6

Assessing Project Health with Maven

This chapter covers:

- How Maven relates to project health
- Organizing documentation and developer reports
- Selecting the right tools to monitor the health of your project
- How to incorporate reporting tools
- Tips for how to use tools more effectively

Life is not an exact science, it is an art.

- Samuel Butler

6.1. What Does Maven Have to do With Project Health?

In the introduction, it was pointed out that Maven's application of patterns provides visibility and comprehensibility. It is these characteristics that assist you in assessing the health of your project.

Through the POM, Maven has access to the information that makes up a project, and using a variety of tools, Maven can analyze, relate, and display that information in a single place. Because the POM is a *declarative* model of the project, new tools that can assess its health are easily integrated. In this chapter, you'll learn how to use a number of these tools effectively.

When referring to health, there are two aspects to consider:

- **Code quality** - determining how well the code works, how well it is tested, and how well it adapts to change.
- **Project vitality** - finding out whether there is any activity on the project, and what the nature of that activity is.

Maven takes all of the information you need to know about your project and brings it together under the project Web site. The next three sections demonstrate how to set up an effective project Web site.

It is important not to get carried away with setting up a fancy Web site full of reports that nobody will ever use (especially when reports contain failures they don't want to know about!). For this reason, many of the reports illustrated can be run as part of the regular build in the form of a “check” that will fail the build if a certain condition is not met.

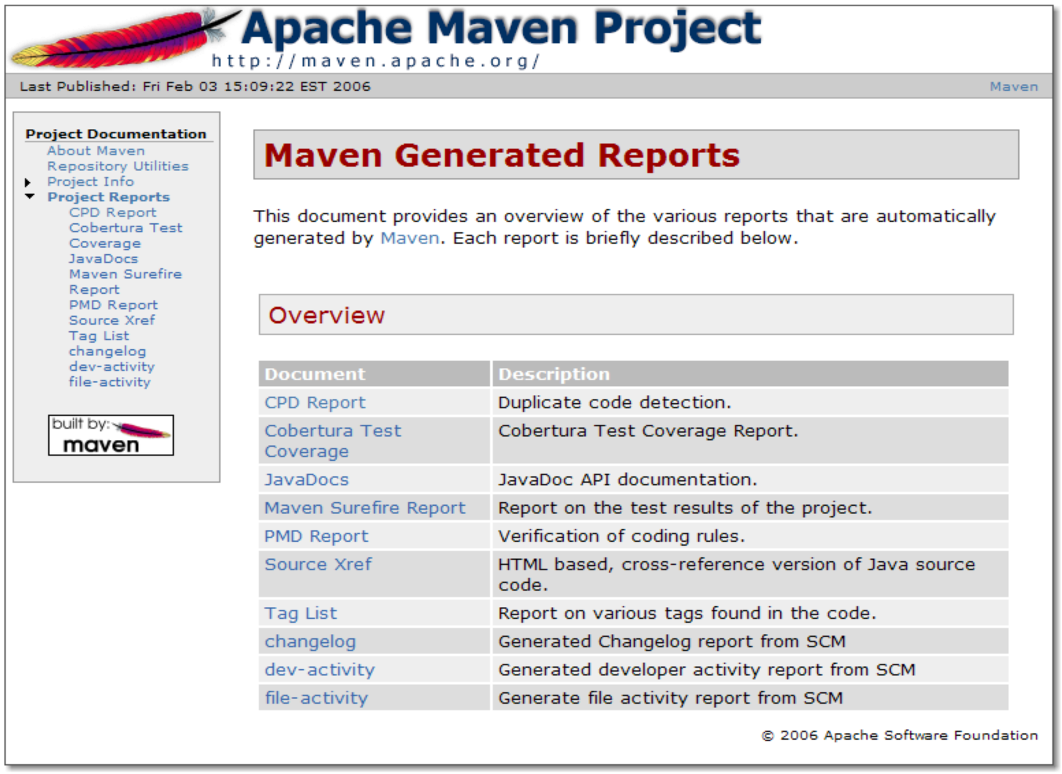
But, why have a site, if the build fails its checks? The Web site also provides a permanent record of a project's health, which everyone can see at any time. It provides additional information to help determine the reasons for a failed build, and whether the conditions for the checks are set correctly. This is important, because if the bar is set too high, there will be too many failed builds. This is unproductive as minor changes are prioritized over more important tasks, to get a build to pass. Conversely, if the bar is set too low, the project will meet only the lowest standard and go no further.

In this chapter, you will be revisiting the Proficio application that was developed in Chapter 3, and learning more about the health of the project. The code that concluded Chapter 3 is also included in `Code_Ch06-1.zip` for convenience as a starting point. To begin, unzip the `Code_Ch06-1.zip` file into `C:\mvnbook` or your selected working directory, and then run `mvn install` from the `proficio` subdirectory to ensure everything is in place.

6.2. Adding Reports to the Project Web site

This section builds on the information on project Web sites in Chapter 2 and Chapter 3, and now shows how to integrate project health information.

To start, review the project Web site shown in figure 6-1.



Maven Generated Reports

This document provides an overview of the various reports that are automatically generated by Maven. Each report is briefly described below.

Overview

Document	Description
CPD Report	Duplicate code detection.
Cobertura Test Coverage	Cobertura Test Coverage Report.
JavaDocs	JavaDoc API documentation.
Maven Surefire Report	Report on the test results of the project.
PMD Report	Verification of coding rules.
Source Xref	HTML based, cross-reference version of Java source code.
Tag List	Report on various tags found in the code.
changelog	Generated Changelog report from SCM
dev-activity	Generated developer activity report from SCM
file-activity	Generate file activity report from SCM

© 2006 Apache Software Foundation

Figure 6-1: The reports generated by Maven

You can see that the navigation on the left contains a number of reports. The *Project Info* menu lists the standard reports Maven includes with your site by default, unless you choose to disable them. These reports are useful for sharing information with others, and to reference as links in your mailing lists, SCM, issue tracker, and so on. For newcomers to the project, having these standard reports means that those familiar with Maven Web sites will always know where to find the information they need.

The second menu (shown opened in figure 6-1), *Project Reports*, is the focus of the rest of this chapter. These reports provide a variety of insights into the quality and vitality of the project.

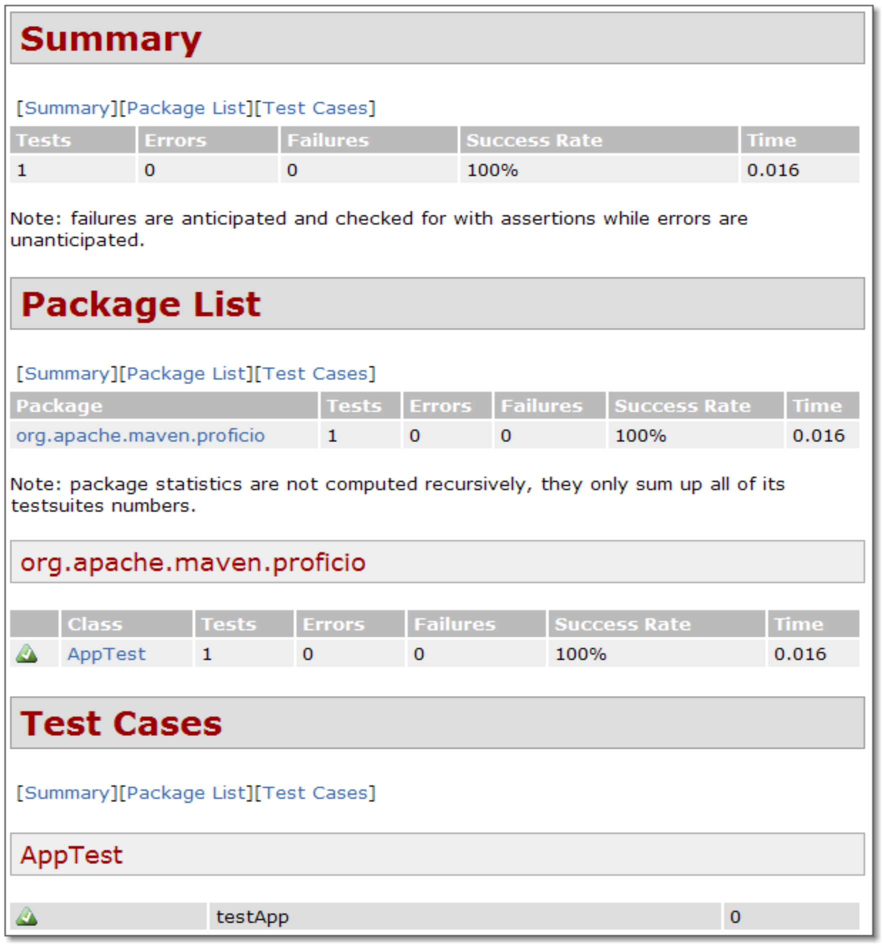
On a new project, this menu doesn't appear as there are no reports included. However, adding a new report is easy. For example, you can add the Surefire report to the sample application, by including the following section in `proficio/pom.xml`:

```
  ...
  <reporting>
   <plugins>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-surefire-report-plugin</artifactId>
    </plugin>
   </plugins>
  </reporting>
  ...
</project>
```

This adds the report to the top level project, and as a result, it will be inherited by all of the child modules. You can now run the following site task in the `proficio-core` directory to regenerate the site.

```
C:\mvnbook\proficio\proficio-core> mvn site
```

This can now be found in the file `target/site/surefire-report.html`, and is shown in figure 6-2.



Summary

[Summary][Package List][Test Cases]

Tests	Errors	Failures	Success Rate	Time
1	0	0	100%	0.016

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Success Rate	Time
org.apache.maven.proficio	1	0	0	100%	0.016

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

org.apache.maven.proficio

	Class	Tests	Errors	Failures	Success Rate	Time
✓	AppTest	1	0	0	100%	0.016

Test Cases

[Summary][Package List][Test Cases]

AppTest

✓	testApp	0

Figure 6-2: The Surefire report

As you may have noticed in the summary, the report shows the test results of the project.

For a quicker turn around, the report can also be run individually using the following standalone goal:

```
C:\mvnbook\proficio\proficio-core> mvn surefire-report:report
```

That's all there is to generating the report! This is possible thanks to key concepts of Maven discussed in Chapter 2: through a **declarative project model**, Maven knows where the tests and test results are, and due to using **convention over configuration**, the defaults are sufficient to get started with a useful report.

6.3. Configuration of Reports

Before stepping any further into using the project Web site, it is important to understand how the report configuration is handled in Maven.

You might recall from Chapter 2 that a plugin is configured using the configuration element inside the plugin declaration in `pom.xml`, for example:

```
...
<build>
 <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>
...
```

Configuration for a reporting plugin is very similar, however it is added to the reporting section of the POM. For example, the report can be modified to only show test failures by adding the following configuration in `pom.xml`:

```
...
<reporting>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin</artifactId>
   <configuration>
    <showSuccess>false</showSuccess>
   </configuration>
  </plugin>
 </plugins>
</reporting>
...
```

The addition of the plugin element triggers the inclusion of the report in the Web site, as seen in the previous section, while the configuration can be used to modify its appearance or behavior.

If a plugin contains multiple reports, they will all be included.

