

testing; see section 4.3.1). This activity is inextricably linked with coding and happens at the same time. Unit tests can ensure that your application is under test from the very beginning.

Of course, your application should undergo other forms of software testing, starting with unit tests and ending with acceptance tests. The previous section outlined the other types of software tests that should be applied to your application.

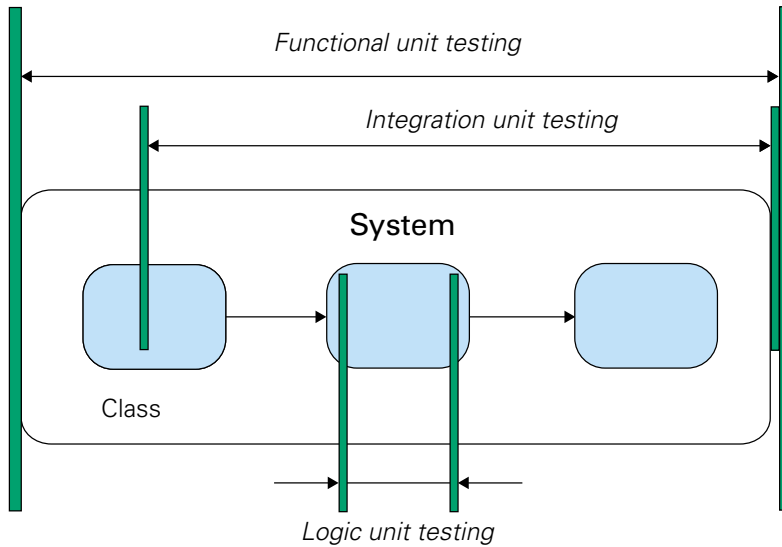
Most applications are divided into subsystems. As a developer, you want to ensure that each of your subsystems works correctly. As you write code, your first tests will probably be logic unit tests. As you write more tests and more code, you will begin to add integration and functional unit tests. At any one time, you will probably be working on a logic unit test, an integration unit test, or a functional unit test. Table 4.3 summarizes the different types of unit tests.

**Table 4.3** Three flavors of unit tests: logic, integration, and functional

Test type	Description
Logic unit tests	Unit tests that focus on exercising the code logic. These tests are usually meant to exercise only a single method and no other. You can control the boundaries of a given method using mock objects or stubs (see chapters 6 and 7).
Integration unit tests	Unit tests that focus on testing the interaction between components in their real environment (or part of the real environment). For example, code that accesses a database has tests that effectively call the database, thus proving that the code-database interaction works (see chapter 11).
Functional unit tests	Unit tests that extend the boundaries of integration unit testing to confirm a stimulus-response. For example, imagine a web page that is protected and that you can access only after being logged in. If you are not logged in, accessing the page results in a redirect to the login page. A functional unit test verifies that behavior by sending an HTTP request to the page and verifying that the result is a 302 response code. It does not, however, verify that the full workflow leads to the login page. Workflow is the domain of pure, software functional testing (see section 4.2.1).

Figure 4.4 illustrates how these three flavors of unit tests interact. The sliders define the boundaries between the types of unit tests. The tests can be defined by the locations of the sliders. All three types of tests are needed to ensure your code works. If you use them, you can sleep well at night and come in the next day, well rested and eager to create more great code!

Strictly speaking, the functional unit tests are not pure unit tests, but neither are they pure functional tests. They are more dependent on an external environment than pure unit tests are, but they do not test a complete workflow, as



**Figure 4.4** Unit testing within the application life cycle

expected by pure functional tests. We put functional unit tests in our scope because they are often useful as part of the battery of tests run in development.

An example is the `StrutsTestCase` (<http://strutstestcase.sourceforge.net/>) framework, which provides functional unit testing of the runtime Struts configuration. These tests tell a developer that the controller is invoking the appropriate software action and forwarding to the expected presentation page, but they do not confirm that the page is actually present and renders correctly.

### 4.3 Determining how good tests are

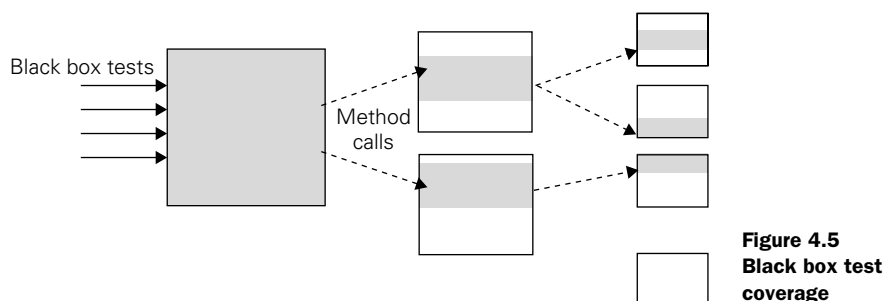
When you test a rivet, you can apply simple, objective standards. You can look at its dimensions, its weight, and whether it can withstand a certain amount of pressure. You can say that your tests cover just these aspects, and let the builder decide if the test coverage is sufficient.

But how do you express what aspects an application's unit tests cover? "Everything that could possibly fail" is a fine standard, but it's rather subjective. What kind of metrics can we apply?

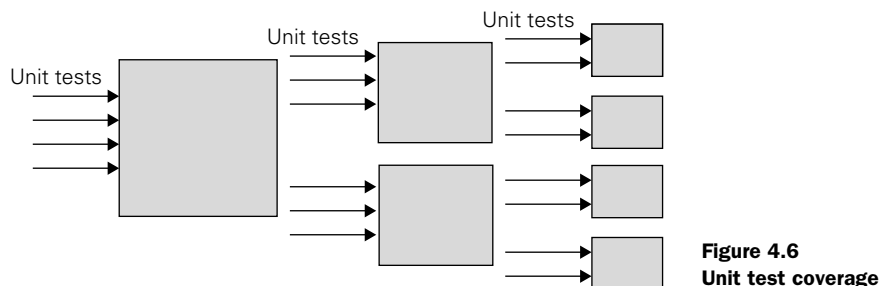
### 4.3.1 Measuring test coverage

One metric of test coverage would be to count how many methods are being called by your tests. This doesn't tell you whether the test is doing the right thing, but it does tell you whether you have a test in place.

Without unit tests, you can only write tests against the application's public API methods. Because you don't need to see inside the application to create the tests, these are called *black box tests*. Figure 4.5 diagrams what an application's test coverage might look like using only black box tests.



Unit tests can be written with an intimate knowledge of how a method is implemented. If a conditional branch exists within the method, you can write two unit tests: one for each branch. Because you need to see into the method to create such a test, this is called *white box testing*. Figure 4.6 shows what an application's test coverage might look like using white box unit tests.



With white box unit tests, it's easier to achieve a higher test coverage metric, mainly because you have access to more methods and because you can control both the inputs to each method and the behavior of secondary objects called (using stubs or mock objects, as you'll see in later chapters). White box unit tests can be written against both package-protected and public methods.

### 4.3.2 Generating test coverage reports

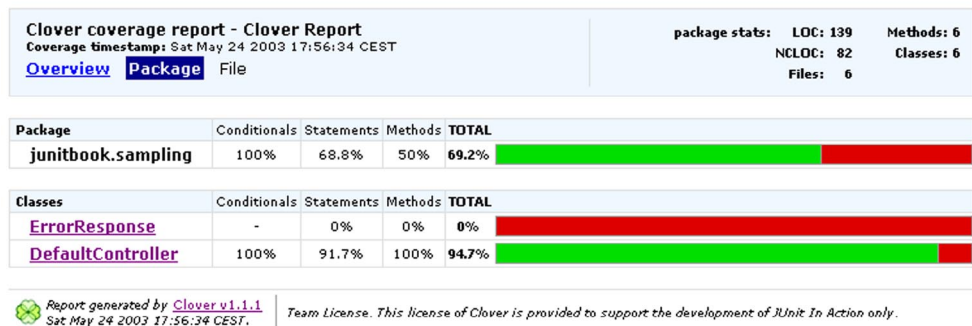
Tools are available for JUnit that can analyze your application and provide an exact report of your application's test coverage. Figure 4.7 shows one such report generated by the Clover tool (<http://www.thecortex.net/clover/>).<sup>2</sup> It is the result of applying Clover on the Controller sample from chapter 3.

Knowing what classes are tested (`DefaultController`) and what classes are not tested (`ErrorResponse`) is useful information. However, it's even better to know why the `DefaultController` has only 94.7% test coverage. Fortunately, Clover is able to drill down at the level of the method implementation, as shown in figure 4.8.

The report shown in figure 4.8 itemizes how many times a given line of source code has been executed by our tests (this is the number in the second column from the left—for example, the `getHandler` method at line 10 has been called three times). But more important, the report shows the lines of code that have *not* been tested. In our example, the case when an `Exception` is thrown by `processRequest` is not being tested (line 32).

Knowing what code is untested is good. However, a good test-coverage tool (such as Clover) should also provide historical reports showing the progression (or regression) of tests across development iterations and integration with your favorite build system (such as Ant). It should have the ability to stop the build if coverage criteria are not met—for example, if the coverage for such a package is below 70%.

On our own projects, we like to check the coverage percentage after a development iteration (say, every two weeks). We adjust the build failure criteria so that the next iteration must have at least the same coverage percentage as the previous iteration. This strategy helps to ensure that our test coverage steadily improves.



**Figure 4.7** A class coverage report generated by the popular Clover product

<sup>2</sup> Clover is a commercial application that is free for noncommercial activities (especially the open source community).

Source file	Conditionals	Statements	Methods	TOTAL
DefaultController.java	100%	91.7%	100%	94.7%

```

1 package junitbook.sampling;
2
3 import java.util.HashMap;
4
5 public class DefaultController implements Controller
6 {
7     private java.util.Map requestHandlers = new HashMap(); // (1)
8
9     // (2)
10 3 public RequestHandler getHandler(Request request)
11 {
12 3     if (!this.requestHandlers.containsKey(request.getName()))
13     {
14 1         String message = "Cannot find handler for request name " +
15             "[" + request.getName() + "]";
16 1         throw new RuntimeException(message); // (3)
17     }
18 2     return (RequestHandler)
19         this.requestHandlers.get(request.getName()); // (4)
20 }
21
22 // (5)
23 1 public Response processRequest(Request request)
24 {
25 1     Response response;
26 1     try
27     {
28 1         response =
29             getHandler(request).process(request); // (6)
30     } catch (Exception exception)
31     {
32 0         response = new ErrorResponse(request, exception); // (7)
33     }
34 1     return response; // (8)
35 }

```

**Figure 4.8** A Clover report showing how many times lines of code have been tested and what portion of the code has not been tested (line 32)

Although they're both helpful and interesting, the reports we have shown do not tell you how "good" your tests are. They only tell you which methods are being tested and which are not. Your tests could be faulty and not test anything at all, and the reports would still be the same! Ascertaining the quality of tests is difficult, and tools such as Jester (<http://jester.sourceforge.net>) can help. Jester works by performing random mutations to the code being tested; it then verifies if your tests still pass. If they do, it means they were not good enough. Jester does this over several iterations and then produces a report showing the quality of the tests.

