse-compiler
          </compilerId>
      <source>1.6</source>
```

```
            <target>1.6</target>
                <!-- set verbose to be true if you
                want lots of uninteresting messages -->
                <!-- <verbose>true</verbose> -->
            </configuration>
            <dependencies>
                <dependency>
                  <groupId>org.codehaus.groovy</groupId>
                  <artifactId>
                    groovy-eclipse-compiler
                  </artifactId>
                  <version>2.8.0-01</version>
                </dependency>

                <!-- for 2.8.0-01 and later you must
                have an explicit dependency on
                groovy-eclipse-batch -->
              <dependency>
                  <groupId>org.codehaus.groovy</groupId>
                  <artifactId>
                    groovy-eclipse-batch
                  </artifactId>
                  <version>2.1.5-03</version>

                  <!-- or choose a different compiler
                  version -->
                  <!-- <version>1.8.6-01</version> -->
                  <!-- <version>1.7.10-06</version> -->
            </dependency>
          </dependencies>
        </plugin>

    </plugins>
  </build>


  <dependencies>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
      <version>2.1.5</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

The above template includes all the facilities discussed in the previous sections. Moreover note that in the dependencies set we have already included also JUnit library[2] but with a scope of *test*. This essentially means that Maven will use the library only when testing the code but not when building the artifacts (data and metadata)to distribute externally.

5.2 The Maven commands available by using the provided template

This is the list of main Maven commands which can be used by using the proposed POM template:

`mvn clean`	Used to clean all the compiled code and generated data.
`mvn compile`	Used to compile Java and Groovy source code.
`mvn test`	Execute all tests contained in `src/test/` directory.
`mvn package`	Used to package the compiled code into a proper archive file (Jar for libraries, War for Web applications, etc.) and to generate all secondary artifacts like bundle artifact and source artifact.
`mvn install`	Used to install the artifacts generated in `package` phase into local Maven repository.
`mvn deploy`	Deploy all the generated artifacts to configured repository.
`mvn site`	Generate a web site containing information about the project.

Remember that by default (as specified in the POM file) the above Maven commands are run in the context of *devel* profile. To switch to another profile, you must prefix the commands with `-P <profile_name>` where `profile_name` can be either *devel* or *release*. For example, to package all code into *release* mode, just execute the command `mvn -P release package`.

References

[1] Apache. Ant. `http://ant.apache.org/`.

[2] Apache. Maven. `http://maven.apache.org/`.

[3] Tiobe Software BV. Tiobe programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[4] The software paradigm "convention over configuration". `http://en.wikipedia.org/wiki/Convention_over_configuration`.

[5] DedaSys LLC. Programming language popularity. `http://www.langpop.com/`.

[6] Oracle. Java. `http://www.java.com/`.

[2] This is the de-facto standard Java library used to easily write unit testings for the code.