However, some reports apply to both the site, and the build. To continue with the Surefire report, consider if you wanted to create a copy of the HTML report in the directory `target/surefire-reports` every time the build ran. To do this, the plugin would need to be configured in the build section instead of, or in addition to, the `reporting` section:

```
...
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin</artifactId>
   <configuration>
    <outputDirectory>
     ${project.build.directory}/surefire-reports
    </outputDirectory>
   </configuration>
   <executions>
    <execution>
     <phase>test</phase>
     <goals>
      <goal>report</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
...
```

“Executions” such as this were introduced in Chapter 3. The plugin is included in the build section to ensure that the configuration, even though it is not specific to the execution, is used only during the build, and not site generation.

Plugins and their associated configuration that are declared in the build section are not used during site generation.

However, what if the location of the Surefire XML reports that are used as input (and would be configured using the `reportsDirectory` parameter) were different to the default location? Initially, you might think that you'd need to configure the parameter in both sections. Fortunately, this isn't the case – adding the configuration to the reporting section is sufficient.

Any plugin configuration declared in the reporting section is also applied to those declared in the build section.

When you configure a reporting plugin, always place the configuration in the reporting section – unless one of the following is true:

1. The reports will not be included in the site
2. The configuration value is specific to the build stage

When you are configuring the plugins to be used in the reporting section, by default all reports available in the plugin are executed once. However, there are cases where only some of the reports that the plugin produces will be required, and cases where a particular report will be run more than once, each time with a different configuration.

Both of these cases can be achieved with the `reportSets` element, which is the reporting equivalent of the executions element in the build section. Each report set can contain configuration, and a list of reports to include. For example, consider if you had run Surefire twice in your build, once for unit tests and once for a set of performance tests, and that you had had generated its XML results to target/surefire-reports/unit and `target/surefire-reports/perf` respectively.

To generate two HTML reports for these results, you would include the following section in your `pom.xml`:

```
...
<reporting>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin</artifactId>
   <reportSets>
    <reportSet>
     <id>unit</id>
     <configuration>
      <reportsDirectory>
       ${project.build.directory}/surefire-reports/unit
      </reportsDirectory>
      <outputName>surefire-report-unit</outputName>
     </configuration>
     <reports>
      <report>report</report>
     </reports>
    </reportSet>
    <reportSet>
     <id>perf</id>
     <configuration>
      <reportsDirectory>
       ${project.build.directory}/surefire-reports/perf
      </reportsDirectory>
      <outputName>surefire-report-perf</outputName>
     </configuration>
     <reports>
      <report>report</report>
     </reports>
    </reportSet>
   </reportSets>
  </plugin>
 </plugins>
</reporting>
...
```

Running `mvn site` with this addition will generate two Surefire reports: `target/site/surefire-report-unit.html` and `target/site/surefire-report-perf.html`.

However, as with executions, running `mvn surefire-report:report` will not use either of these configurations. When a report is executed individually, Maven will use only the configuration that is specified in the plugin element itself, outside of any report sets.

The reports element in the report set is a required element. If you want all of the reports in a plugin to be generated, they must be enumerated in this list. The reports in this list are identified by the goal names that would be used if they were run from the command line.

It is also possible to include only a subset of the reports in a plugin. For example, to generate only the mailing list and license pages of the standard reports, add the following to the reporting section of the `pom.xml` file:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-project-info-reports-plugin</artifactId>
 <reportSets>
  <reportSet>
   <reports>
    <report>mailing-list</report>
    <report>license</report>
   </reports>
  </reportSet>
 </reportSets>
</plugin>
...
```

While the defaults are usually sufficient, this customization will allow you to configure reports in a way that is just as flexible as your build.

6.4. Separating Developer Reports From User Documentation

After adding a report, there's something subtly wrong with the project Web site. On the entrance page there are usage instructions for Proficio, which are targeted at an end user, but in the navigation there are reports about the health of the project, which are targeted at the developers.

This may be confusing for the first time visitor, who isn't interested in the state of the source code, and an inconvenience to the developer who doesn't want to wade through end user documentation to find out the current state of a project's test coverage.

This approach to balancing these competing requirements will vary, depending on the project. Consider the following:

- The commercial product, where the end user documentation is on a completely different server than the developer information, and most likely doesn't use Maven to generate it;
- The open source graphical application, where the developer information is available, but quite separate to the end user documentation;
- The open source reusable library, where much of the source code and Javadoc reference is of interest to the end user.

To determine the correct balance, each section of the site needs to be considered; in some cases down to individual reports. Table 6-1 lists the content that a project Web site may contain, and the content's characteristics.

Table 6-1: Project Web site content types

Content	Description	Updated	Distributed	Separated
News, FAQs and general Web site	This is the content that is considered part of the Web site rather than part of the documentation.	Yes	No	Yes
End user documentation	This is documentation for the end user including usage instructions and guides. It refers to a particular version of the software.	Yes	Yes	No
Source code reference material	This is reference material (for example, Javadoc) that in a library or framework is useful to the end user, but usually not distributed or displayed in an application.	No	Yes	No
Project health and vitality reports	These are the reports discussed in this chapter that display the current state of the project to the developers.	Yes	No	No

In the table, the **Updated** column indicates whether the content is regularly updated, regardless of releases. This is true of the news and FAQs, which are based on time and the current state of the project. Some standard reports, like mailing list information and the location of the issue tracker and SCM are updated also. It is also true of the project quality and vitality reports, which are continuously published and not generally of interest for a particular release. However, source code references should be given a version and remain unchanged after being released.

The situation is different for end user documentation. It is good to update the documentation on the Web site between releases, and to maintain only one set of documentation.

Features that are available only in more recent releases should be marked to say when they were introduced. It is important not to include documentation for features that don't exist in the last release, as it is confusing for those reading the site who expect it to reflect the latest release.

The best compromise between not updating between releases, and not introducing incorrect documentation, is to branch the end user documentation in the same way as source code. You can maintain a stable branch, that can be updated between releases without risk of including new features, and a development branch where new features can be documented for when that version is released.

The **Distributed** column in the table indicates whether that form of documentation is typically distributed with the project. This is typically true for the end user documentation. For libraries and frameworks, the Javadoc and other reference material are usually distributed for reference as well. Sometimes these are included in the main bundle, and sometimes they are available for download separately.

The **Separated** column indicates whether the documentation can be a separate module or project. While there are some exceptions, the source code reference material and reports are usually generated from the modules that hold the source code and perform the build. For a single module library, including the end user documentation in the normal build is reasonable as it is closely tied to the source code reference.

However, in most cases, the documentation and Web site should be kept in a separate module dedicated to generating a site. This avoids including inappropriate report information and navigation elements.

This separated documentation may be a module of the main project, or maybe totally independent. You would make it a module when you wanted to distribute it with the rest of the project, but make it an independent project when it forms the overall site with news and FAQs, and is not distributed with the project.

It is important to note that none of these are restrictions placed on a project by Maven. While these recommendations can help properly link or separate content according to how it will be used, you are free to place content wherever it best suits your project.

In Proficio, the site currently contains end user documentation and a simple report. In the following example, you will learn how to separate the content and add an independent project for the news and information Web site.

The current structure of the project is shown in figure 6-3.

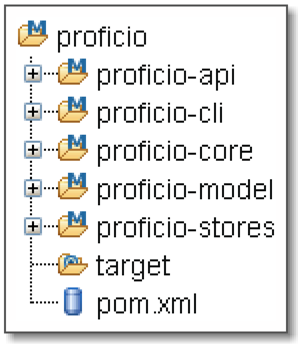


Figure 6-3: The initial setup

The first step is to create a module called user-guide for the end user documentation. In this case, a module is created since it is not related to the source code reference material. This is done using the site archetype :

```
C:\mvnbook\proficio> mvn archetype:create -DartifactId=user-guide \
           -DgroupId=com.mergere.mvnbook.proficio \
           -DarchetypeArtifactId=maven-archetype-site-simple
```

This archetype creates a very basic site in the user-guide subdirectory, which you can later add content to. The resulting structure is shown in figure 6-4.

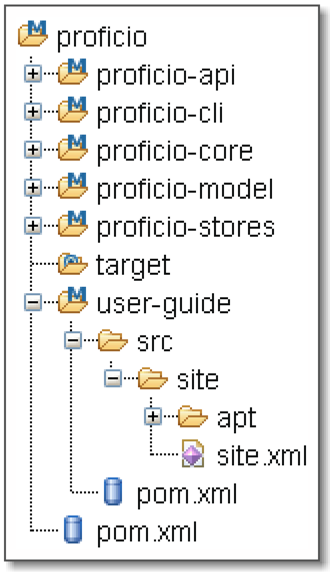


Figure 6-4: The directory layout with a user guide

The next step is to ensure the layout on the Web site is correct. Previously, the URL and deployment location were set to the root of the Web site: http://library.mergere.com/mvnbook/proficio. Under the current structure, the development documentation would go to that location, and the user guide to http://library.mergere.com/mvnbook/proficio/user-guide.

In this example, the development documentation will be moved to a `/reference/`*`version`* subdirectory so that the top level directory is available for a user-facing web site.

Adding the version to the development documentation, while optional, is useful if you are maintaining multiple public versions, whether to maintain history or to maintain a release and a development preview.

First, edit the top level `pom.xml` file to change the site deployment `url`:

```
...
<distributionManagement>
 <site>
  ...
  <url>
   scp://mergere.com/www/library/mvnbook/proficio/reference/${pom.version}
  </url>
 </site>
</distributionManagement>
...
```

Next, edit the `user-guide/pom.xml` file to set the site deployment `url` for the module:

```
...
 <distributionManagement>
  <site>
   <id>mvnbook.site</id>
   <url>
    scp://mergere.com/www/library/mvnbook/proficio/user-guide
   </url>
 </site>
</distributionManagement>
...
```

There are now two sub-sites ready to be deployed:

- http://library.mergere.com/mvnbook/proficio/user-guide/
- http://library.mergere.com/mvnbook/proficio/reference/1.0-SNAPSHOT/

You will not be able to deploy the Web site to the location
`scp://mergere.com/www/library/mvnbook/proficio/user-guide`
It is included here only for illustrative purposes.

Now that the content has moved, a top level site for the project is required. This will include news and FAQs about the project that change regularly.

As before, you can create a new site using the archetype. This time, run it one directory above the `proficio` directory, in the `ch06-1` directory.

```
C:\mvnbook> mvn archetype:create -DartifactId=proficio-site \
      -DgroupId=com.mergere.mvnbook.proficio \
      -DarchetypeArtifactId=maven-archetype-site-simple
```

The resulting structure is shown in figure 6-5.

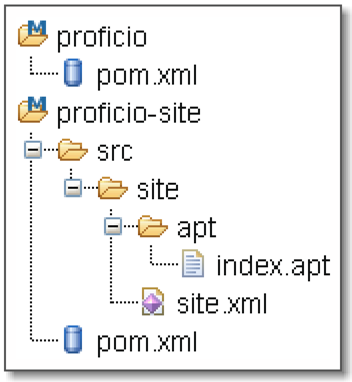


Figure 6-5: The new Web site

You will need to add the same elements to the POM for the `url` and `distributionManagement` as were set originally for `proficio/pom.xml` as follows:

```
...
<url>http://library.mergere.com/mvnbook/proficio</url>
...
<distributionManagement>
 <site>
  <id>mvnbook.website</id>
  <url>scp://mergere.com/www/library/mvnbook/proficio</url>
 </site>
</distributionManagement>
...
```

Next, replace the `src/site/apt/index.apt` file with a more interesting news page, like the following:

```
 -----
Proficio
 -----
Joe Blogs
 -----
2 February 2006
 -----

Proficio

 Proficio is super.