What is important to remember is that *100% test coverage does not guarantee that your application is 100% tested*. Your test coverage is only as good as your tests! If your tests are poorly conceived, your application will be inadequately tested, no matter how many tests you have.

### 4.3.3 Testing Interactions

So, if we can achieve higher test coverage with white box unit tests, and we can generate some fancy reports to prove it, do we need to bother with black box tests at all?

If you think about the differences between figure 4.5 and figure 4.6, there's more going on than how many methods are being tested. The black box tests in figure 4.5 are testing the interactions between objects. The white box unit tests in figure 4.6, by definition, do not test object interactions. If a white box test does interact with another object, that object is usually a stub or a mock object (see chapters 6 and 7), designed to produce specific test behavior.

If you want to fully test your application, including how the runtime objects interact with each other, you need to include black box integration tests as part of your regimen. Each type of test has its own place in the scheme of things.

## 4.4 Test-Driven Development

---

In chapter 3, you designed an application controller and quickly wrote some tests to prove your design. As you wrote the tests, the tests helped improve the initial design. As you write more unit tests, positive reinforcement encourages you to write them earlier. Pretty soon, as you design the implementation, it becomes natural to wonder about how you will test a class. Following this methodology, more developers are making the quantum leap from test-friendly designs to Test-Driven Development.<sup>3</sup>

**DEFINITION** *Test-Driven Development (TDD)*—Test-Driven Development is a programming practice that instructs developers to write new code only if an automated test has failed, *and* to eliminate duplication. The goal of TDD is “clean code that works.”

### 4.4.1 Tweaking the cycle

When you develop code, you design an application programming interface (API) and then implement the behavior promised by the interface. When you unit-test code, you verify the promised behavior through a method's API. The test is a client of the method's API, just as your domain code is a client of the method's API.

---

<sup>3</sup> Kent Beck, *Test Driven Development: By Example* (Boston: Addison-Wesley, 2003).

The conventional development cycle goes something like this: [code, *test*, (repeat), commit]. Developers practicing TDD make a seemingly slight but surprisingly effective adjustment: [*test*, code, (repeat), commit]. (More on this later.) The test drives the design and becomes the method's first client.

Listing 4.3 illustrates how unit tests can help design the implementation. The `testGetBalanceOk` method shows that the `getBalance` method of `Account` returns the account balance as a long and that this balance can be set in the `Account` constructor. At this point, the implementation of `Account` is purely hypothetical, but writing the unit tests allows you to focus on the design of the code. As soon as you implement the class, you can run the test to prove that the implementation works. If the test fails, then you can continue working on the implementation until it passes the test. When the test passes, you know that your contract is fulfilled and that the code works as advertised.

#### Listing 4.3 Unit tests as a design guide

```
import junit.framework.TestCase;

public class TestAccount extends TestCase
{
    public void testGetBalanceOk ()
    {
        long balance = 1000;
        Account account = new Account(balance);
        long result = account.getBalance();
        assertEquals(balance, result);
    }
}
```

When you use the test as the method's first client, it becomes easier to focus purely on the API. Writing the tests first provides the following:

- Means to design the code
- Documentation as to how the code works
- Unit tests for the code (*waddyaknow*)

Someone new to the project can understand the system by studying the functional test suite (with the help of some high-level UML diagrams, for example). To analyze a specific portion of the application in detail, someone can drill down into individual unit tests.

#### 4.4.2 The TDD two-step

Earlier, we said that TDD tweaks the development cycle to go something like [test, code, (repeat), ship]. The problem with this chant is that it leaves out a key step. It should go more like this: [test, code, *refactor*, (repeat), ship].

The core tenets of TDD are to:

- 1 Write a failing automatic test before writing new code
- 2 Eliminate duplication<sup>4</sup>

The *eliminate duplication* step ensures that you write code that is not only testable but also *maintainable*. When you eliminate duplication, you tend to increase cohesion and decrease dependency. These are hallmarks of code that is easier to maintain over time.

Other coding practices have encouraged us to write maintainable code by anticipating change. In contrast, TDD encourages us to write maintainable code *by eliminating duplication*. Developers following this practice have found that test-backed, well-factored code is, by its very nature, *easy and safe* to change. TDD gives us the confidence to solve today's problems today and tomorrow's problems tomorrow. *Carpe diem!*

##### **JUnit best practice: test first (never write a line of new code without a failing test)**

If you take the TDD development pattern to heart, an interesting thing happens: Before you can write any code, *you must write a test that fails*. Why does it fail? *Because you haven't written the code to make it succeed.*

Faced with a situation like this, most of us begin by writing a simple implementation to let the test pass. Now that the test succeeds, you could stop and move on to the next problem. Being a professional, you would take a few minutes to *refactor* the implementation to remove redundancy, clarify intent, and optimize the investment in the new code. But as long as the test succeeds, technically, you're done.

The end game? If you always test first, you will never write a line of new code without a failing test.

---

<sup>4</sup> Ibid.

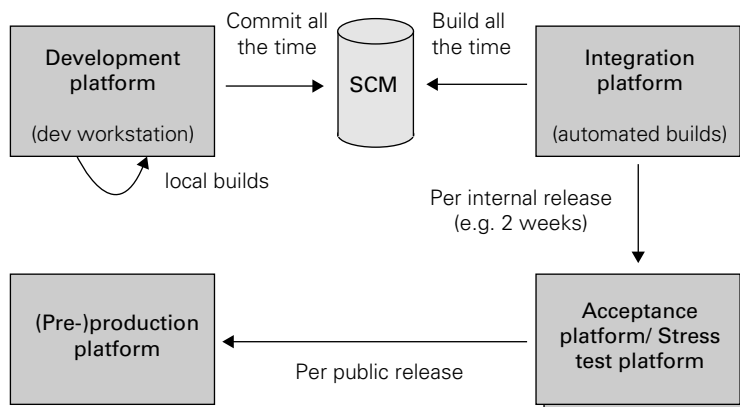


## 4.5 Testing in the development cycle

Testing occurs at different places and times during the development cycle. Let's first introduce a development life cycle and then use it as a base for deciding what types of tests are executed when. Figure 4.9 shows a typical development cycle we have used effectively in small to large teams.

The life cycle is divided into four or five platforms:

- *Development platform*—This is where the coding happens. It consists of developers' workstations. One important rule is usually to commit (or *check in*, depending on the terminology used) several times per day to your common Source Control Management (SCM) tool (CVS, ClearCase, Visual SourceSafe, Starteam, and so on). Once you commit, others can begin using what you have committed. However, it is important to only commit something that “works.” In order to know if it works, a typical strategy is to have an automated build (see chapter 5) and run it before each commit.
- *Integration platform*—The goal of this platform is to build the application from its different pieces (which may have been developed by different teams) and ensure that they all fit together. This step is extremely valuable, because problems are often discovered here. It is so valuable that we want to automate it. It is then called *continuous integration* (see <http://www.martinfowler.com/articles/continuousIntegration.html>) and can be achieved by automatically building the application as part of the build process (more on that in chapter 5 and later).
- *Acceptance platform / stress test platform*—Depending on how rich your project is, this can be one or two platforms. The stress test platform exercises the



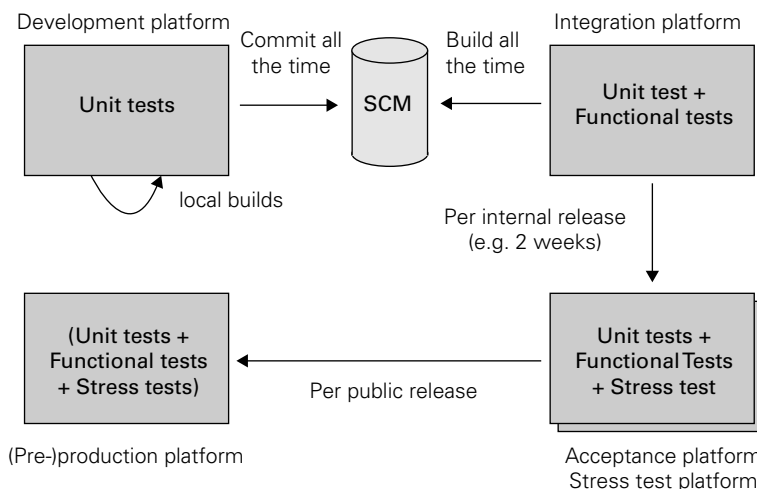
**Figure 4.9**  
A typical application development life cycle using the continuous integration principle

application under load and verifies that it scales correctly (with respect to size and response time). The acceptance platform is where the project's customers accept (sign off on) the system. It is highly recommended that the system be deployed on the acceptance platform as often as possible in order to get user feedback.

- *(Pre-)production platform*—The pre-production platform is the last staging area before production. It is optional, and small or noncritical projects can do without it.

Let's see how testing fits in the development cycle. Figure 4.10 highlights the different types of tests you can perform on each platform:

- On the *development platform*, you execute logic unit tests (tests that can be executed in isolation from the environment). These tests execute very quickly, and you usually execute them from your IDE to verify that any change you have brought to the code has not broken anything. They are also executed by your automated build before you commit the code to your SCM. You could also execute integration unit tests; however, they often take much longer, because they need some part of the environment to be set up (database, application server, and so on). In practice, you would execute only a subset of all integration unit tests, including any new integration unit tests you have written.
- The *integration platform* usually runs the build process automatically to package and deploy the application and then executes unit and functional tests.



**Figure 4.10**  
The different types  
of tests performed  
on each platform of  
the development  
cycle

Usually, only a subset of all functional tests is run on the integration platform, because compared to the target production platform it is a simple platform that lack elements (for example, it may be missing a connection to an external system being accessed). All types of unit tests are executed on the integration platform (logic unit tests, integration unit tests, and functional unit tests). Time is less important, and the whole build can take several hours with no impact on development.

- On the *acceptance platform / stress test platform*, you re-execute the same tests executed by the integration platform; in addition, you run stress tests (performance and load tests). The acceptance platform is extremely close to the production platform, and more functional tests can also be executed.
- It is always a good habit to try to run on the *(pre-)production platform* the tests you ran on the acceptance platform. Doing so acts as a sanity check to verify that everything is set up correctly.

### **JUnit best practice: continuous regression testing**

Most tests are written for the here and now. You write a new feature, you write a new test. You see if the feature plays well with others, and if the users like it. If everyone is happy, you can lock the feature and move on to the next item on your list. Most software is written in a progressive fashion: You add one feature and then another.

Most often, each new feature is built over a path paved by existing features. If an existing method can service a new feature, you reuse the method and save the cost of writing a new one. Of course, it's never quite that easy. Sometimes you need to change an existing method to make it work with a new feature. When this happens, you need to confirm that all the old features still work with the amended method.