* News

 * <16 Jan 2006> - Proficio project started
```

Finally, add some menus to `src/site/site.xml` that point to the other documentation as follows:

```
...
<menu name="Documentation">
 <item name="User's Guide" href="/user-guide/" />
</menu>

<menu name="Reference">
 <item name="API" href="/reference/1.0-SNAPSHOT/apidocs/" />
 <item name="Developer Info" href="/reference/1.0-SNAPSHOT/" />
</menu>
...
```

You can now run `mvn site` in `proficio-site` to see how the separate site will look. If you deploy both sites to a server using `mvn site-deploy` as you learned in Chapter 3, you will then be able to navigate through the links and see how they relate.

Note that you haven't produced the `apidocs` directory yet, so that link won't work even if the site is deployed. Generating reference documentation is covered in section 6.6 of this chapter.

The rest of this chapter will focus on using the developer section of the site effectively and how to build in related conditions to regularly monitor and improve the quality of your project.

6.5. Choosing Which Reports to Include

Choosing which reports to include, and which checks to perform during the build, is an important decision that will determine the effectiveness of your build reports. Report results and checks performed should be accurate and conclusive – every developer should know what they mean and how to address them.

In some instances, the performance of your build will be affected by this choice. In particular, the reports that utilize unit tests often have to re-run the tests with new parameters. While future versions of Maven will aim to streamline this, it is recommended that these checks be constrained to the continuous integration and release environments if they cause lengthy builds. See Chapter 7, *Team Collaboration with Maven*, for more information.

Table 6-2 covers the reports discussed in this chapter and reasons to use them. For each report, there is also a note about whether it has an associated visual report (for project site inclusion), and an applicable build check (for testing a certain condition and failing the build if it doesn't pass).

You can use this table to determine which reports apply to your project specifically and limit your reading to just those relevant sections of the chapter, or you can walk through all of the examples one by one, and look at the output to determine which reports to use.

While these aren't all the reports available for Maven, the guidelines should help you to determine whether you need to use other reports.

You may notice that many of these tools are Java-centric. While this is certainly the case at present, it is possible in the future that reports for other languages will be available, in addition to the generic reports such as those for dependencies and change tracking.

Table 6-2: Report highlights

Report	Description	Visual	Check	Notes
Javadoc	Produces an API reference from Javadoc.	Yes	N/A	✔ Useful for most Java software ✔ Important for any projects publishing a public API
JXR	Produces a source cross reference for any Java code.	Yes	N/A	✔ Companion to Javadoc that shows the source code ✔ Important to include when using other reports that can refer to it, such as Checkstyle ✗ Doesn't handle JDK 5.0 features
Checkstyle	Checks your source code against a standard descriptor for formatting issues.	Yes	Yes	✔ Use to enforce a standard code style. ✔ Recommended to enhance readability of the code. ✗ Not useful if there are a lot of errors to be fixed – it will be slow and the result unhelpful.

Report	Description	Visual	Check	Notes
PMD	Checks your source code against known rules for code smells.	Yes	Yes	✔ Should be used to improve readability and identify simple and common bugs. ✔ Some overlap with Checkstyle rules
CPD	Part of PMD, checks for duplicate source code blocks that indicates it was copy/pasted.	Yes	No	✔ Can be used to identify lazy copy/pasted code that might be refactored into a shared method. ✔ Avoids issues when one piece of code is fixed/updated and the other forgotten
Tag List	Simple report on outstanding tasks or other markers in source code	Yes	No	✔ Useful for tracking TODO items ✔ Very simple, convenient set up ✔ Can be implemented using Checkstyle rules instead.
Cobertura	Analyze code statement coverage during unit tests or other code execution.	Yes	Yes	✔ Recommended for teams with a focus on tests ✔ Can help identify untested or even unused code. ✗ Doesn't identify all missing or inadequate tests, so additional tools may be required.
Surefire Report	Show the results of unit tests visually.	Yes	Yes	✔ Recommended for easier browsing of results. ✔ Can also show any tests that are long running and slowing the build. ✔ Check already performed by surefire:test.
Dependency Convergence	Examine the state of dependencies in a multiple module build	Yes	No	✔ Recommended for multiple module builds where consistent versions are important. ✔ Can help find snapshots prior to release.
Clirr	Compare two versions of a JAR for binary compatibility	Yes	Yes	✔ Recommended for libraries and frameworks with a public API ✔ Also important for reviewing changes to the internal structure of a project that are still exposed publicly.
Changes	Produce release notes and road maps from issue tracking systems	Yes	N/A	✔ Recommended for all publicly released projects. ✔ Should be used for keeping teams up to date on internal projects also.

6.6. Creating Reference Material

Source code reference materials are usually the first reports configured for a new project, because it is often of interest to the end user of a library or framework, as well as to the developer of the project itself.

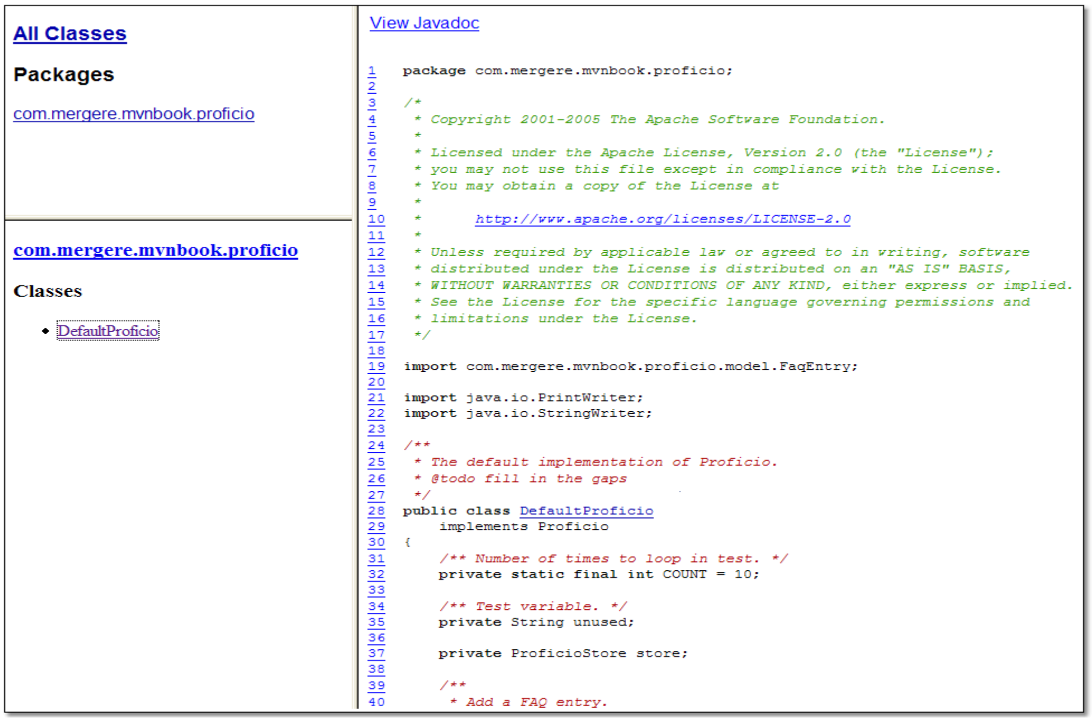
The two reports this section illustrates are:

- JXR – the Java source cross reference, and,
- Javadoc – the Java documentation tool

You can get started with JXR on the example project very quickly, by running the following command:

```
C:\mvnbook\proficio\proficio-core> mvn jxr:jxr
```

You should now see a `target/site/jxr.html` created. This page links to the two sets of cross references: one for the main source tree, and one for the unit tests.



```
package com.mergere.mvnbook.proficio;

/*
 * Copyright 2001-2005 The Apache Software Foundation.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import com.mergere.mvnbook.proficio.model.FaqEntry;

import java.io.PrintWriter;
import java.io.StringWriter;

/**
 * The default implementation of Proficio.
 * @todo fill in the gaps
 */
public class DefaultProficio
    implements Proficio
{
    /** Number of times to loop in test. */
    private static final int COUNT = 10;

    /** Test variable. */
    private String unused;

    private ProficioStore store;

    /**
     * Add a FAQ entry.
```

Figure 6-6: An example source code cross reference

Figure 6-6 shows an example of the cross reference. Those familiar with Javadoc will recognize the framed navigation layout, however the content pane is now replaced with a syntax-highlighted, cross-referenced Java source file for the selected class. The hyper links in the content pane can be used to navigate to other classes and interfaces within the cross reference.

A useful way to leverage the cross reference is to use the links given for each line number in a source file to point team mates at a particular piece of code. Or, if you don't have the project open in your IDE, the links can be used to quickly find the source belonging to a particular exception.

Including JXR as a permanent fixture of the site for the project is simple, and can be done by adding the following to `proficio/pom.xml`:

```
...
<reporting>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jxr-plugin</artifactId>
  </plugin>
  ...
 </plugins>
</reporting>
...
```

You can now run `mvn site` in `proficio-core` and see the *Source Xref* item listed in the *Project Reports* menu of the generated site.

In most cases, the default JXR configuration is sufficient, however if you'd like a list of available configuration options, see the plugin reference at http://maven.apache.org/plugins/maven-jxr-plugin/.

Now that you have a source cross reference, many of the other reports demonstrated in this chapter will be able to link to the actual code to highlight an issue. However, browsing source code is too cumbersome for the developer if they only want to know about how the API works, so an equally important piece of reference material is the Javadoc report.

A Javadoc report is only as good as your Javadoc! Make sure you document the methods you intend to display in the report, and if possible use Checkstyle to ensure they are documented.

Using Javadoc is very similar to the JXR report and most other reports in Maven. Again, you can run it on its own using the following command:

```
C:\mvnbook\proficio\proficio-core> mvn javadoc:javadoc
```

Since it will be included as part of the project site, you should include it in `proficio/pom.xml` as a site report to ensure it is run every time the site is regenerated:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
</plugin>
...
```

The end result is the familiar Javadoc output, in `target/site/apidocs`.

Unlike JXR, the Javadoc report is quite configurable, with most of the command line options of the Javadoc tool available.

One useful option to configure is links. In the online mode, this will link to an external Javadoc reference at a given URL.

For example, the following configuration, when added to `proficio/pom.xml`, will link both the JDK 1.4 API documentation and the Plexus container API documentation used by Proficio:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-javadoc-plugin</artifactId>
 <configuration>
  <links>
   <link>http://java.sun.com/j2se/1.4.2/docs/api</link>
   <link>http://plexus.codehaus.org/ref/1.0-alpha-9/apidocs</link>
  </links>
 </configuration>
</plugin>
...
```

If you regenerate the site in `proficio-core` with `mvn site` again, you'll see that all references to the standard JDK classes such as `java.lang.String` and `java.lang.Object`, are linked to API documentation on the Sun website, as well as any references to classes in Plexus.

Setting up Javadoc has been very convenient, but it results in a separate set of API documentation for each library in a multi-module build. Since it is preferred to have discrete functional pieces separated into distinct modules, but conversely to have the Javadoc closely related, this is not sufficient.

One option would be to introduce links to the other modules (automatically generated by Maven based on dependencies, of course!), but this would still limit the available classes in the navigation as you hop from module to module. Instead, the Javadoc plugin provides a way to produce a single set of API documentation for the entire project.

Edit the configuration of the existing Javadoc plugin in `proficio/pom.xml` by adding the following line:

```
...
<configuration>
 <aggregate>true</aggregate>
 ...
</configuration>
...
```

When built from the top level project, this simple change will produce an aggregated Javadoc and ignore the Javadoc report in the individual modules. This setting must go into the reporting section so that it is used for both reports and if the command is executed separately. However, this setting is always ignored by the `javadoc:jar` goal, ensuring that the deployed Javadoc corresponds directly to the artifact with which it is deployed for use in an IDE.

Try running `mvn clean javadoc:javadoc` in the `proficio` directory to produce the aggregated Javadoc in `target/site/apidocs/index.html`.

Now that the sample application has a complete reference for the source code, the next section will allow you to start monitoring and improving its health.

6.7. Monitoring and Improving the Health of Your Source Code

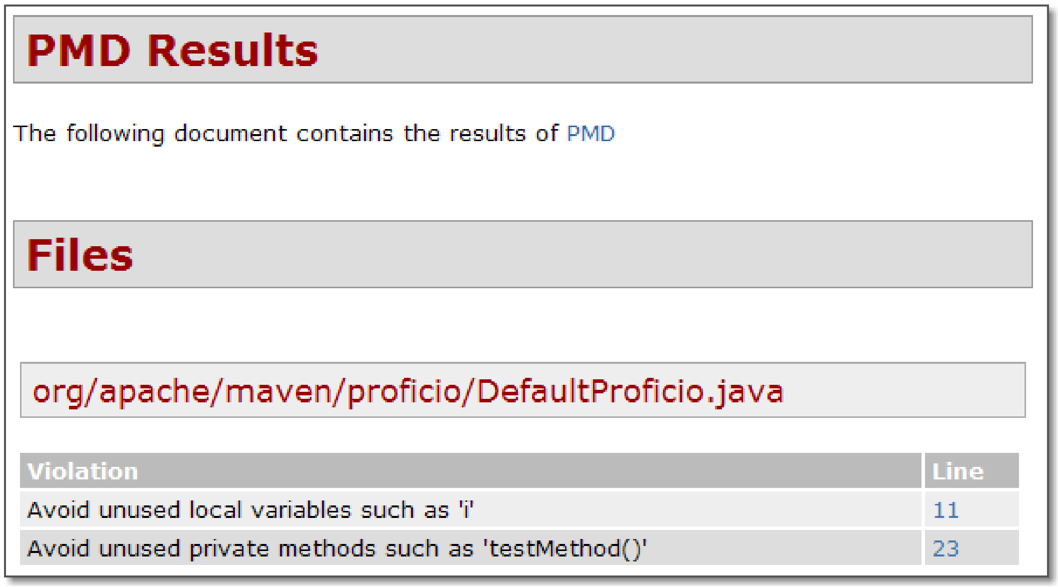
There are several factors that contribute to the health of your source code:

- **Accuracy** – whether the code does what it is expected to do
- **Robustness** – whether the code gracefully handles exceptional conditions
- **Extensibility** – how easily the code can be changed without affecting accuracy or requiring changes to a large amount of other code
- **Readability** – how easily the code can be understood (in a team environment, this is important for both the efficiency of other team members and also to increase the overall level of code comprehension, which in turn reduces the risk that its accuracy will be affected by change)

Maven has reports that can help with each of these health factors, and this section will look at three:

- PMD (http://pmd.sf.net/)
- Checkstyle (http://checkstyle.sf.net/)
- Tag List

PMD takes a set of either predefined or user-defined rule sets and evaluates the rules across your Java source code. The result can help identify bugs, copy-and-pasted code, and violations of a coding standard. Figure 6-7 shows the output of a PMD report on `proficio-core`, which is obtained by running `mvn pmd:pmd`.



PMD Results

The following document contains the results of PMD

Files

org/apache/maven/proficio/DefaultProficio.java

Violation	Line
Avoid unused local variables such as 'i'	11
Avoid unused private methods such as 'testMethod()'	23

Figure 6-7: An example PMD report

As you can see, some source files are identified as having problems that could be addressed, such as unused methods and variables. Also, since the JXR report was included earlier, the line numbers in the report are linked to the actual source code so you can browse the issues.

Adding the default PMD report to the site is just like adding any other report – you can include it in the reporting section in the `proficio/pom.xml` file:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-pmd-plugin</artifactId>
</plugin>
...
```

The default PMD report includes the *basic*, *unused code,* and *imports* rule sets. The "basic" rule set includes checks on empty blocks, unnecessary statements and possible bugs – such as incorrect loop variables. The "unused code" rule set will locate unused private fields, methods, variables and parameters. The "imports" rule set will detect duplicate, redundant or unused import declarations.

Adding new rule sets is easy, by passing the `rulesets` configuration to the plugin. However, if you configure these, you must configure *all* of them – including the defaults explicitly. For example, to include the default rules, and the `finalizer` rule sets, add the following to the plugin configuration you declared earlier:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-pmd-plugin</artifactId>
 <configuration>
  <rulesets>
   <ruleset>/rulesets/basic.xml</ruleset>
   <ruleset>/rulesets/imports.xml</ruleset>
   <ruleset>/rulesets/unusedcode.xml</ruleset>
   <ruleset>/rulesets/finalizers.xml</ruleset>
  </rulesets>
 </configuration>
</plugin>
...
```

You may find that you like some rules in a rule set, but not others. Or, you may use the same rule sets in a number of projects. In either case, you can choose to create a custom rule set. For example, you could create a rule set with all the default rules, but exclude the "unused private field" rule. To try this, create a file in the `proficio-core` directory of the sample application called `src/main/pmd/custom.xml`, with the following content:

```
<?xml version="1.0"?>
<ruleset name="custom">
 <description>
  Default rules, no unused private field warning
 </description>
 <rule ref="/rulesets/basic.xml" />
 <rule ref="/rulesets/imports.xml" />
 <rule ref="/rulesets/unusedcode.xml">
  <exclude name="UnusedPrivateField" />
 </rule>
</ruleset>
```

To use this rule set, change the configuration by overriding it in the `proficio-core/pom.xml` file, adding:

```
...
<reporting>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-pmd-plugin</artifactId>
   <configuration>
    <rulesets>
     <ruleset>${basedir}/src/main/pmd/custom.xml</ruleset>
    </rulesets>
   </configuration>
  </plugin>
 </plugins>
</reporting>
...
```

For more examples on customizing the rule sets, see the instructions on the PMD Web site at http://pmd.sf.net/howtomakearuleset.html. It is also possible to write your own rules if you find that existing ones do not cover recurring problems in your source code.

One important question is how to select appropriate rules. For PMD, try the following guidelines from the Web site at http://pmd.sf.net/bestpractices.html:

- Pick the rules that are right for you. There is no point having hundreds of violations you won't fix.
- Start small, and add more as needed. `basic`, `unusedcode`, and `imports` are useful in most scenarios and easily fixed. From this starting, select the rules that apply to your own project.

If you've done all the work to select the right rules and are correcting all the issues being discovered, you need to make sure it stays that way.

Try this now by running `mvn pmd:check` on `proficio-core`. You'll see that the build fails with the following 3 errors:

```
[INFO] ----------------------------------------------------------------------------
[INFO] Building Maven Proficio Core
[INFO]  task-segment: [pmd:check]
[INFO] ----------------------------------------------------------------------------
[INFO] Preparing pmd:check
[INFO] [pmd:pmd]
[INFO] [pmd:check]
[INFO] ----------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ----------------------------------------------------------------------------
[INFO] You have 3 PMD violations.
[INFO] ----------------------------------------------------------------------------
```

Before correcting these errors, you should include the check in the build, so that it is regularly tested. This is done by binding the goal to the build life cycle. To do so, add the following section to the `proficio/pom.xml` file:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-pmd-plugin</artifactId>
   <executions>
    <execution>
     <goals>
      <goal>check</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
  ...
 </plugins>
</build>
```

You may have noticed that there is no configuration here, but recall from *Configuring Reports and Checks* section of this chapter that the reporting configuration is applied to the build as well.

By default, the `pmd:check` goal is run in the verify phase, which occurs after the packaging phase. If you need to run checks earlier, you could add the following to the execution block to ensure that the check runs just after all sources exist:

```
<phase>process-sources</phase>
```

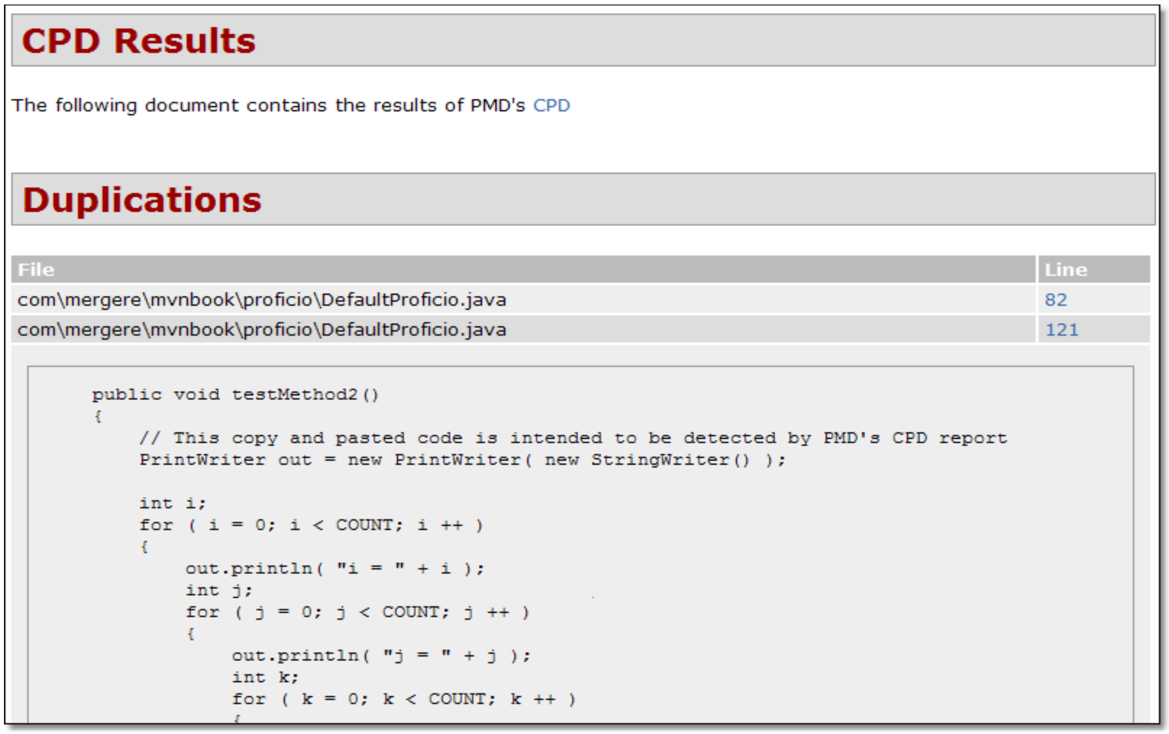
To test this new setting, try running `mvn` verify in the `proficio-core` directory. You will see that the build fails. To correct this, fix the errors in the `src/main/java/com/mergere/mvnbook/proficio/DefaultProficio.java` file by adding a `//NOPMD` comment to the unused variables and method:

```
...
// Trigger PMD and checkstyle
int i; // NOPMD
...
int j; // NOPMD
...
private void testMethod() // NOPMD
{
}
...
```

If you run `mvn verify` again, the build will succeed.