A strong benefit of JUnit is that the test cases are easy to automate. When a change is made to a method, you can run the test for that method. If that one passes, then you can run the rest. If any fail, you can change the code (or the tests) until everyone is happy again.

Using old tests to guard against new changes is a form of regression testing. Any kind of test can be used as a regression test, but running unit tests after every change is your first, best line of defense.

The best way to ensure that regression testing is done is to automate your test suites. See chapter 5 for more about automating JUnit.

## 4.6 Summary

---

The pace of change is increasing. Project time frames are getting shorter, and we need to react quickly to change. In addition, the development process is shifting—development as the art of writing code is not enough. Development must be the art of writing solutions.

To accommodate rapid change, we must break with asynchronous approaches where software testing is done after development by a separate team. Late testing does not scale when change and swiftness are paramount.

The agile approaches favor working in small vertical slices rather than big horizontal ones. This means small teams performing several activities at once (designing, testing, coding) and delivering solutions, slice by slice. Automated tests are hallmarks of applications that work. Tests enable refactoring, and refactoring enables the elegant addition of new solutions, slice by slice.

When it comes to unit-testing an application, you can use several types of tests: logic unit tests, integration unit tests, and functional unit tests. All are useful during development, and they complement each other. They also complement the other software tests that should be performed by quality assurance personnel and by the customer.

One of the great benefits of unit tests is that they are easy to automate. In the next chapter, we look at several tools to help you automate unit testing.

# 5

## *Automating JUnit*

---

### ***This chapter covers***

- Integrating JUnit into your development environment
- Running JUnit from Ant, Maven, and Eclipse

*It's supposed to be automatic, but you still have to press the button.*

—John Brunner

In this chapter, we will look at three products with direct support for JUnit: Ant, Maven, and Eclipse. Ant and Maven are build tools that can be used with any Java programming environment. Eclipse is an integrated development environment (IDE). We will demonstrate how you can be productive with JUnit and these environments and how to automate running JUnit tests.

At the end of this chapter, you will know how to set up your environment on your machine to build Java projects, including execution of JUnit tests and generation of JUnit reports.

## 5.1 A day in the life

---

For unit tests to be effective, they should be part of the development routine. Most development cycles begin by checking out a module from the project's source-code repository. Before making any changes, prudent developers first run the full unit-test suite. Many teams have a rule that all the unit tests on the working repository must pass. Before starting any development of your own, you should see for yourself that no one has broken the all-green rule. You should always be sure that your work progresses from a known baseline.<sup>6</sup>

The next step is to write the code for a new use case (or modify an existing one). If you are a Test-Driven Development (TDD) practitioner, you'll start by writing new tests for the use case. (For more about TDD, see chapter 4.) Generally, the test will show that your use case isn't supported and either will not compile or will display a red bar when executed. Once you write the code to implement the use case, the bar turns green, and you can check in your code. Non-TDD practitioners will implement the use case and then write the tests to prove it. Once the bar turns green, the code and the tests can be checked in.

In any event, before you move on to code the next feature, you should have a test to prove the new feature works. After you code the next feature, you can run the tests for the prior feature too. In this way, you can ensure that new development does not break old development. If the old feature needs to change to accommodate the new feature, then you update its test and make the change.

If you test rigorously, both to help you design new code (TDD) and to ensure that old code works with new (regression testing), you must continually run the unit tests as a normal part of the development cycle. The test runners must become your best friends. And, like any best friend, the test runners should be on

speed dial. You need to be able to run the tests automatically and effortlessly throughout the day.

**DEFINITION** *regression tests*—When new code is added to existing code, regression tests verify that the existing code continues to work correctly.<sup>1</sup>

In chapter 1, section 1.3, we discussed running JUnit from the command line. Running a single JUnit test case against a single class is not difficult. But it is not a practical approach for running continuous tests against a project with hundreds or even thousands of classes.

A project that is fully tested has at least as many test classes as production classes. Developers can't be expected to run an entire set of regression tests every day by hand. So, you must have a way to run key tests easily and automatically, without relying on already-overworked human beings.

Because you are writing so many tests, you need to write and run tests in the most effective way possible. Using JUnit should be seamless, like calling a build tool or plugging in a code highlighter.

Three tools that many developers already use are Ant, Maven, and Eclipse (or any other IDE). Ant is the de facto standard tool for building Java applications; it is an excellent tool for managing and automating JUnit tests. Maven extends Ant's features to provide broader project-management support. Like Ant, it is on its way to becoming a de facto standard.

Ant and Maven will happily build your applications, but they don't help write them. Although many Java applications are still written with text editors, more and more developers use full-featured IDEs. Several very competent IDEs are now available, both as open source and as retail products. We'll look at how one of these products, Eclipse, integrates JUnit into an omnibus development platform.

## 5.2 Running tests from Ant

---

Compiling and testing a single class, like the `DefaultController` class from chapter 3, is not difficult. Compiling a larger project with multiple classes can be a huge headache if your only tool is the stock `javac` compiler. Increasing numbers of classes refer to each other, and so more classes need to be on the classpath where the compiler can find them. On any one build, only a few classes will change, so

---

<sup>1</sup> Derek Sisson, "Types of Tests": <http://www.philosophe.com/testing/tests.html>.

there is also the issue of which classes to build. Re-running your JUnit tests by hand after each build can be equally inconvenient, for all the same reasons.

Happily, the answer to both problems is the fabulous tool called Ant. Ant is not only an essential tool for building applications, but also a great way to run your JUnit regression tests.

### 5.2.1 Ant, indispensable Ant

Apache's Ant product (<http://ant.apache.org/>) is a build tool that lets you easily compile and test applications (among other things). It is the de facto standard for building Java applications. One reason for Ant's popularity is that it is more than a tool: Ant is a framework for running tools. In addition to using Ant to configure and launch a Java compiler, you can use it to generate code, invoke JDBC queries, and, as you will see, run JUnit test suites.

Like many modern projects, Ant is configured through an XML document. This document is referred to as the *buildfile* and is named `build.xml` by default. The Ant buildfile describes each task that you want to apply on your project. A task might be compiling Java source code, generating Javadocs, transferring files, querying databases, or running tests. A buildfile can have several *targets*, or entry points, so that you can run a single task or chain several together. Let's look at using Ant to automatically run tests as part of the build process. If (*gasp!*) you don't have Ant installed, see the following sidebars. For full details, consult the Ant manual (<http://ant.apache.org/manual/>).

#### Installing Ant on Windows

To install Ant on Windows, follow these steps:

- 1 Unzip the Zip distribution file to a directory on your computer system (for example, `C:\Ant`).
- 2 Under this directory, Unzip creates a subdirectory for the Ant distribution you downloaded—for example, `C:\Ant\jakarta-ant-1.5.3`. Add an `ANT_HOME` variable to your environment with this directory as the value. For example:

```
Variable Name: ANT_HOME
Variable Value: C:\Ant\jakarta-ant-1.5.3
```

- 3 Edit your system's `PATH` environment variable to include the `ANT_HOME\bin` folder:

```
Variable Name: PATH
Variable Value: %ANT_HOME%\bin;...
```



**Installing Ant on Windows** (continued)

- 4 We recommend that you also specify the location of your Java Developer's Kit (JDK) as the `JAVA_HOME` environment variable:

```
Variable Name: JAVA_HOME  
Variable Value: C:\j2sdk1.4.2
```

This value, like the others, may vary depending on where you installed the JDK on your system.

- 5 To enable Ant's JUnit task, you must put `junit.jar` in the `ANT_HOME\lib` folder.

**Installing Ant on UNIX (bash)**

To install Ant on UNIX (or Linux), follow these steps:

- 1 Untar the Ant tarball to a directory on your computer system (for example, `/opt/ant`).
- 2 Under this directory, tar creates a subdirectory for the Ant distribution you downloaded—for example, `/opt/ant/jakarta-ant-1.5.3`. Add this subdirectory to your environment as `ANT_HOME`. For example:

```
export ANT_HOME=/opt/ant/jakarta-ant-1.5.3
```

- 3 Add the `ANT_HOME/bin` folder to your system's command path:

```
export PATH=${PATH}:${ANT_HOME}/bin
```

- 4 We recommend that you also specify the location of your JDK as the `JAVA_HOME` environment variable:

```
export JAVA_HOME=/usr/java/j2sdk1.4.2
```

- 5 To enable Ant's JUnit task, you must put `junit.jar` in the `ANT_HOME/lib` folder.

**5.2.2 Ant targets, projects, properties, and tasks**

When you build a software project, you are often interested in more than just binary code. For a final distribution, you may want to generate Javadocs along with the binary classes. For an interim compile during development, you may skip that step. Sometimes, you want to run a clean build from scratch. Other times, you want to build the classes that have changed.

To help you manage the build process, Ant lets you create a buildfile for each of your projects. The buildfile may have several targets, encapsulating the different

tasks needed to create your application and related resources. To make the buildfiles easier to configure and reuse, Ant lets you define dynamic property elements. These Ant essentials are as follows:

- *Buildfile*—Each buildfile is usually associated with a particular development project. Ant uses the `project` XML tag as the outermost element in `build.xml`. The `project` element defines a project. It also lets you specify a default target, so you can run Ant without any parameters.
- *Target*—When you run Ant, you can specify one or more targets for it to build. Targets can also declare that they depend on other targets. If you ask Ant to run one target, the buildfile might run several others first. This lets you create a distribution target that depends on other targets like `clean`, `compile`, `javadoc`, and `war`.
- *Property elements*—Many of the targets within a project will share the same settings. Ant lets you create property elements to encapsulate specific settings and reuse them throughout your buildfile. If a buildfile is carefully written, the property elements can make it easy to adapt the buildfile to a new environment. To refer to a property within a buildfile, you place the property within a special notation: `${property}`. To refer to the property named `target.dir`, you would write `${target.dir}`.

As mentioned, Ant is not so much a tool as a framework for running tools. You can use property elements to set the parameters a tool needs and a *task* to run the tool. A great number of tasks come bundled with Ant, and you can also write your own. For more about developing with Ant, we highly recommend *Java Development with Ant*.<sup>2</sup>

Listing 5.1 shows the top of the buildfile for the `sampling` project from chapter 3. This segment of the buildfile sets the default target and the properties your tasks will use.