While this check is very useful, it can be slow and obtrusive during general development. For that reason, adding the check to a profile, which is executed only in an appropriate environment, can make the check optional for developers, but mandatory in an integration environment. See *Continuous Integration with Continuum* section in the next chapter for information on using profiles and continuous integration.

While the PMD report allows you to run a number of different rules, there is one that is in a separate report. This is the CPD, or copy/paste detection report, and it includes a list of duplicate code fragments discovered across your entire source base. An example report is shown in figure 6-8.This report is included by default when you enable the PMD plugin in your reporting section, and will appear as “CPD report” in the Project Reports menu.



CPD Results

The following document contains the results of PMD's CPD

Duplications

File	Line
com\mergere\mvnbook\proficio\DefaultProficio.java	82
com\mergere\mvnbook\proficio\DefaultProficio.java	121

```
    public void testMethod2()
    {
        // This copy and pasted code is intended to be detected by PMD's CPD report
        PrintWriter out = new PrintWriter( new StringWriter() );

        int i;
        for ( i = 0; i < COUNT; i ++ )
        {
            out.println( "i = " + i );
            int j;
            for ( j = 0; j < COUNT; j ++ )
            {
                out.println( "j = " + j );
                int k;
                for ( k = 0; k < COUNT; k ++ )
```

Figure 6-8: An example CPD report

In a similar way to the main check, `pmd:cpd-check` can be used to enforce a failure if duplicate source code is found. However, the CPD report contains only one variable to configure: `minimumTokenCount`, which defaults to 100. With this setting you can fine tune the size of the copies detected. This may not give you enough control to effectively set a rule for the source code, resulting in developers attempting to avoid detection by making only slight modifications, rather than identifying a possible factoring of the source code. Whether to use the report only, or to enforce a check will depend on the environment in which you are working.

There are other alternatives for copy and paste detection, such as Checkstyle, and a commercial product called Simian (http://www.redhillconsulting.com.au/products/simian/). Simian can also be used through Checkstyle and has a larger variety of configuration options for detecting duplicate source code.

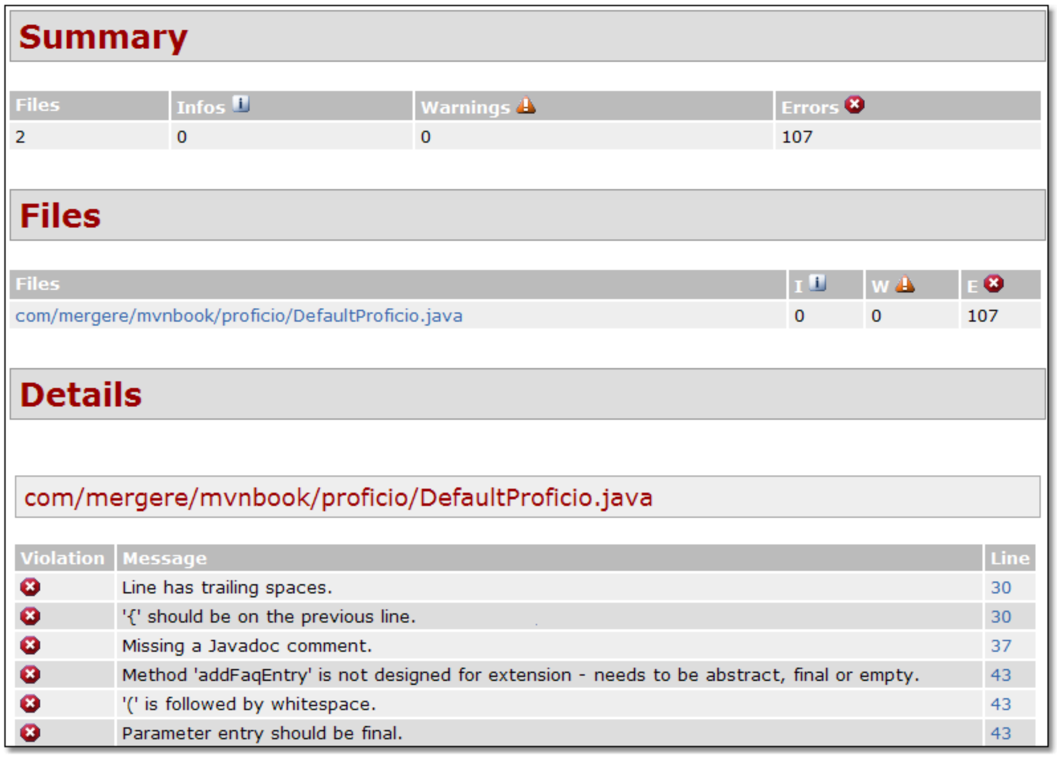
Checkstyle is a tool that is, in many ways, similar to PMD. It was originally designed to address issues of format and style, but has more recently added checks for other code issues.

Depending on your environment, you may choose to use it in one of the following ways:

- Use it to check code formatting only, and rely on other tools for detecting other problems.
- Use it to check code formatting and selected other problems, and still rely on other tools for greater coverage.
- Use it to check code formatting and to detect other problems exclusively

This section focuses on the first usage scenario. If you need to learn more about the available modules in Checkstyle, refer to the list on the Web site at http://checkstyle.sf.net/availablechecks.html.

Figure 6-9 shows the Checkstyle report obtained by running `mvn checkstyle:checkstyle` from the `proficio-core` directory. Some of the extra summary information for overall number of errors and the list of checks used has been trimmed from this display.



Summary

Files	Infos	Warnings	Errors
2	0	0	107

Files

Files	I	W	E
com/mergere/mvnbook/proficio/DefaultProficio.java	0	0	107

Details

com/mergere/mvnbook/proficio/DefaultProficio.java

Violation	Message	Line
✖	Line has trailing spaces.	30
✖	'{' should be on the previous line.	30
✖	Missing a Javadoc comment.	37
✖	Method 'addFaqEntry' is not designed for extension - needs to be abstract, final or empty.	43
✖	'(' is followed by whitespace.	43
✖	Parameter entry should be final.	43

Figure 6-9: An example Checkstyle report

You'll see that each file with notices, warnings or errors is listed in a summary, and then the errors are shown, with a link to the corresponding source line – if the JXR report was enabled.

That's a lot of errors! By default, the rules used are those of the Sun Java coding conventions, but Proficio is using the Maven team's code style.

This style is also bundled with the Checkstyle plugin, so to include the report in the site and configure it to use the Maven style, add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-checkstyle-plugin</artifactId>
 <configuration>
  <configLocation>config/maven_checks.xml</configLocation>
 </configuration>
</plugin>
```

Table 6-3 shows the configurations that are built into the Checkstyle plugin.

Table 6-3: Built-in Checkstyle configurations

Configuration	Description	Reference
`config/sun_checks.xml`	Sun Java Coding Conventions	http://java.sun.com/docs/codeconv/
`config/maven_checks.xml`	Maven team's coding conventions	http://maven.apache.org/guides/development/guide-m2-development.html#Maven%20Code%20Style
`config/turbine_checks.xml`	Conventions from the Jakarta Turbine project	http://jakarta.apache.org/turbine/common/code-standards.html
`config/avalon_checks.xml`	Conventions from the Apache Avalon project	No longer online – the Avalon project has closed. These checks are for backwards compatibility only.

The `configLocation` parameter can be set to a file within your build, a URL, or a resource within a special dependency also.

It is a good idea to reuse an existing Checkstyle configuration for your project if possible – if the style you use is common, then it is likely to be more readable and easily learned by people joining your project. The built-in Sun and Maven standards are quite different, and typically, one or the other will be suitable for most people. However, if you have developed a standard that differs from these, or would like to use the additional checks introduced in Checkstyle 3.0 and above, you will need to create a Checkstyle configuration.

While this chapter will not go into an example of how to do this, the Checkstyle documentation provides an excellent reference at http://checkstyle.sf.net/config.html.

The Checkstyle plugin itself has a large number of configuration options that allow you to customize the appearance of the report, filter the results, and to parameterize the Checkstyle configuration for creating a baseline organizational standard that can be customized by individual projects. It is also possible to share a Checkstyle configuration among multiple projects, as explained at http://maven.apache.org/plugins/maven-checkstyle-plugin/tips.html.

Before completing this section it is worth mentioning the Tag List plugin. This report, known as “Task List” in Maven 1.0, will look through your source code for known tags and provide a report on those it finds. By default, this will identify the tags `TODO` and `@todo` in the comments of your source code.

To try this plugin, add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>taglist-maven-plugin</artifactId>
 <configuration>
  <tags>
   <tag>TODO</tag>
   <tag>@todo</tag>
   <tag>FIXME</tag>
   <tag>XXX</tag>
  </tags>
 </configuration>
</plugin>
...
```

This configuration will locate any instances of `TODO`, `@todo`, `FIXME`, or `XXX` in your source code. It is actually possible to achieve this using Checkstyle or PMD rules, however this plugin is a more convenient way to get a simple report of items that need to be addressed at some point later in time.

PMD, Checkstyle, and Tag List are just three of the many tools available for assessing the health of your project's source code. Some other similar tools, such as FindBugs, JavaNCSS and JDepend, have beta versions of plugins available from the http://mojo.codehaus.org/ project at the time of this writing, and more plugins are being added every day.

6.8. Monitoring and Improving the Health of Your Tests

One of the important (and often controversial) features of Maven is the emphasis on testing as part of the production of your code. In the build life cycle defined in Chapter 2, you saw that tests are run *before* the packaging of the library or application for distribution, based on the theory that you shouldn't even try to use something before it has been tested. There are additional testing stages that can occur after the packaging step to verify that the assembled package works under other circumstances.

As you learned in section 6.2, *Setting Up the Project Web Site*, it is easy to add a report to the Web site that shows the results of the tests that have been run. While the default Surefire configuration fails the build if the tests fail, the report (run either on its own, or as part of the site), will ignore these failures when generated to show the current test state. Failing the build is still recommended – but the report allows you to provide a better visual representation of the results. In addition to that, it can be a useful report for demonstrating the number of tests available and the time it takes to run certain tests for a package.