**Listing 5.1** The Ant buildfile `project` and `property` elements

```
<project name="sampling" default="test"> ❶
  <property file="build.properties"/> ❷
  <property name="src.dir" location="src"/>
  <property name="src.java.dir" location="${src.dir}/java"/> ❸
```

<sup>2</sup> Erik Hatcher and Steve Loughran, *Java Development with Ant* (Greenwich, CT: Manning, 2003); <http://www.manning.com/hatcher/>.

```
<property name="src.test.dir" location="${src.dir}/test"/>

<property name="target.dir" location="target"/>
<property name="target.classes.java.dir"
  location="${target.dir}/classes/java"/>
<property name="target.classes.test.dir"
  location="${target.dir}/classes/test"/>

[...]
```

4

- ❶ Give the project the name `sampling` and set the default target to `test`. (The test target appears in listing 5.3.)
- ❷ You include a `build.properties` file. This file contains Ant properties that may need to be changed on a user's system because they depend on the executing environment. For example, these properties can include the locations of redistributable jars. Because programmers may store jars in different locations, it is good practice to use a `build.properties` file to define them. Many open source projects provide a `build.properties.sample` file you can copy as `build.properties` and then edit to match your environment. For this project, you won't need to define any properties in it.
- ❸ ❹ As you will see, your targets need to know the location of your production and test source code. You use the Ant `property` task to define these values so that they can be reused and easily changed. At ❸, you define properties related to the source tree; at ❹, you define those related to the output tree (where the build-generated files will go). Notice that you use different properties to define where the compiled production and tests classes will be put. Putting them in different directories is a good practice because it allows you to easily package the production classes in a jar without mixing test classes.

An interesting thing about Ant properties is that they are *immutable*—once they are set, they cannot be modified. For example, if any properties are redefined after the `build.properties` file is loaded, the new value is ignored. The first definition always wins.

### 5.2.3 The *javac* task

For simple jobs, running the Java Compiler (`javac`) from the command line is easy enough. But for multipackage products, the care and feeding of `javac` and your classpath becomes a Herculean task. Ant's `javac` task tames the compiler and its classpath, making building projects effortless and automatic.

The Ant `javac` task is usually employed from within a target with a name like `compile`. Before and after running the `javac` task, you can perform any needed file management as part of the target. The `javac` task lets you set any of the standard options, including the destination directory. You can also supply a list of paths for your source files. The latter is handy for projects with tests, because you may tend to keep production classes in one folder and test classes in another.

Listing 5.2 shows the `compile` targets that call the Java Compiler for the sampling project, both for the production code and for the test code.

**Listing 5.2** The buildfile compile targets

```
<target name="compile.java">
  <mkdir dir="${target.classes.java.dir}"/>
  <javac destdir="${target.classes.java.dir}">
    <src path="${src.java.dir}"/>
  </javac>
</target>

<target name="compile.test" depends="compile.java">
  <mkdir dir="${target.classes.test.dir}"/>
  <javac destdir="${target.classes.test.dir}">
    <src path="${src.test.dir}"/>
    <classpath>
      <pathelement location="${target.classes.java.dir}"/>
    </classpath>
  </javac>
</target>

<target name="compile" depends="compile.java,compile.test"/>
```

- ❶ Declare the target to compile the java production sources, naming it `compile.java`.
- ❷ Ensure that the directory where you will generate your production class files exists. Ant resolves the property you set at the top of the buildfile (see listing 5.1) and inserts it in place of the variable notation `${target.classes.java.dir}`. If the directory already exists, Ant quietly continues.
- ❸ Call the Java Compiler (`javac`) and pass it the destination directory to use.
- ❹ Tell the `javac` task what sources to compile.
- ❺ Compile the test sources exactly the same way you just did for the production sources. Your `compile.test` target has a dependency on the `compile.java` target, so you must add a `depends` element to that `compile.test` target definition (`depends="compile.java"`). You may have noticed that you don't explicitly add the JUnit jar to the classpath. Remember that when you installed Ant, you put the JUnit

jar in `ANT_HOME/lib` (this is necessary in order to use the `junit` Ant task). As a consequence, `junit.jar` is already on your classpath, and you don't need to specify it in the `javac` task to properly compile your tests.

- 6 You need to add a nested `classpath` element in order to add the production classes you just compiled to the classpath. This is because test classes call production classes.
- 7 Create a `compile` target that automatically calls the `compile.java` and `compile.test` targets.

### 5.2.4 The JUnit task

In chapter 3, you ran the `DefaultController` tests by hand. That meant between any changes, you had to

- Compile the source code
- Run the `TestDefaultController` test case against the compiled classes

You can get Ant to perform both these steps as part of the same build target. Listing 5.3 shows the test target for the `sampling` buildfile.

**Listing 5.3** The buildfile test target

```

<target name="test" depends="compile">
  <junit printsummary="yes" haltonerror="yes" haltonfailure="yes"
    fork="yes">
    <formatter type="plain" usefile="false"/>
    <test name="junitbook.sampling.TestDefaultController"/>
    <classpath>
      <pathelement location="${target.classes.java.dir}"/>
      <pathelement location="${target.classes.test.dir}"/>
    </classpath>
  </junit>
</target>
</project>

```

- 1 Give the target a name and declare that it relies on the `compile` target. If you ask Ant to run the test target, it will run the `compile` target before running test.
- 2 Here you get into the JUnit-specific attributes. The `printsummary` attribute says to render a one-line summary at the end of the test. By setting `fork` to `yes`, you force Ant to use a separate Java Virtual Machine (JVM) for each test. This is always a good practice as it avoids interferences between test cases. The `haltonfailure` and `haltonerror` attributes say that the build should stop if any test returns a

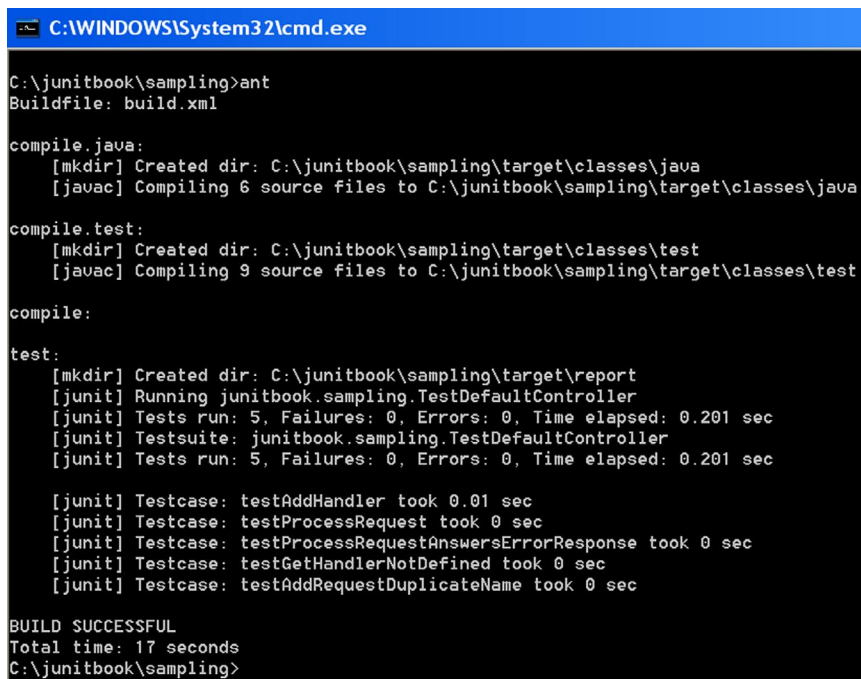
failure or an error (an error is an unexpected error, whereas a failure happens if one of the test asserts does not pass).

- ❸ Configure the `junit` task formatter to use plain text and output the test result to the console.
- ❹ Provide the class name of the test you want to run.
- ❺ Extend the classpath to use for this task to include the classes you just compiled.

### 5.2.5 Putting Ant to the task

Now that you've assembled the buildfile, you can run it from the command line by changing to your project directory and entering `ant`. Figure 5.1 shows what Ant renders in response.

You can now build and test the `sampling` project all at the same time. If any of the tests fail, the `haltonfailure/haltonerror` settings will stop the build, bringing the failure to your attention.



```
C:\WINDOWS\System32\cmd.exe

C:\junitbook\sampling>ant
Buildfile: build.xml

compile.java:
[mkdir] Created dir: C:\junitbook\sampling\target\classes\java
[javac] Compiling 6 source files to C:\junitbook\sampling\target\classes\java

compile.test:
[mkdir] Created dir: C:\junitbook\sampling\target\classes\test
[javac] Compiling 9 source files to C:\junitbook\sampling\target\classes\test

compile:

test:
[mkdir] Created dir: C:\junitbook\sampling\target\report
[junit] Running junitbook.sampling.TestDefaultController
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0.201 sec
[junit] Testsuite: junitbook.sampling.TestDefaultController
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0.201 sec

[junit] Testcase: testAddHandler took 0.01 sec
[junit] Testcase: testProcessRequest took 0 sec
[junit] Testcase: testProcessRequestAnswersErrorResponse took 0 sec
[junit] Testcase: testGetHandlerNotDefined took 0 sec
[junit] Testcase: testAddRequestDuplicateName took 0 sec

BUILD SUCCESSFUL
Total time: 17 seconds
C:\junitbook\sampling>
```

Figure 5.1 Running the buildfile from the command line

**Running optional tasks**

The `junit` task is one of several components bundled in Ant's `optional.jar`. The `optional.jar` file should already be in your `ANT_HOME/lib` directory. Ant does not bundle a copy of JUnit, so you must be sure that `junit.jar` is on your system classpath or in the `ANT_HOME/lib` directory. (The `optional.jar` file is for tasks that depend on another package, like JUnit.) For more about installing Ant, see the sidebars on pages 89–90. If you have any trouble running the Ant buildfiles presented in this chapter, make sure the Ant `optional.jar` is in the `ANT_HOME/lib` folder and `junit.jar` is either on your system classpath or also in the `ANT_HOME/lib` folder.

**The empty classpath**

Given a tool like Ant, many developers don't bother with a system classpath anymore: You can let Ant take care of all that. Ant's `classpath` element makes it easy to build the classpath you need when you need it.

The only blind spot is the jars you need in order to run one of the optional Ant tasks, like `junit`. To provide the flexibility you need for other circumstances, Ant uses Sun's delegation model to create whatever classpath you need at runtime. In the case of the optional tasks, there's a bootstrap issue. To employ a task, whatever libraries a task needs must be on the *same* classpath as the code for the task. This means you need to load `junit.jar` in the same place you load `optional.jar`. Meanwhile, you also need to load the task (and any external libraries) before you can use the task in your buildfile. In short, you can't specify the path to `junit.jar` as part of the `junit` task.

The simplest solution is to move `junit.jar` to `ANT_HOME/lib`. There are alternative configurations, but they are usually more trouble than they are worth.<sup>3</sup>

So, to keep a clean classpath and use optional tasks like JUnit, you should move the jar for the external library to `ANT_HOME/lib`. Ant will then automatically load `optional.jar` and the external libraries together, enabling use of the optional tasks in your buildfiles. Just remember to update `junit.jar` in `ANT_HOME/lib` whenever you install a new version of either Ant or JUnit.

**5.2.6 Pretty printing with JUnitReport**

A report like the one in figure 5.1 is fine when you are running tests interactively. But what if you want to run a test suite and review the results later? For

---

<sup>3</sup> Ant 1.6 will let you put optional task jars in places other than `ANT_HOME/lib`.

Unit Test Results

Designed for use with [JUnit](#) and [Ant](#).

**Class**  
**junitbook.sampling.TestDefaultController**

Name	Tests	Errors	Failures	Time(s)
TestDefaultController	5	0	0	0.541

**Tests**

Name	Status	Type	Time(s)
testMethod	Success		0.010
testAddHandler	Success		0.000
testProcessRequest	Success		0.000
testGetHandlerNotDefined	Success		0.000
testAddRequestDuplicateName	Success		0.000

[Properties »](#)

**Figure 5.2** Output of the `junitreport` Ant task

example, the tests might be run automatically every day by a cron job (whether you liked it or not!).

Another optional Ant task, `junitreport`, is designed to output the result of the tests as XML. To finish the job, `junitreport` renders the XML into HTML using an XSL stylesheet. The result is an attractive and functional **report** that you (or your boss) can peruse with any browser. A `junitreport` page for the `sampling` project is shown in figure 5.2.

Listing 5.4 shows the changes (in bold) necessary in the buildfile to generate this report. To execute the script, type **`ant report`** on the command line in the `sampling` project.

#### Listing 5.4 Adding a `JUnitReport` task to the buildfile

```
<project name="sampling" default="test">
[...]
```

**<property name="target.report.dir"**

location="\${target.dir}/report"/>

[...]

**<target name="test" depends="compile">**

**<mkdir dir="\${target.report.dir}"/>**

**<junit printsummary="yes" haltonerror="yes" haltonfailure="yes"**

fork="yes">

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

**<formatter type="xml"/>**

**<test name="junitbook.sampling.TestDefaultController"**



```

        todir="${target.report.dir}"/> ❹
    <classpath>
        <pathelement location="${target.classes.java.dir}"/>
        <pathelement location="${target.classes.test.dir}"/>
    </classpath>
</junit>
</target>

<target name="report" depends="test"> ❺
    <mkdir dir="${target.report.dir}/html"/> ❻
    <junitreport todir="${target.report.dir}"> ❼
        <fileset dir="${target.report.dir}"> ❸
            <include name="TEST-*.xml"/>
        </fileset>
        <report todir="${target.report.dir}/html"/> ❹
    </junitreport>
</target>
</project>

```

- ❶ Define a property holding the target location where your reports will be generated.
- ❷ Create that directory.
- ❸ You need to modify the junit task so that it outputs the test results as XML. The junitreport task works by transforming the XML test result into an HTML report.
- ❹ Tell the junit task to create a report file in the \${target.report.dir} directory.
- ❺ Introduce a new report target that generates the HTML report.
- ❻ You begin by creating the directory where the HTML will be generated.
- ❼ Call the junitreport task to create the report.
- ❸ The junitreport task works by scanning the list of XML test results you specify as an Ant fileset.
- ❹ Tell the junitreport task where to generate the HTML report.

### 5.2.7 Automatically finding the tests to run

The buildfile you have written is using the `test` element of the `junit` task to tell JUnit what test to execute. Although this is fine when there are only a few test cases, it becomes tiresome when your test suite grows. The biggest issue then becomes ensuring that you haven't forgotten to include a test in the buildfile. Fortunately, the `junit` task has a convenient `batchtest` element that lets you specify test cases using wildcards. Listing 5.5 shows how to use it (changes from listing 5.4 are shown in bold).

## Listing 5.5 A better buildfile using batchtest

```

<project name="sampling" default="test">
[...]
```

1

2

```

    <target name="test" depends="compile">
      <mkdir dir="${target.report.dir}"/>
      <property name="tests" value="Test*" />
      <junit printsummary="yes" haltonerror="yes" haltonfailure="yes"
        fork="yes">
        <formatter type="plain" usefile="false"/>
        <formatter type="xml"/>
        <batchtest todir="${target.report.dir}">
          <fileset dir="${src.test.dir}">
            <include name="**/${tests}.java"/>
            <exclude name="**/Test*All.java"/>
          </fileset>
        </batchtest>
        <classpath>
          <pathelement location="${target.classes.java.dir}"/>
          <pathelement location="${target.classes.test.dir}"/>
        </classpath>
      </junit>
    </target>
[...]
```

3

```

    <target name="clean">
      <delete dir="${target.dir}"/>
    </target>
  </project>

```

- ❶ You may wonder why you define a property here when you could have put the wildcards directly into the `fileset` element at ❷. Using this trick, you can define the `tests` property on the command line and run a single test (or a specific set of tests) instead. This is an easy way to run a test against the class you are working on right now. Of course, once it's working, you still run the full test suite to be sure everyone is on the same page. Here is an example that only executes the `TestDefaultController` test case:

```
ant -Dtests=TestDefaultController test
```

- ❷ You improve the buildfile by making the `test` target more flexible. Whereas before you had to explicitly name the different tests you wanted to execute, here you leverage the `junit` task's nested `batchtest` element. With `batchtest`, you can specify the test to run as a *fileset*, thus allowing the use of wildcards.
- ❸ Add the always-useful `clean` target to remove all build-generated files. Doing so lets you start with a fresh build with no side effects from obsolete classes. Typically, a `dist` target that generates the project distributable depends on the `clean` target.

**Are automated unit tests a panacea?**

Absolutely not! Automated tests can find a significant number of bugs, but manual testing is still required to find as many bugs as possible. In general, automated regression tests catch 15–30% of all bugs found; manual testing finds the other 70–85% (<http://www.testingcraft.com/regression-test-bugs.html>).

**Are you sure about that?**

Some test-first design / unit testing enthusiasts are now reporting remarkably low numbers of bug counts, on the order of one or two per month or fewer. But these results need to be substantiated by formal studies and replicated by other teams. Your mileage will definitely vary.

### 5.3 Running tests from Maven

---

Once you have used Ant on several projects, you'll notice that most projects almost always need the same Ant scripts (or at least a good percentage). These scripts are easy enough to reuse through cut and paste, but each new project requires a bit of fussing to get the Ant buildfiles working just right. In addition, each project usually ends up having several subprojects, each of which requires you to create and maintain an Ant buildfile.

Maven (<http://maven.apache.org/>) picks up where Ant leaves off, making it a natural fit for many teams. Like Ant, Maven is a tool for running other tools, but Maven is designed to take tool reuse to the next level. If Ant is a source-building framework, Maven is a source-building *environment*.

#### 5.3.1 Maven the goal-seeker

Behind each target in every Ant buildfile lies a goal. The goal might be to generate the unit tests, to assemble the Javadocs, or to compile the distribution. The driving force behind Maven is that under the hood, each software project almost always does things the same way, following several years of best practices. Most differences are arbitrary, such as whether you call the target output directory `target` or `output`.

Instead of asking developers to write their own targets with tasks, Maven provides ready-to-use plugins to achieve the goals. At the time of this writing, Maven boasts more than 70 plugins. Once Maven is installed (see the sidebar on the next page), you can type `maven -g` to get the full list of the available plugins and goals. Reference documentation for the plugins is available at <http://maven.apache.org/reference/plugins>. The following list describes a few common Maven plugins:

- `jar`—Generates a project jar and deploys it to a local or remote jar repository
- `junit`—Executes JUnit tests
- `site`—Generates a project documentation web site that contains lots of useful reports and project information, in addition to containing any docs you wish to include
- `changelog`—Generates a change log report (CVS changelog, Starteam changelog, and so forth)
- `checkstyle`—Runs Checkstyle on the source code and generates a report
- `clover`—Runs Clover on the source code and generates a Clover report
- `eclipse`—Automatically generates Eclipse project files from the Maven project description
- `ear`—Packages the application as an ear file
- `cactus`—Automatically packages your code, deploys it in a container of your choice, starts the container, and runs Cactus tests (see chapter 8)
- `jboss`—Supports creation of JBoss Server configurations and deployments of war, ear, and EJB-jar in JBoss using a simple copy or JMX

Having well-defined plugins not only provides unprecedented ease of use, it also standardizes project builds, making it easy for developers to go from project to project.

### **Installing Maven**

Installing Maven is a three-step process:

- 1 Download the latest distribution from <http://maven.apache.org/builds/release/> and unzip/untar it in the directory of your choice (for example, `c:\maven` on Windows or `/opt/maven` on UNIX).
- 2 Define a `MAVEN_HOME` environment variable pointing to where you have installed Maven.
- 3 Add `MAVEN_HOME\bin` (`MAVEN_HOME/bin` on UNIX) to your `PATH` environment variable so that you can type **maven** from any directory.

You are now ready to use Maven. The first time you execute a plugin, make sure your Internet connection is on, because Maven will automatically download from the Web all the third-party jars the plugin requires.

### 5.3.2 Configuring Maven for a project

Using Ant alone, you describe your build at the level of the tasks. With Maven, you describe your project structure and the plugins use this directory structure, so you don't have to be an Ant wizard to set up your project. Maven handles the wizardry.

Let's look at a Maven description for a simple project based on the `sampling` project you wrote in chapter 3 and that you ran with Ant earlier in the chapter. The goal is to run your unit tests with Maven.

Configuring Maven for a project requires writing only one file: `project.xml` (also called the POM, short for project object model). It contains the full project description. Listing 5.6 shows the first part of this file, which contains background information about the project.

**Listing 5.6** First part of `project.xml` showing background project information

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project>
  <pomVersion>3</pomVersion>
  <id>junitbook-sampling</id>
  <name>JUnit in Action - Sampling JUnit</name>
  <currentVersion>1.0</currentVersion>
  <organization>
    <name>Manning Publications Co.</name>
    <url>http://www.manning.com/</url>
    <logo>http://www.manning.com/front/dance.gif</logo>
  </organization>
  <inceptionYear>2002</inceptionYear>
  <package>junitbook.sampling</package>
  <logo>/images/jia.jpg</logo>

  <description>
    Chapter 3 presents a sophisticated test case to show how JUnit
    works with larger components. The subject of our case study is
    a component found in many applications: a controller. We
    introduce the case-study code, identify what code to test, and
    then show how to test it. Once we know that the code works as
    expected, we create tests for exceptional conditions, to be
    sure our code behaves well even when things go wrong.
  </description>
  <shortDescription>
    Chapter 3 of JUnit in Action: Sampling JUnit
  </shortDescription>

  <url>http://sourceforge.net/projects/junitbook/</url>

  <developers>
    <developer>
      <name>Vincent Massol</name>
```

1  
2  
3  
4

5

6