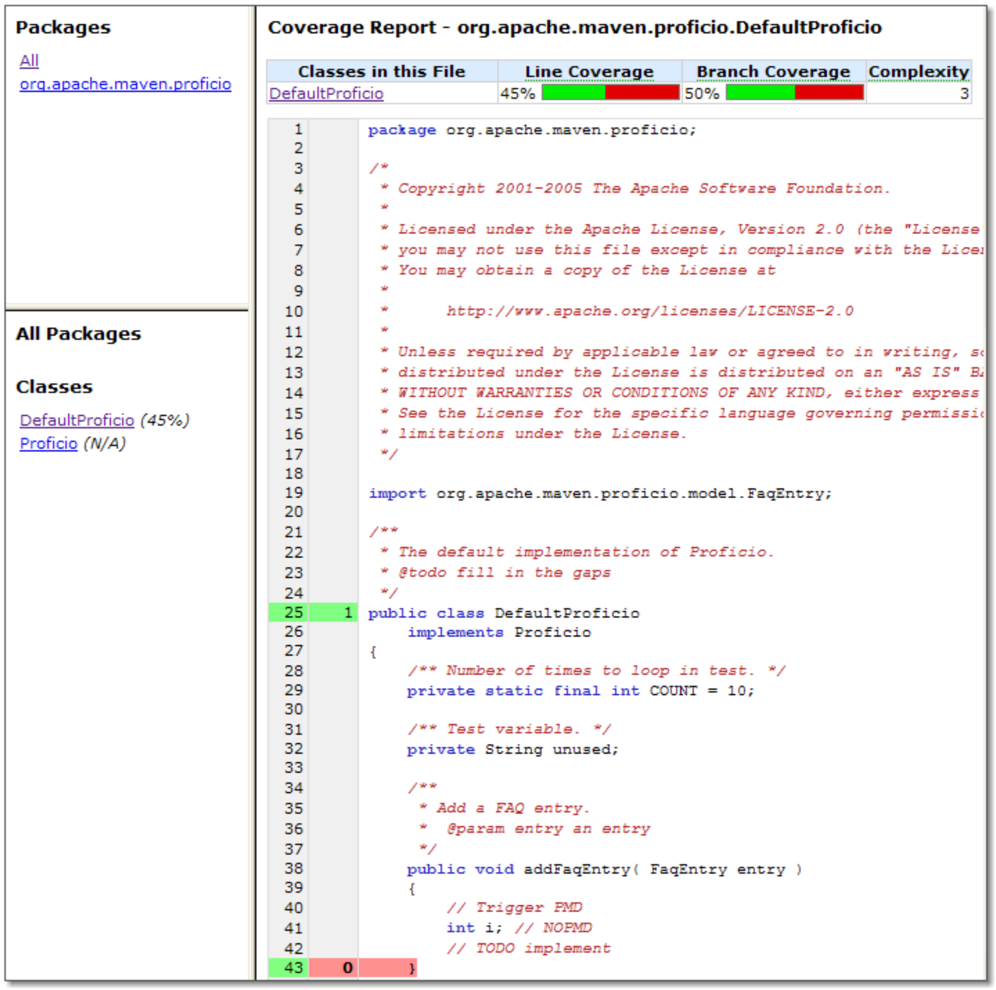
Knowing whether your tests pass is an obvious and important assessment of their health. Another critical technique is to determine how much of your source code is covered by the test execution. At the time of writing, for assessing coverage, Cobertura (http://cobertura.sf.net) is the open source tool best integrated with Maven. While you are writing your tests, using this report on a regular basis can be very helpful in spotting any holes in the test plan.

To see what Cobertura is able to report, run `mvn cobertura:cobertura` in the `proficio-core` directory of the sample application. Figure 6-10 shows the output that you can view in `target/site/cobertura/index.html`.

The report contains both an overall summary, and a line-by-line coverage analysis of each source file, in the familiar Javadoc style framed layout. For a source file, you'll notice the following markings:

- Unmarked lines are those that do not have any executable code associated with them. This includes method and class declarations, comments and white space.
- Each line with an executable statement has a number in the second column that indicates during the test run how many times a particular statement was run.
- Lines in red are statements that were not executed (if the count is 0), or for which all possible branches were not executed. For example, a branch is an if statement that can behave differently depending on whether the condition is true or false.

Unmarked lines with a green number in the second column are those that have been completely covered by the test execution.



Packages

All
org.apache.maven.proficio

All Packages

Classes

DefaultProficio *(45%)*
Proficio *(N/A)*

Coverage Report - org.apache.maven.proficio.DefaultProficio

Classes in this File	Line Coverage	Branch Coverage	Complexity
DefaultProficio	45% 🟩🟥	50% 🟩🟥	3

```
 1     package org.apache.maven.proficio;
 2
 3     /*
 4      * Copyright 2001-2005 The Apache Software Foundation.
 5      *
 6      * Licensed under the Apache License, Version 2.0 (the "License
 7      * you may not use this file except in compliance with the Lice
 8      * You may obtain a copy of the License at
 9      *
10      *      http://www.apache.org/licenses/LICENSE-2.0
11      *
12      * Unless required by applicable law or agreed to in writing, s
13      * distributed under the License is distributed on an "AS IS" B
14      * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
15      * See the License for the specific language governing permissi
16      * limitations under the License.
17      */
18
19     import org.apache.maven.proficio.model.FaqEntry;
20
21     /**
22      * The default implementation of Proficio.
23      * @todo fill in the gaps
24      */
25   1 public class DefaultProficio
26         implements Proficio
27     {
28         /** Number of times to loop in test. */
29         private static final int COUNT = 10;
30
31         /** Test variable. */
32         private String unused;
33
34         /**
35          * Add a FAQ entry.
36          *  @param entry an entry
37          */
38         public void addFaqEntry( FaqEntry entry )
39         {
40             // Trigger PMD
41             int i; // NOPMD
42             // TODO implement
43   0     }
```

Figure 6-10: An example Cobertura report

The complexity indicated in the top right is the cyclomatic complexity of the methods in the class, which measures the number of branches that occur in a particular method. High numbers (for example, over 10), might indicate a method should be re-factored into simpler pieces, as it can be hard to visualize and test the large number of alternate code paths. If this is a metric of interest, you might consider having PMD monitor it.

The Cobertura report doesn't have any notable configuration, so including it in the site is simple. Add the following to the reporting section of `proficio/pom.xml`:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>cobertura-maven-plugin</artifactId>
</plugin>
...
```

If you now run `mvn site` under `proficio-core`, the report will be generated in `target/site/cobertura/index.html`.

While not required, there is another useful setting to add to the *build* section. Due to a hard-coded path in Cobertura, the database used is stored in the project directory as `cobertura.ser`, and is not cleaned with the rest of the project. To ensure that this happens, add the following to the build section of `proficio/pom.xml`:

```
...
<build>
 <plugins>
  <plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>cobertura-maven-plugin</artifactId>
   <executions>
    <execution>
     <id>clean</id>
     <goals>
      <goal>clean</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
...
```

If you now run `mvn clean` in `proficio-core`, you'll see that the `cobertura.ser` file is deleted, as well as the target directory.

The Cobertura plugin also contains a goal called `cobertura:check` that is used to ensure that the coverage of your source code is maintained at a certain percentage.

To configure this goal for Proficio, add a configuration and another execution to the build plugin definition you added above when cleaning the Cobertura database:

```
...
<configuration>
 <check>
  <totalLineRate>100</totalLineRate>
  <totalBranchRate>100</totalBranchRate>
 </check>
</configuration>
<executions>
 ...
 <execution>
  <id>check</id>
  <goals>
   <goal>check</goal>
  </goals>
 </execution>
</executions>
...
```

Note that the configuration element is *outside* of the executions. This ensures that if you run `mvn cobertura:check` from the command line, the configuration will be applied. This wouldn't be the case if it were associated with the life-cycle bound check execution.

If you now run `mvn verify` under `proficio-core`, the check will be performed.

You'll notice that your tests are run twice. This is because Cobertura needs to instrument your class files, and the tests are re-run using those class files instead of the normal ones (however, these are instrumented in a separate directory, so are not packaged in your application). The Surefire report may also re-run tests if they were already run – both of these are due to a limitation in the way the life cycle is constructed that will be improved in future versions of Maven.

The rules that are being used in this configuration are 100% overall line coverage rate, and 100% branch coverage rate. You would have seen in the previous examples that there were some lines not covered, so running the check fails.

Normally, you would add unit tests for the functions that are missing tests, as in the Proficio example. However, looking through the report, you may decide that only some exceptional cases are untested, and decide to reduce the overall average required. You can do this for Proficio to have the tests pass by changing the setting in `proficio/pom.xml`:

```
...
<configuration>
 <check>
  <totalLineRate>80</totalLineRate>
...
```

If you run `mvn` verify again, the check passes.

These settings remain quite demanding though, only allowing a small number of lines to be untested. This will allow for some constructs to remain untested, such as handling checked exceptions that are unexpected in a properly configured system and difficult to test. It is just as important to allow these exceptions, as it is to require that the other code be tested. Remember, the easiest way to increase coverage is to remove code that handles untested, exceptional cases – and that's certainly not something you want!

The settings above are requirements for averages across the entire source tree. You may want to enforce this for each file individually as well, using `lineRate` and `branchRate`, or as the average across each package, using `packageLineRate` and `packageBranchRate`. It is also possible to set requirements on individual packages or classes using the `regexes` parameter. For more information, refer to the Cobertura plugin configuration reference at http://mojo.codehaus.org/cobertura-maven-plugin.

Choosing appropriate settings is the most difficult part of configuring any of the reporting metrics in Maven. Some helpful hints for determining the right code coverage settings are:

- Like all metrics, involve the whole development team in the decision, so that they understand and agree with the choice.
- Don't set it too low, as it will become a minimum benchmark to attain and rarely more.
- Don't set it too high, as it will discourage writing code to handle exceptional cases that aren't being tested.
- Set some known guidelines for what type of code can remain untested.
- Consider setting any package rates higher than the per-class rate, and setting the total rate higher than both.
- Remain flexible – consider changes over time rather than hard and fast rules. Choose to reduce coverage requirements on particular classes or packages rather than lowering them globally.

Cobertura is not the only solution available for assessing test coverage. The best known commercial offering is Clover, which is very well integrated with Maven as well. It behaves very similarly to Cobertura, and you can evaluate it for 30 days when used in conjunction with Maven. For more information, see the Clover plugin reference on the Maven Web site at http://maven.apache.org/plugins/maven-clover-plugin/.

Of course, there is more to assessing the health of tests than success and coverage. These reports won't tell you if all the features have been implemented – this requires functional or acceptance testing. It also won't tell you whether the results of untested input values produce the correct results. Tools like Jester (http://jester.sf.net), although not yet integrated with Maven directly, may be of assistance there. Jester mutates the code that you've already determined is covered and checks that it causes the test to fail when run a second time with the wrong code.

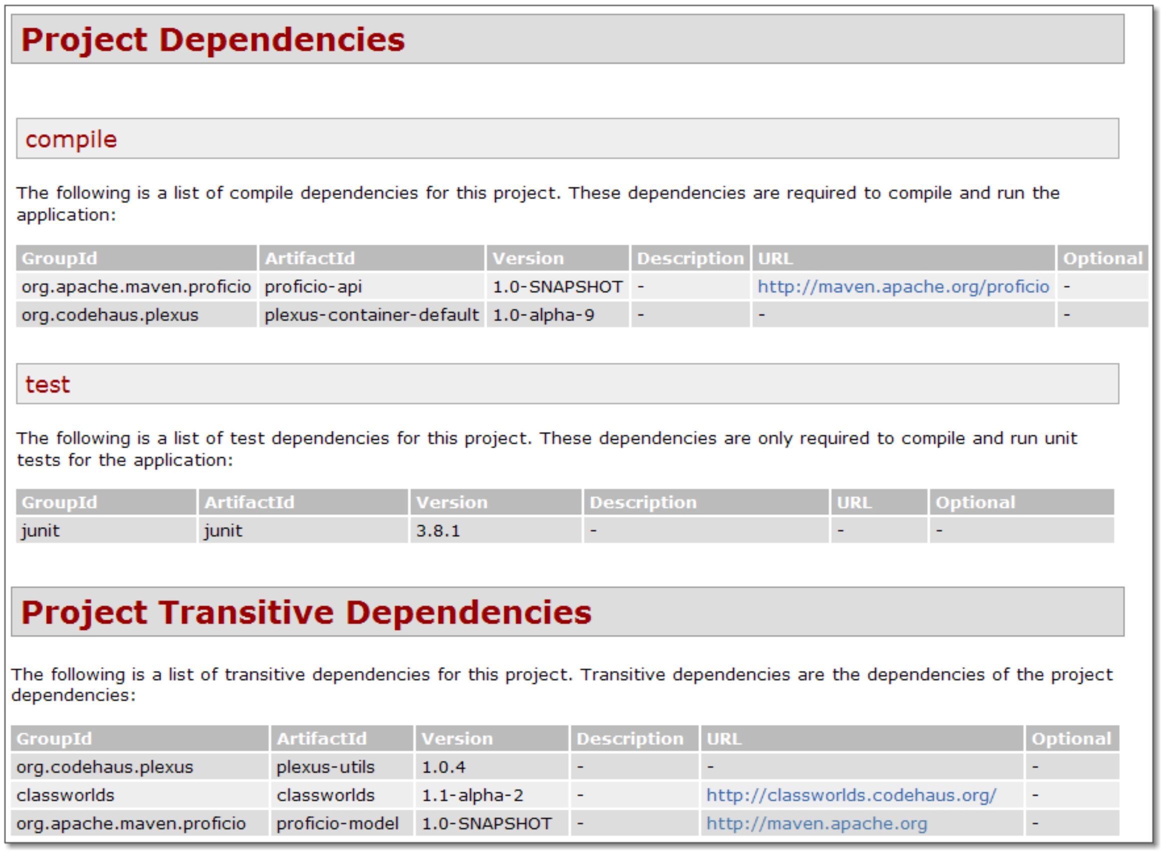
To conclude this section on testing, it is worth noting that one of the benefits of Maven's use of the Surefire abstraction is that the tools above will work for any type of runner introduced. For example, Surefire supports tests written with TestNG, and at the time of writing experimental JUnit 4.0 support is also available. In both cases, these reports work unmodified with those test types. If you have another tool that can operate under the Surefire framework, it is possible for you to write a provider to use the new tool, and get integration with these other tools for free.

6.9. Monitoring and Improving the Health of Your Dependencies

Many people use Maven primarily as a dependency manager. While this is only one of Maven's features, used well it is a significant time saver.

Maven 2.0 introduced transitive dependencies, where the dependencies of dependencies are included in a build, and a number of other features such as scoping and version selection. This brought much more power to Maven's dependency mechanism, but does introduce a drawback: poor dependency maintenance or poor scope and version selection affects not only your own project, but any projects that depend on your project. Left unchecked, the full graph of a project's dependencies can quickly balloon in size and start to introduce conflicts.

The first step to effectively maintaining your dependencies is to review the standard report included with the Maven site. If you haven't done so already, run `mvn site` in the `proficio-core` directory, and browse to the file generated in `target/site/dependencies.html`. The result is shown in figure 6-11.



Project Dependencies

compile

The following is a list of compile dependencies for this project. These dependencies are required to compile and run the application:

GroupId	ArtifactId	Version	Description	URL	Optional
org.apache.maven.proficio	proficio-api	1.0-SNAPSHOT	-	http://maven.apache.org/proficio	-
org.codehaus.plexus	plexus-container-default	1.0-alpha-9	-	-	-

test

The following is a list of test dependencies for this project. These dependencies are only required to compile and run unit tests for the application:

GroupId	ArtifactId	Version	Description	URL	Optional
junit	junit	3.8.1	-	-	-

Project Transitive Dependencies

The following is a list of transitive dependencies for this project. Transitive dependencies are the dependencies of the project dependencies:

GroupId	ArtifactId	Version	Description	URL	Optional
org.codehaus.plexus	plexus-utils	1.0.4	-	-	-
classworlds	classworlds	1.1-alpha-2	-	http://classworlds.codehaus.org/	-
org.apache.maven.proficio	proficio-model	1.0-SNAPSHOT	-	http://maven.apache.org	-

Figure 6-11: An example dependency report

This report shows detailed information about your direct dependencies, but more importantly in the second section it will list all of the transitive dependencies included through those dependencies. It's here that you might see something that you didn't expect – an extra dependency, an incorrect version, or an incorrect scope – and choose to investigate its inclusion.

Currently, this requires running your build with debug turned on, such as `mvn -X package`. This will output the dependency tree as it is calculated, using indentation to indicate which dependencies introduce other dependencies, as well as comments about what versions and scopes are selected, and why. For example, here is the resolution process of the dependencies of `proficio-core` (some fields have been omitted for brevity):

```
proficio-core:1.0-SNAPSHOT
 junit:3.8.1 (selected for test)
 plexus-container-default:1.0-alpha-9 (selected for compile)
  plexus-utils:1.0.4 (selected for compile)
  classworlds:1.1-alpha-2 (selected for compile)
  junit:3.8.1 (not setting scope to: compile; local scope test wins)
 proficio-api:1.0-SNAPSHOT (selected for compile)
  proficio-model:1.0-SNAPSHOT (selected for compile)
```

Here you can see that, for example, `proficio-model` is introduced by `proficio-api`, and that plexus-container-default attempts to introduce `junit` as a compile dependency, but that it is overridden by the test scoped dependency in `proficio-core`.

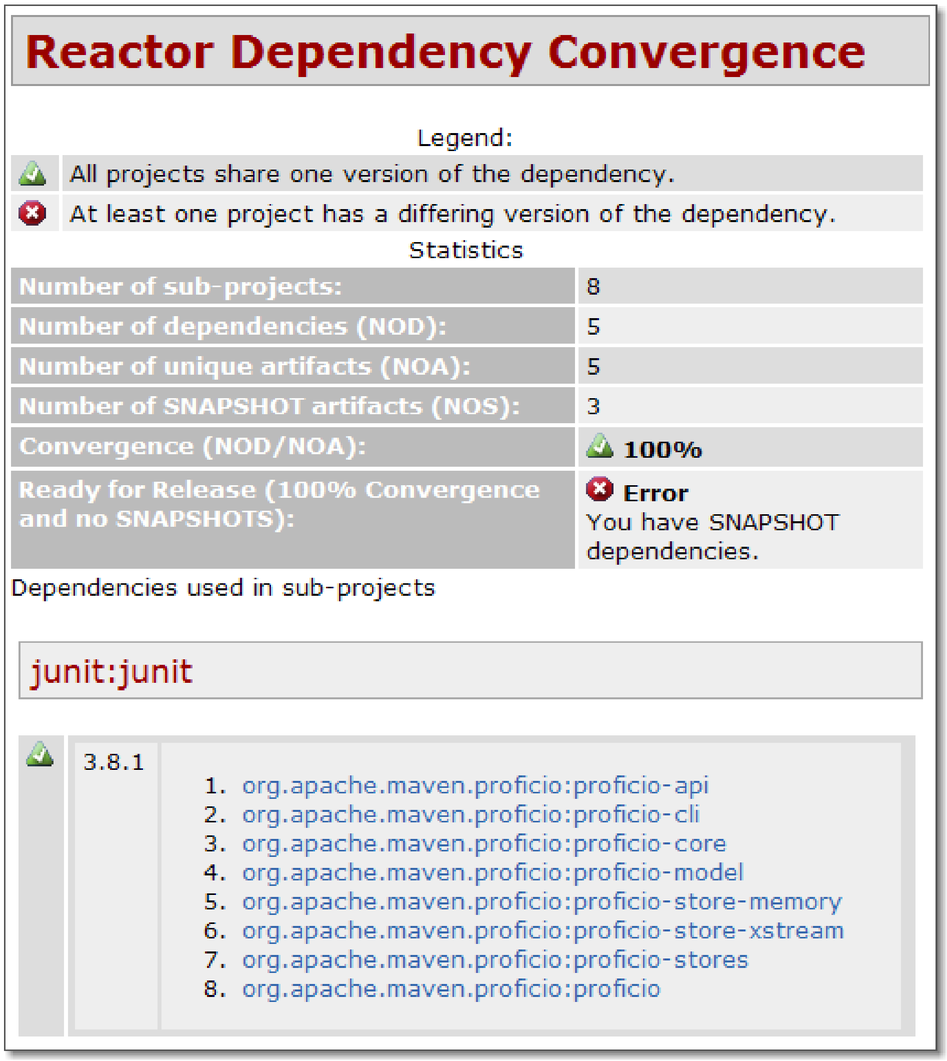
This can be quite difficult to read, so at the time of this writing there are two features in progress that are aimed at helping in this area:

- The Maven Repository Manager will allow you to navigate the dependency tree through the metadata stored in the Ibiblio repository.
- A dependency graphing plugin that will render a graphical representation of the information.

Another report that is available is the “Dependency Convergence Report”. This report is also a standard report, but appears in a multi-module build only. To see the report for the Proficio project, run `mvn site` from the base `proficio` directory. The file `target/site/dependency-convergence.html` will be created, and is shown in figure 6-12.

The report shows all of the dependencies included in all of the modules within the project. It also includes some statistics and reports on two important factors:

- Whether the versions of dependencies used for each module is in alignment. This helps ensure your build is consistent and reduces the probability of introducing an accidental incompatibility.
- Whether there are outstanding SNAPSHOT dependencies in the build, which indicates dependencies that are in development, and must be updated before the project can be released.



Reactor Dependency Convergence

Legend:

✓	All projects share one version of the dependency.
✗	At least one project has a differing version of the dependency.

Statistics

Number of sub-projects:	8
Number of dependencies (NOD):	5
Number of unique artifacts (NOA):	5
Number of SNAPSHOT artifacts (NOS):	3
Convergence (NOD/NOA):	✓ **100%**
Ready for Release (100% Convergence and no SNAPSHOTS):	✗ **Error** You have SNAPSHOT dependencies.

Dependencies used in sub-projects

junit:junit

✓	3.8.1	1. org.apache.maven.proficio:proficio-api 2. org.apache.maven.proficio:proficio-cli 3. org.apache.maven.proficio:proficio-core 4. org.apache.maven.proficio:proficio-model 5. org.apache.maven.proficio:proficio-store-memory 6. org.apache.maven.proficio:proficio-store-xstream 7. org.apache.maven.proficio:proficio-stores 8. org.apache.maven.proficio:proficio

Figure 6-12: The dependency convergence report

These reports are passive – there are no associated checks for them. However, they can provide basic help in identifying the state of your dependencies once you know what to find. To improve your project's health and the ability to reuse it as a dependency itself, try the following recommendations for your dependencies:

- Look for dependencies in your project that are no longer used
- Check that the scope of your dependencies are set correctly (to test if only used for unit tests, or runtime if it is needed to bundle with or run the application but not for compiling your source code).
- Use a range of supported dependency versions, declaring the absolute minimum supported as the lower boundary, rather than using the latest available. You can control what version is actually used by declaring the dependency version in a project that packages or runs the application.

- Add exclusions to dependencies to remove poorly defined dependencies from the tree. This is particularly the case for dependencies that are optional and unused by your project.

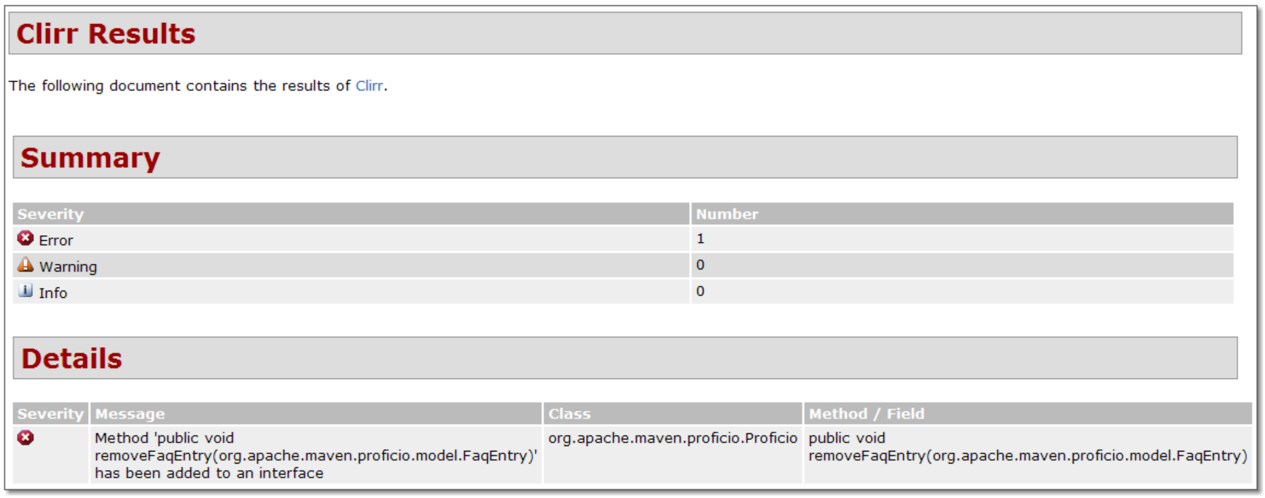
Given the importance of this task, more tools are needed in Maven. Two that are in progress were listed above, but there are plans for more:

- A class analysis plugin that helps identify dependencies that are unused in your current project
- Improved dependency management features including different mechanisms for selecting versions that will allow you to deal with conflicting versions, specification dependencies that let you depend on an API and manage the implementation at runtime, and more.

6.10. Monitoring and Improving the Health of Your Releases

Releasing a project is one of the most important procedures you will perform, but it is often tedious and error prone. While the next chapter will go into more detail about how Maven can help automate that task and make it more reliable, this section will focus on improving the quality of the code released, and the information released with it.

An important tool in determining whether a project is ready to be released is Clirr (http://clirr.sf.net/). Clirr detects whether the current version of a library has introduced any binary incompatibilities with the previous release. Catching these before a release can eliminate problems that are quite difficult to resolve once the code is "in the wild". An example Clirr report is shown in figure 6-13.



Clirr Results

The following document contains the results of Clirr.

Summary

Severity	Number
Error	1
Warning	0
Info	0

Details

Severity	Message	Class	Method / Field
✖	Method 'public void removeFaqEntry(org.apache.maven.proficio.model.FaqEntry)' has been added to an interface	org.apache.maven.proficio.Proficio	public void removeFaqEntry(org.apache.maven.proficio.model.FaqEntry)

Figure 6-13: An example Clirr report

This is particularly important if you are building a library or framework that will be consumed by developers outside of your own project.

Libraries will often be substituted by newer versions to obtain new features or bug fixes, but then expected to continue working as they always have.

Because existing libraries are not recompiled every time a version is changed, there is no verification that a library is binary-compatible – incompatibility will be discovered only when there's a failure.

But does binary compatibility apply if you are not developing a library for external consumption? While it may be of less importance, the answer here is clearly – yes. As a project grows, the interactions between the project's own components will start behaving as if they were externally-linked. Different modules may use different versions, or a quick patch may need to be made and a new version deployed into an existing application.

This is particularly true in a Maven-based environment, where the dependency mechanism is based on the assumption of binary compatibility between versions. While methods of marking incompatibility are planned for future versions, Maven currently works best if any version of an artifact is backwards compatible, back to the first release.

By default, the Clirr report shows only errors and warnings. However, you can configure the plugin to show all informational messages, by setting the `minSeverity` parameter. This gives you an overview of all the changes since the last release, even if they are binary compatible. To see this in action, add the following to the reporting section of `proficio-api`/`pom.xml`:

```
...
<reporting>
 <plugins>
  <plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>clirr-maven-plugin</artifactId>
   <configuration>
    <minSeverity>info</minSeverity>
   </configuration>
  </plugin>
 </plugins>
</reporting>
...
```

If you run `mvn site` in `proficio-api`, the report will be generated in `target/site/clirr-report.html`. You can obtain the same result by running the report on its own using `mvn clirr:clirr`.

If you run either of these commands, you'll notice that Maven reports that it is using version 0.9 of `proficio-api` against which to compare (and that it is downloaded if you don't have it already):

```
...
[INFO] [clirr:clirr]
[INFO] Comparing to version: 0.9
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
...
```

This version is determined by looking for the newest release in repository, that is before the current development version.

You can change the version used with the `comparisonVersion` parameter. For example, to compare the current code to the 0.8 release, run the following command:

```
mvn clirr:clirr -DcomparisonVersion=0.8
```

These versions of `proficio-api` are retrieved from the repository, however you can see the original sources by extracting the `Code_Ch06-2.zip` file.

You'll notice there are a more errors in the report, since this early development version had a different API, and later was redesigned to make sure that version 1.0 would be more stable in the long run.

It is best to make changes earlier in the development cycle, so that fewer people are affected. The longer poor choices remain, the harder they are to change as adoption increases. Once a version has been released that is intended to remain binary-compatible going forward, it is almost always preferable to deprecate an old API and add a new one, delegating the code, rather than removing or changing the original API and breaking binary compatibility.

In this instance, you are monitoring the `proficio-api` component for binary compatibility changes only. This is the most important one to check, as it will be used as the interface into the implementation by other applications. If it is the only one that the development team will worry about breaking, then there is no point in checking the others – it will create noise that devalues the report's content in relation to the important components.

However, if the team is prepared to do so, it is a good idea to monitor as many components as possible. Even if they are designed only for use inside the project, there is nothing in Java preventing them from being used elsewhere, and it can assist in making your own project more stable.

Like all of the quality metrics, it is important to agree up front, on the acceptable incompatibilities, to discuss and document the practices that will be used, and to check them automatically. The Clirr plugin is also capable of automatically checking for introduced incompatibilities through the `clirr:check` goal.

To add the check to the `proficio-api/pom.xml` file, add the following to the build section:

```
...
<build>
 <plugins>
  <plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>clirr-maven-plugin</artifactId>
   <executions>
    <execution>
     <goals>
      <goal>check</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
  ...
 </plugins>
</build>
...
```

If you now run `mvn verify`, you will see that the build fails due to the binary incompatibility introduced between the 0.9 preview release and the final 1.0 version. Since this was an acceptable incompatibility due to the preview nature of the 0.9 release, you can choose to exclude that from the report by adding the following configuration to the plugin:

```
...
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>clirr-maven-plugin</artifactId>
 <configuration>
  <excludes>
   <exclude>**/Proficio</exclude>
  </excludes>
 </configuration>
 ...
</plugin>
```

This will prevent failures in the Proficio class from breaking the build in the future. Note that in this instance, it is listed only in the *build* configuration, so the report still lists the incompatibility. This allows the results to be collected over time to form documentation about known incompatibilities for applications using the library.

A limitation of this feature is that it will eliminate a class entirely, not just the one acceptable failure. Hopefully a future version of Clirr will allow acceptable incompatibilities to be documented in the source code, and ignored in the same way that PMD does.

With this simple setup, you can create a very useful mechanism for identifying potential release disasters much earlier in the development process, and then act accordingly. While the topic of designing a strong public API and maintaining binary compatibility is beyond the scope of this book, the following articles and books can be recommended:

- Evolving Java-based APIs contains a description of the problem of maintaining binary compatibility, as well as strategies for evolving an API without breaking it.
- Effective Java describes a number of practical rules that are generally helpful to writing code in Java, and particularly so if you are designing a public API.

A similar tool to Clirr that can be used for analyzing changes between releases is JDiff. Built as a Javadoc doclet, it takes a very different approach, taking two source trees and comparing the differences in method signatures and Javadoc annotations. This can be useful in getting a greater level of detail than Clirr on specific class changes. However, it will not pinpoint potential problems for you, and so is most useful for browsing. It has a functional Maven 2 plugin, which is available at http://mojo.codehaus.org/jdiff-maven-plugin.

6.11. Viewing Overall Project Health

In the previous sections of this chapter, a large amount of information was presented about a project, each in discrete reports. Some of the reports linked to one another, but none related information from another report to itself, and few of the reports aggregated information across a multiple module build. Finally, none of the reports presented how the information changes over time other than the release announcements. These are all important features to have to get an overall view of the health of a project. While some attempts were made to address this in Maven 1.0 (for example, the Dashboard plugin), they did not address all of these requirements, and have not yet been implemented for Maven 2.0.

However, it should be noted that the Maven reporting API was written with these requirements in mind specifically, and as the report set stabilizes – summary reports will start to appear.

In the absence of these reports, enforcing good, individual checks that fail the build when they're not met, will reduce the need to gather information from various sources about the health of the project, as there is a constant background monitor that ensures the health of the project is being maintained.

6.12. Summary

The power of Maven's declarative project model is that with a very simple setup (often only 4 lines in `pom.xml`), a new set of information about your project can be added to a shared Web site to help your team visualize the health of the project. Best of all, the model remains flexible enough to make it easy to extend and customize the information published on your project web site.

However, it is important that your project information not remain passive. Most Maven plugins allow you to integrate rules into the build that check certain constraints on that piece of information once it is well understood. The purpose, then, of the visual display is to aid in deriving the appropriate constraints to use.

How well this works in your own projects will depend on the development culture of your team. It is important that developers are involved in the decision making process regarding build constraints, so that they feel that they are achievable. In some cases, it requires a shift from a focus on time and deadlines, to a focus on quality. Once established, this focus and automated monitoring will have the natural effect of improving productivity and reducing time of delivery again.

The additions and changes to Proficio made in this chapter can be found in the `Code_Ch06-1.zip` source archive, and will be used as the basis for the next chapter.

The next chapter examines team development and collaboration, and incorporates the concepts learned in this chapter, along with techniques to ensure that the build checks are now automated, regularly scheduled, and run in the appropriate environment.