```

        <id>vmassol</id>
        <email>vmassol@users.sourceforge.net</email>
        <organization>Pivolis</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
    <developer>
        <name>Ted Husted</name>
        <id>thusted</id>
        <email>thusted@users.sourceforge.net</email>
        <organization>Husted dot Com</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
</developers>
[...]
```



- ❶ Tell Maven the version of the POM you are using to describe the project. As of this writing, the version to use is 3. Maven uses it to perform automatic migration of old POM versions to the new one if need be.
- ❷ Define the project ID. Several plugins use this ID to name files that are generated. For example, if you run the `jar` plugin on the project, it generates a jar named `junitbook-sampling-1.0.jar` (`<id>.<currentVersion>.jar`).
- ❸ Give a human-readable name for your project. It is used, for example, by the `site` plugin, which generates the documentation web site.
- ❹ This is the current version of your project. For example, the version suffixed to the jar name comes from the definition here.
- ❺ Describe background information about your project that is used by the `site` plugin for the web site.
- ❻ Describe the developers working on this project and their roles. This information is used in a report generated by the `site` plugin.

### Executing Maven web-site generation

Let's use the Maven `site` plugin to generate the web site and see how the information you have provided is used. Open a command-line prompt in the `sampling/project` directory (see chapter 3, section 3.4 for details of setting up the project directory structure) and enter **maven site**, as shown in figure 5.3.



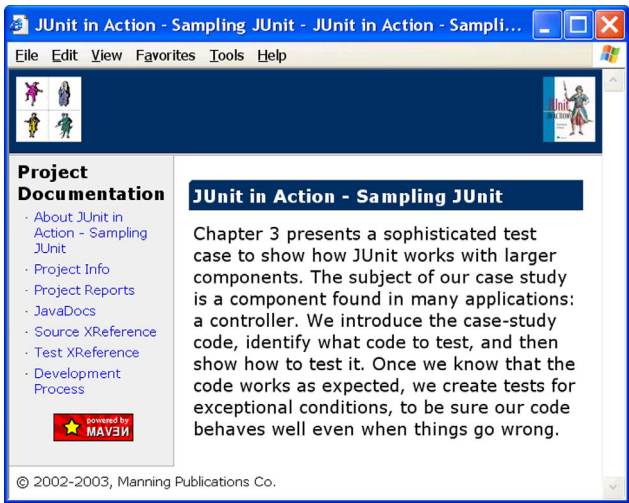


Figure 5.4 Welcome page of the generated site showing how the information entered in `project.xml` is used

<ul style="list-style-type: none"><li>Project Reports<ul style="list-style-type: none"><li>Metrics</li><li>Checkstyle</li><li>Change Log</li><li>Developer Activity</li><li>File Activity</li><li>Project License</li><li>JavaDocs</li><li>JavaDoc Report</li><li>Source Xref</li><li>Test Xref</li><li>Unit Tests</li><li>Link Check Report</li><li>Task List</li></ul></li><li>JavaDocs</li><li>Source XReference</li><li>Test XReference</li><li>Development Process</li></ul> <div>powered by MAVEN</div>	Overview	
	Document	Description
	<a href="#">Metrics</a>	Report on source code metrics.
	<a href="#">Checkstyle</a>	Report on coding style conventions.
	<a href="#">Change Log</a>	Report on the source control changelog.
	<a href="#">Developer Activity</a>	Report on the amount of developer activity.
	<a href="#">File Activity</a>	Report on file activity.
	<a href="#">Project License</a>	Displays the primary license for the project.
	<a href="#">JavaDocs</a>	JavaDoc API documentation.
	<a href="#">JavaDoc Report</a>	Report on the generation of JavaDoc.
	<a href="#">Source Xref</a>	A set of browsable cross-referenced sources.
	<a href="#">Test Xref</a>	A set of browsable cross-referenced test sources.
	<a href="#">Unit Tests</a>	Report on the results of the unit tests.
	<a href="#">Link Check Report</a>	Report on the validity of all links in the documentation.
	<a href="#">Task List</a>	Report on tasks specified in the source code.

Figure 5.5 List of Maven-generated default reports for the `sampling` project



It is possible to control exactly what reports you want for your web site by explicitly listing the desired reports in `project.xml`. For example, if you want only the unit test report and the checkstyle report, you can write the following at the end of `project.xml`:

```
<reports>
  <report>maven-junit-report-plugin</report>
  <report>maven-checkstyle-plugin</report>
</reports>
```

### ***Describing build-related information***

Let's complete the project object model (POM) by entering build-related information into `project.xml` (listing 5.7).

**Listing 5.7** Second part of `project.xml` containing build-related information

```
<!--dependencies>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.8</version>
</dependencies-->

<build>
  <sourceDirectory>src/java</sourceDirectory>
  <unitTestSourceDirectory>src/test</unitTestSourceDirectory>
  <unitTest>
    <includes>
      <include>**/Test*.java</include>
    </includes>
    <excludes>
      <exclude>**/Test*All.java</exclude>
      <exclude>**/TestDefaultController?.java</exclude>
    </excludes>
  </unitTest>
</build>

</project>
```

- ❶ Describe the project's external dependencies. A dependency typically specifies a jar, but a dependency can be of any type. All jar dependencies are added to the classpath and used by the different plugins, such as the `junit` plugin. In this case, you have no external dependencies, which is why the `log4j` dependency is commented out in listing 5.7. In section 5.3.4, we describe in detail how Maven handles dependencies with the notion of local and remote *repositories*.
- ❷ Describe the location of the runtime sources.
- ❸ This is the location of the test sources.

- ④ Define the test classes you expect to include/exclude in the tests. Notice that you exclude the `TestDefaultController?.java` classes created in chapter 3 (the `?` stands for any character), because they are unfinished tutorial classes and are not meant to be executed.
- ③ ④ These code segments are used by the `junit` plugin.

Given just the description in listing 5.7, you can now run any of Maven's plugins to compile, package, and test your project, and more.

### 5.3.3 Executing JUnit tests with Maven

Executing JUnit tests in Maven is as simple as invoking the `junit` plugin with `maven test` from a command shell (see figure 5.6). This is close to the result of running the Ant script, back in figure 5.1—but *without writing a single line of script!* Generating a JUnit report is just as easy: Enter **maven site**, and the web site is generated, along with your JUnit report (among others). Figure 5.7 shows the JUnit report summary page for the `sampling` project.

### 5.3.4 Handling dependent jars with Maven

Maven solves another difficult issue: jar proliferation. You have probably noticed that more and more high-quality libraries are available in the Java community. Instead of reinventing the wheel, increasing numbers of projects import third-party libraries. The dark side is that building a project from its sources can be a nightmare, because you have to gather all the external jars, all in their correct versions.

Maven handles project *dependencies* (also called *artifacts*) through the use of two repositories: a remote repository and a local one. Figure 5.8 explains the workflow.

The first step for a project is to declare its dependencies in its `project.xml`. Although the `sampling` project does not depend on any external jars, let's imagine it needs to use `Log4j`. You would add the following to `project.xml`:

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.8</version>
</dependency>
```

When you execute a Maven goal on a project, here's what happens (following the numbers from figure 5.8):

- ① Check dependencies. Maven parses the dependencies located in `project.xml`.
- ② Check the dependency's existence in the local repository. For each dependency, Maven checks if it can be found in the local repository. This local repository is

```
C:\WINDOWS\system32\cmd.exe
```

```
C:\junitbook\sampling>maven test

|--_--
|  \/_ | _ _ Apache _ _ _
|  \| /  / _ \ U / -_) ~ intelligent projects ~
|__| |_ \_\_\_|_ \/_ \_\_\_|_||_| v. 1.0-beta-10

java:prepare-filesystem:
[mkdir] Created dir: C:\junitbook\sampling\target\classes

java:compile:
[echo] Compiling to C:\junitbook\sampling\target\classes
[javac] Compiling 6 source files to C:\junitbook\sampling\target\classes

java:jar-resources:

test:prepare-filesystem:
[mkdir] Created dir: C:\junitbook\sampling\target\test-classes
[mkdir] Created dir: C:\junitbook\sampling\target\test-reports

test:test-resources:





test:compile:
[javac] Compiling 9 source files to C:\junitbook\sampling\target\test-classes





test:test:
[junit] dir attribute ignored if running in the same VM
[junit] Running junitbook.sampling.TestDefaultController
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0.04 sec





BUILD SUCCESSFUL
Total time: 7 seconds





C:\junitbook\sampling>
```

**Figure 5.6** Results of executing `maven test` on the sampling project









































































































































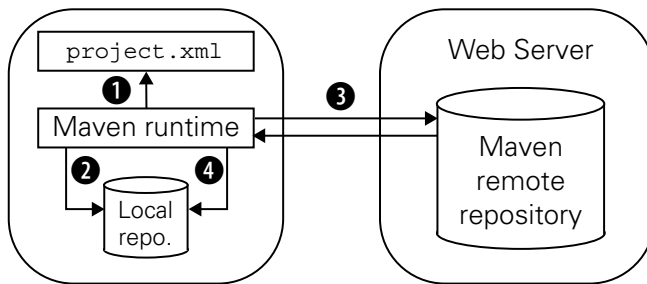









**Figure 5.7** JUnit report generated by Maven



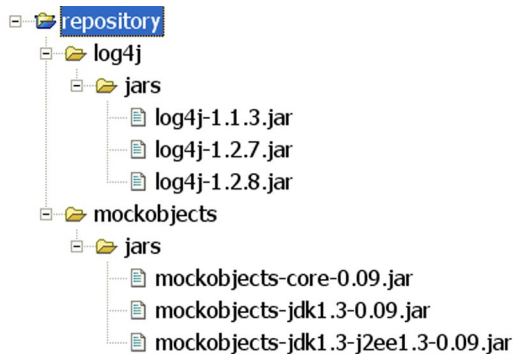
**Figure 5.8**  
How Maven resolves external dependencies

automatically created when you execute Maven the first time (it is located in your home user directory under `.maven/repository/`).

- 3 Download the dependency. If the dependency is not found in the local repository, Maven downloads it from a Maven remote repository. The default Maven remote repository is <http://www.ibiblio.org/maven/>. You can easily override this default by setting the `maven.repo.remote` property in a `project.properties` or `build.properties` file in the same location as your `project.xml` file. This is especially useful if you wish to set up a project-wide or company-wide Maven remote repository.
- 4 Store the dependency. Once the dependency has been downloaded, Maven stores it in your local repository to prevent having to fetch it again next time.

The structure of the local and remote repositories is the same. Figure 5.9 shows a very simple repository.

In figure 5.9 you can see that jars are put in a `jars/` directory. The names are suffixed with the version to let you put several versions in the same directory and for easy identification. (For example, the `log4j` jar is available in versions 1.1.3,



**Figure 5.9** Very simple portion of a Maven repository (local or remote)

1.2.7, and 1.2.8.) In addition, you can collect several jars into a common group. Figure 5.9 also shows the MockObjects jars organized into a mockobjects group. Although the figure only shows jars, it is possible to put any type of dependency in a Maven repository.

## 5.4 Running tests from Eclipse

---

Ant gives you the ability to both build and test your projects in one fell swoop. Maven goes beyond Ant to provide a comprehensive code-building environment. But how do you go about creating the code to build?

Many excellent Java applications have been written using pure editors, like Emacs, JEdit, and TextPad, to name a few. Many applications are still being written with tools like these. But more and more developers are adopting one of the many IDEs now available for Java. The IDEs have come a long way over the last few years, and many developers now consider an IDE an indispensable tool.

Most of the Java IDEs work hand in hand with build tools like Ant and Maven. On a daily basis, many developers create and test code using an IDE and then use Ant or Maven to distribute or deploy the latest version. Sometimes, the developers on a team all use the same IDE; other times they don't. But as a rule, they all use the same build system (Ant or Maven).

The Java IDEs have also been quick to adopt JUnit as part of their toolset. Most IDEs let you launch JUnit from within the environment. Developers can now debug, edit, compile, and test a class, all from within a seamless environment.

Reviewing each Java IDE is out of the scope of this book. But to give you a feel for what these IDEs can do (or what your IDE *should* be doing), we will walk through setting up a project and running tests using Eclipse.

Eclipse (<http://www.eclipse.org/>) is a very popular open source project, available for download at no charge. That Eclipse has excellent support for JUnit should be no surprise. Erich Gamma, one of the original authors of JUnit, is a key member of the Eclipse team.

If you are not using Eclipse for development, don't worry—we won't use any features specific to Eclipse. In other words, you will be able to follow along using your favorite IDE.

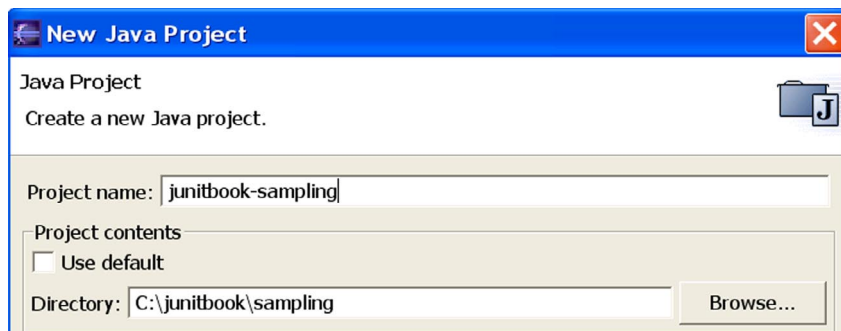
### 5.4.1 Creating an Eclipse project

Eclipse comes with a full-featured installation program that makes setup a breeze. Detailed instructions for installing and configuring Eclipse for this book can be found in appendix B. This appendix demonstrates how to import this book's

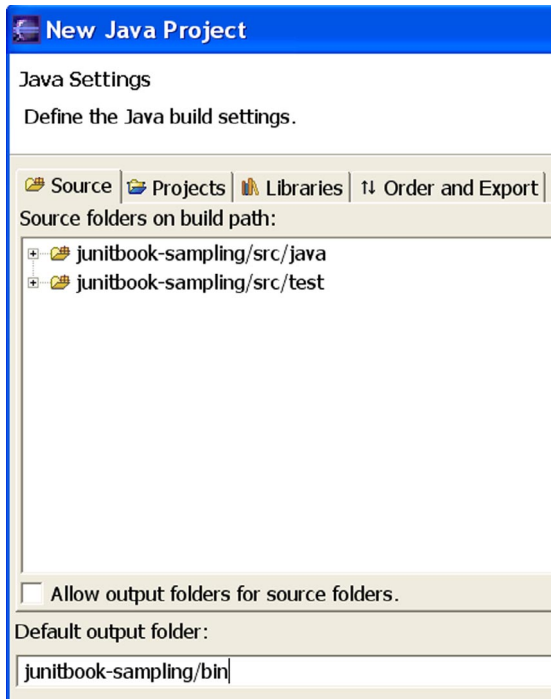
sources into Eclipse. We will demonstrate here how to create a new project from scratch in Eclipse (using the `sampling` project directory structure). We assume that you have all the book sources on your hard drive as explained in appendix A.

Follow these steps:

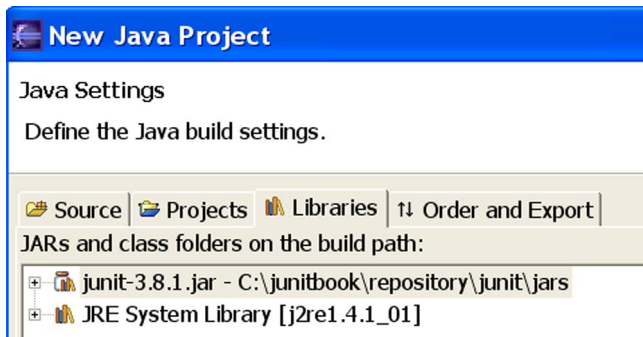
- 1 Create a new project by selecting File→New→Project.
- 2 In the dialog box that opens, select Java Project and click Next.
- 3 The screen that follows is shown in figure 5.10. Here you can choose a name for the project (for example, `junitbook-sampling`). You already have source files for this project, so be sure to *unselect* the Use Default checkbox. You simply want to map your existing files to an Eclipse project.
- 4 Point the Directory field to where you put the `sampling` project on your local disk; for example, `C:\junitbook\sampling`. Click Next.
- 5 Eclipse asks whether it should automatically detect existing classpaths. Click Yes.
- 6 Eclipse should find your two source directories: `junitbook-sampling/src/java` and `junitbook-sampling/src/test`. You should have the same entries as are shown in figure 5.11.
- 7 You need the JUnit jar to compile your project, so the next step is to add it to your list of libraries. Click the Libraries tab and then click Add External Jars. Select the JUnit jar you already placed in the `C:\junitbook\repository\junit\jars` directory. Figure 5.12 shows the Libraries tab using the example paths.
- 8 Click Finish. Eclipse creates and compiles the project.



**Figure 5.10** Enter the name and location of the new project in Eclipse.



**Figure 5.11**  
Source paths and build output  
folder for the `sampling` project



**Figure 5.12**  
Libraries definition for the  
`sampling` project

### 5.4.2 Running JUnit tests in Eclipse

Now you can run the JUnit `TestDefaultController` test case from Eclipse. Open the Java Perspective and click the `TestDefaultController` class. In the toolbar, select the Run icon and then Run As→JUnit Test, as shown in figure 5.13. The result of the execution is shown in figure 5.14.

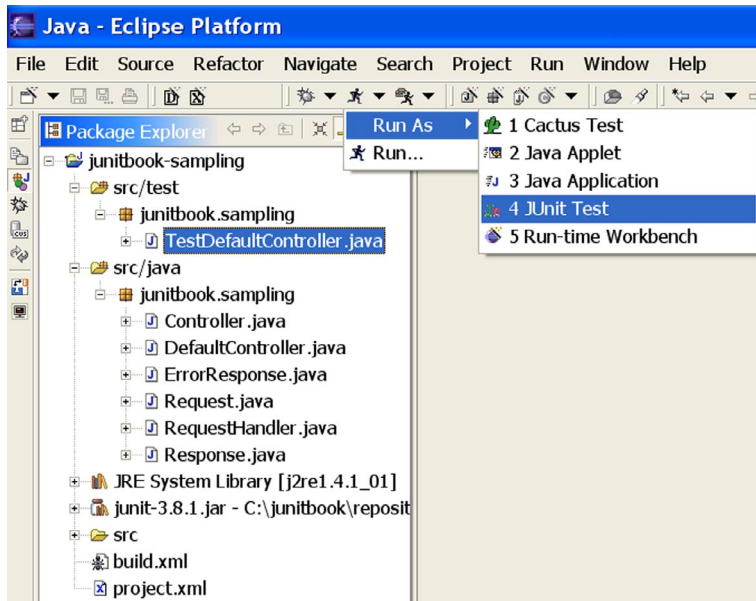


Figure 5.13 Running the TestDefaultController JUnit test case

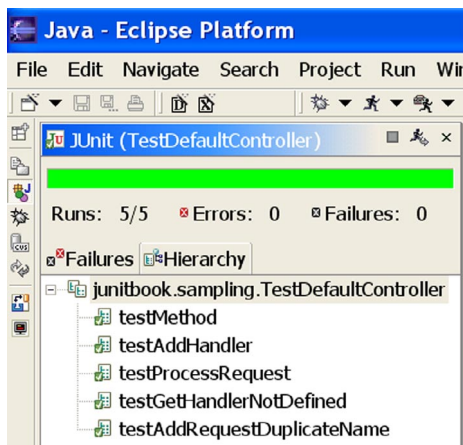


Figure 5.14  
Result of executing the  
TestDefaultController  
test case

If a test fails, you can jump back to the editing window, make some changes, compile the class, and run the test again—all without leaving the Eclipse environment.



## 5.5 Summary

---

An essential element of a unit test is that, someday, it may fail. To realize the full value of your tests, you should run them continually, even when you don't expect anything to fail. Unit tests expect the unexpected.

Using the `sampling` project from chapter 3, we walked through compiling the project and running JUnit test cases using three popular products: Ant, Maven, and Eclipse. Ant and Maven are automation tools, and Eclipse is a productivity tool (an IDE).

These products all require very little effort to set up a stable unit-testing environment. The example is quite simple, but the same automation techniques apply equally well to larger and more complex projects.

However, the unit-testing techniques we've shown so far do not scale as well. As the classes we need to test become more complex and intertwined, we need better strategies to create tests.

In the second part of this book, we'll look at some of the strategies and tools we can use to test applications piece by piece. Or, as Julius Caesar said, *Divide et impera*: We must divide and conquer.

## Part 2

# *Testing strategies*

**P**art 1 was about learning the basics of unit-testing with JUnit. However, just knowing how JUnit works and how to use it on simple examples is not enough when it comes to testing a real application. For that, JUnit alone is not enough; you need to develop testing strategies that allow you to unit-test a full-fledged application. The main issue is testing in isolation. When you're writing unit tests, you want to test your application bit by bit. How do you separate bits of functionality so that they can be tested separately? Part 2 answers this crucial question.

Chapter 6 presents the stub strategy, which allows you to test relatively coarse-grained portions of code in isolation. In chapter 7, you'll learn about a relatively new technique called mock objects, which permits fine-grained testing in isolation. With mock objects, you'll discover not only a new way of unit-testing your code but also a new way of writing it. Chapter 8 takes you into the realm of unit-testing your code when it runs in its container (in-container testing). Nowadays, almost all code runs in some kind of container with which it interacts. You'll discover a strategy to unit-test J2EE code when it runs in its container, and you'll learn about the pros and cons of this strategy when compared with the mock-objects approach.

After reading part 2, you'll be familiar with these three strategies to unit-test code in isolation. You'll be ready to tackle the last step of our journey: unit-testing all types of J2EE components (servlets, filters, JSPs, taglibs, code accessing the database, and EJBs).



# *Coarse-grained testing with stubs*

---

## ***This chapter covers***

- Introducing stubs
- Using an embedded server in place of a real web server
- Unit-testing an HTTP connection sample with stubs

*And yet it moves.*

—Galileo

As you develop your applications, you will find that the code you want to test depends on other classes, which themselves depend on other classes, which depend on the environment. For example, you might be developing an application that uses JDBC to access a database, a J2EE application (one that relies on a J2EE container for security, persistence, and other services), an application that accesses a filesystem, or an application that connects to some resource using HTTP, SOAP, or another protocol.

For applications that depend on an environment, writing unit tests is a challenge. Your tests need to be stable, and when you run them over and over, they need to yield the same results. You need a way to control the environment in which they run. One solution is to set up the real required environment as part of the tests and run the tests from within that environment. In some cases, this approach is practical and brings real added value (see chapter 8, which discusses in-container testing). However, it works well only if you can set up the real environment on your development platform, which isn't always the case.

For example, if your application uses HTTP to connect to a web server provided by another company, you usually won't have that server application available in your development environment. So, you need a way to simulate that server so you can still write tests for your code.

Or, suppose you are working with other developers on a project. What if you want to test your part of the application, but the other part isn't ready? One solution is to simulate the missing part by replacing it with a fake that behaves the same way.

There are two strategies for providing these fake objects: stubbing and using mock objects. Stubs, the original solution, are still very popular, mostly because they allow you to test code without changing it to make it testable. This is not the case with mock objects. This chapter is dedicated to stubbing, and chapter 7 covers mock objects.

## 6.1 *Introducing stubs*

---

*Stubs* are a mechanism for faking the behavior of real code that may exist or that may not have been written yet. Stubs allow you to test a portion of a system without the other part being available. They usually do not change the code you're testing but instead adapt to provide seamless integration.

**DEFINITION** *stub*—A stub is a portion of code that is inserted at runtime in place of the real code, in order to isolate calling code from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some portion of the real code.

Here are some examples of when you might use stubs:

- When you cannot modify an existing system because it is too complex and fragile
- For coarse-grained testing, such as integration testing between different subsystems

Stubs usually provide very good confidence in the system being tested. With stubs, you are not modifying the objects under test, and what you are testing is the same as what will execute in production. Tests involving stubs are usually executed in their running environment, providing additional confidence.

On the downside, stubs are usually hard to write, especially when the system to fake is complex. The stub needs to implement the same logic as the code it is replacing, and that is difficult to get right for complex logic. This issue often leads to having to debug the stubs! Here are some cons of stubbing:

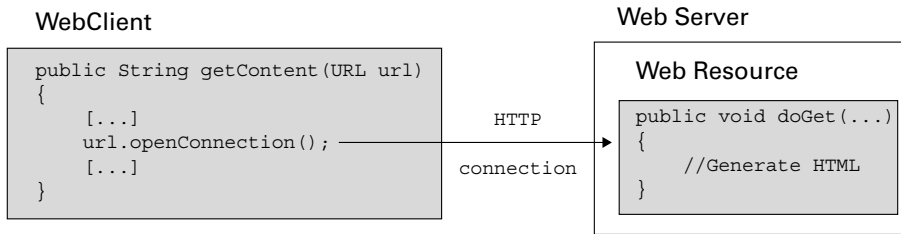
- Stubs are often complex to write and need debugging themselves.
- Stubs can be difficult to maintain because they're complex.
- A stub does not lend itself well to fine-grained unit testing.
- Each situation requires a different strategy.

In general, stubs are better adapted for replacing coarse-grained portions of code. You would usually use stubs to replace a full-blown external system like a filesystem, a connection to a server, a database, and so forth. Using stubs to replace a method call to a single class can be done, but it is more difficult. (We will demonstrate how to do this with mock objects in chapter 7.)

## 6.2 *Practicing on an HTTP connection sample*

---

To demonstrate what stubs can do, let's build some stubs for a simple application that opens an HTTP connection to a URL and reads its content. Figure 6.1 shows the sample application (limited to a `WebClient.getContent` method) performing an HTTP connection to a remote web resource. We have supposed that the remote



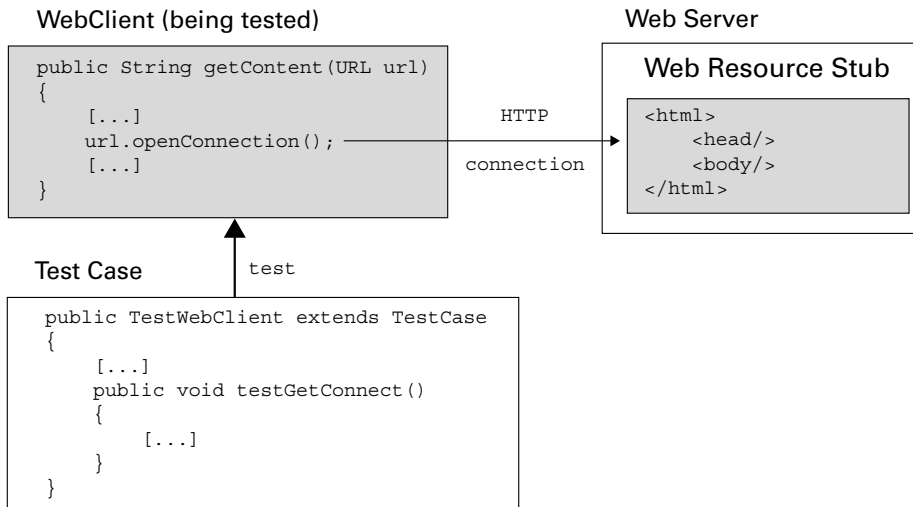
**Figure 6.1** The sample application makes an HTTP connection to a remote web resource. This is the real code in the stub definition.

web resource is a servlet, which by some means (say, by calling a JSP) generates an HTML response. Figure 6.1 is what we called the *real code* in the stub definition.

Our goal in this chapter is to unit-test the `getContent` method by stubbing the remote web resource, as demonstrated in figure 6.2. As you can see, you replace the servlet web resource with the stub, a simple HTML page returning whatever you need for the `TestWebClient` test case.

This approach allows you to test the `getContent` method independently of the implementation of the web resource (which in turn could call several other objects down the execution chain, possibly up to a database).

The important point to notice with stubbing is that `getContent` has not been modified to accept the stub. It is transparent to the application under test. In order to allow this, the external code to be replaced needs to have a well-defined



**Figure 6.2** Adding a test case and replacing the real web resource with a stub

interface and allow plugging of different implementations (the stub one, for example). In the example in figure 6.1, the interface is `URLConnection`, which cleanly isolates the implementation of the page from its caller.

Let's see a stub in action using the simple HTTP connection sample. The example in listing 6.1 from the sample application demonstrates a code snippet opening an HTTP connection to a given URL and reading the content found at that URL. Imagine the method is one part of a bigger application that you want to unit-test, and let's unit-test that method.

**Listing 6.1** Sample method opening an HTTP connection

```
package junitbook.coarse.try1;

import java.net.URL;
import java.net.HttpURLConnection;
import java.io.InputStream;
import java.io.IOException;

public class WebClient
{
    public String getContent(URL url)
    {
        StringBuffer content = new StringBuffer();

        try
        {
            HttpURLConnection connection =
                (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);

            InputStream is = connection.getInputStream();

            byte[] buffer = new byte[2048];
            int count;
            while (-1 != (count = is.read(buffer)))
            {
                content.append(new String(buffer, 0, count));
            }
        }
        catch (IOException e)
        {
            return null;
        }

        return content.toString();
    }
}
```

1 Open HTTP connection to URL

2 Start reading remote data

3 Read all data in stream

4 Return null on error

- 1 Open an HTTP connection using the `HttpURLConnection` class.



- ② ③ Read the content until there is nothing more to read.
- ④ If an error occurs, you return `null`. One might argue that a better implementation might instead return a runtime exception (or a checked exception). However, for testing purposes, returning `null` is good enough.

### 6.2.1 Choosing a stubbing solution

There are two possible scenarios in the sample application: the remote web server (see figure 6.1) could be located outside of the development platform (such as on a partner site), or it could be part of the platform where your application is deployed. However, in both cases, you need to introduce a server into your development platform in order to be able to unit-test the `WebClient` class. One relatively easy solution would be to install an Apache test server and drop some test web pages in its document root. This is a typical, widely used stubbing solution. However, it has several drawbacks:

- *Reliance on the environment*—You need to be sure the full environment is up and running before the test. If the web server is down, and the test is executed, it will fail! You will then try to debug why it is failing. Next, you will discover that the code is working fine—it was only an environmental issue generating a false warning. This kind of thing is time consuming and annoying. When you're unit testing, it is important to be able to control as much as possible of the environment in which the test executes, so that test results are reproducible.
- *Separated test logic*—The test logic is scattered in two separate places: in the JUnit test case and in the test web page. Both types of resources need to be kept in sync for the test to succeed.
- *Tests that are difficult to automate*—Automating the execution of the tests is difficult because it involves automatically deploying the web pages on the web server, automatically starting the web server, and then only running the unit tests.

Fortunately, there is an easier solution that consists of using an embedded server. You are testing in Java, so the easiest option would be to use a Java web server that you could embed in the test case class. Such a nice beast exists; it's called Jetty. For the purpose of this book, we will use Jetty to set up stubs. For more information about Jetty in general, visit <http://jetty.mortbay.org/jetty/index.html>.

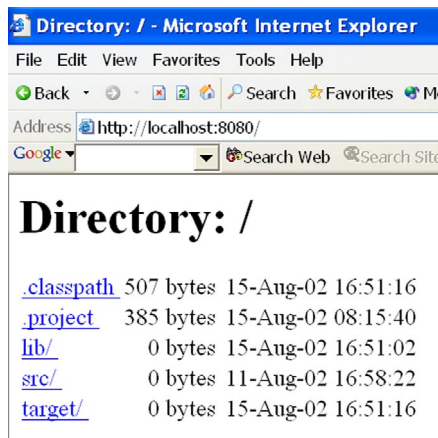
Why Jetty? Because it's fast (important when running tests), it's lightweight (a single jar file to put in the classpath), and it can be completely controlled in Java

from your test case, which means you can tell it how to behave for the purpose of your tests. Additionally, it is a very good web/servlet container that you can use in production. This is not specifically needed for your tests, but it is always nice to use best-of-breed technology.

Using Jetty allows you to eliminate the drawbacks: The server is started from the JUnit test case, the tests are all written in Java in one location, and automating the test suite is a nonissue. Thanks to Jetty's modularity, the real point of the exercise is only to stub the Jetty handlers and not the whole server from the ground up.

## 6.2.2 Using Jetty as an embedded server

In order to better understand how to set up and control Jetty from your tests, let's implement a simple example that starts Jetty from your Java code. Listing 6.2 shows how to start it from Java and how to define a document root (/) from which to start serving files. Figure 6.3 shows the result when you run the application and open a browser on the URL `http://localhost:8080`.



**Figure 6.3**  
Testing the `JettySample` in a browser. These are the results when you run listing 6.2 and open a browser on `http://localhost:8080`.

### Listing 6.2 Starting Jetty in embedded mode—`JettySample` class

```
package junitbook.coarse;

import org.mortbay.http.HttpContext;
import org.mortbay.http.HttpServer;
import org.mortbay.http.SocketListener;
import org.mortbay.http.handler.ResourceHandler;

public class JettySample
{
    public static void main(String[] args) throws Exception
```