count and the stack trace for failing tests. Figure 2.4 shows what a failed test looks like in the Swing test runner.

JUnit distinguishes between failures and errors. *Failures* are to be expected. From time to time changes in the code will cause an assertion to fail. When it does, you fix the code so the failure goes away. An *error* is an unexpected condition that is not expected by the test, such as an exception in a conventional program.

Of course, an error may indicate a failure in the underlying environment. The test itself may not be broken. A good heuristic in the face of an error is the following:

- Check the environment. (Is the database up? What about the network?)
- Check the test.
- Check the code.

Figure 2.5 shows what an error condition looks like in the Swing test runner. At the end of a suite, the test runner provides a tally of how many tests passed and failed, along with details of any test that failed.

With JUnit 3.8.1, you can use the automatic suite mechanism to ensure that any new tests you write are included in the run. If you'd like to retain the results

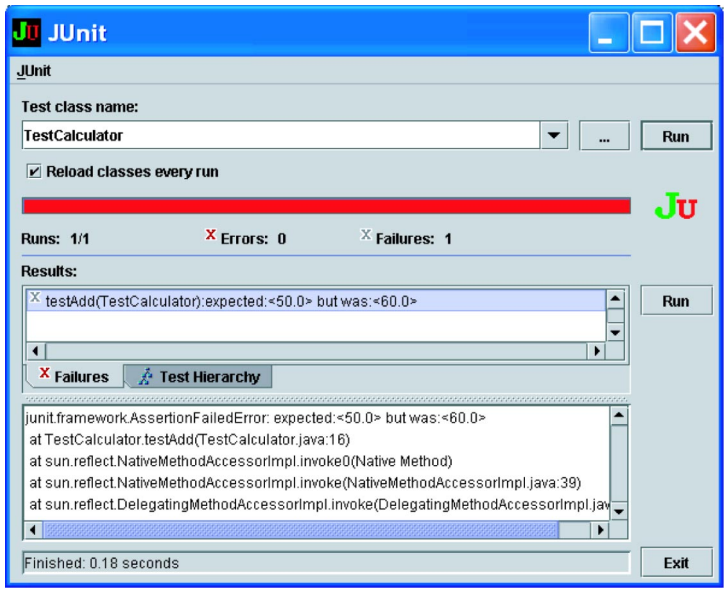

Figure 2.4 Oops—time to fix the code!

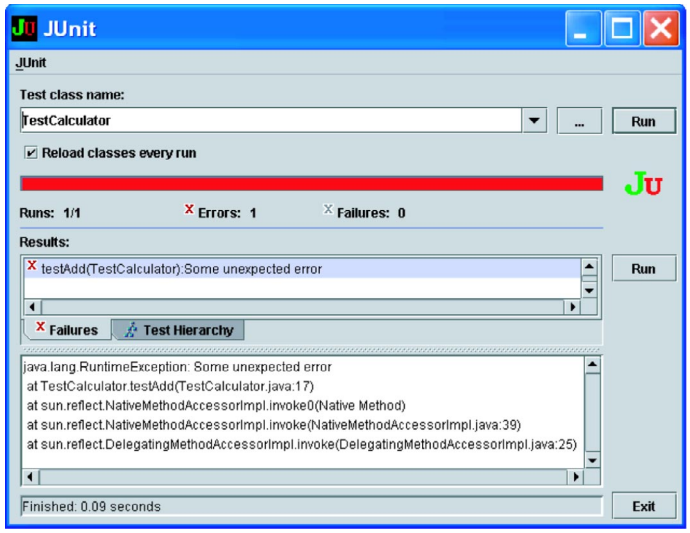

Figure 2.5
Oh, no! You have to fix the test!

for later review, you can do so using Ant, Eclipse, and other tools. See chapter 5 for more about using automated tools.

The `TestResult` is used by almost all the JUnit classes internally. As a JUnit test writer, you won't interact with the `TestResult` directly; but in trying to understand how JUnit works, it's helpful to know that it exists.

> ***Design patterns in action: Collecting Parameter***
>
> "When you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you."[3] The `TestResult` class is an example of the Collecting Parameter pattern.

2.5 Observing results with TestListener

The `TestResult` collects information about the test, and the `TestRunner` reports it. But does an object have to be a `TestRunner` to report on a test? Can more than one object report on a test at once?

The JUnit framework provides the `TestListener` interface to help objects access the `TestResult` and create useful reports. The `TestRunners` implement `TestListener`, as do many of the special JUnit extensions. Any number of `TestListeners` can register with the framework and do whatever they need to do

[3] Kent Beck, *Smalltalk Best Practice Patterns* (Upper Saddle River, NJ: Prentice Hall, 1996).

with the information provided by the `TestResult`. Table 2.2 describes the `TestListener` interface.

> **NOTE** Although the `TestListener` interface is an essential part of the JUnit framework, it is not an interface that you will implement when writing your own tests. We provide this section for completeness only.

Table 2.2 The `TestListener` interface

Method	Description
`void` **`addError`**`(Test test, Throwable t)`	Called when an error occurs
`void` **`addFailure`**`(Test test, AssertionFailedError e)`	Called when a failure occurs
`void` **`endTest`**`(Test test)`	Called when a test ends
`void` **`startTest`**`(Test test)`	Called when a test begins

As mentioned, although the `TestListener` interface is an interesting bit of plumbing, you would only need to implement it if you were extending the JUnit framework, rather than just using it.

Design patterns in action: Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically."[4] The `TestRunner` registering as a `TestListener` with the `TestResult` is an example of the Observer pattern.

2.6 Working with TestCase

The JUnit big picture is that the `TestRunner` runs a `TestSuite`, which contains one or more `TestCases` (or other `TestSuites`). On a daily basis, you usually work only with the `TestCases`.

The framework ships with ready-to-use graphical and textual `TestRunners`. The framework can also generate a default runtime `TestSuite` for you. So, the only class you absolutely must provide yourself is the `TestCase`. A typical `TestCase` includes two major components: the fixture and the unit tests.

[4] Ibid.

2.6.1 Managing resources with a fixture

Some tests require resources that can be a bother to set up. A prime example is something like a database connection. Several tests in a `TestCase` may need a connection to a test database and access to a number of test tables. Another series of tests may require complex data structures or a long series of random inputs.

Putting common setup code in your tests doesn't make sense. You are not testing your ability to create a resource—you just need a stable background environment in which to run tests. The set of background resources that you need to run a test is commonly called a *test fixture*.

DEFINITION *fixture*—The set of common resources or data that you need to run one or more tests.

A fixture is automatically created and destroyed by a `TestCase` through its `setUp` and `tearDown` methods. The `TestCase` calls `setUp` before running *each* of its tests and then calls `tearDown` when *each* test is complete. A key reason why you put more than one test method into the same `TestCase` is to share the fixture code. The `TestCase` life cycle is depicted in figure 2.6.

In practice, many developers now use *mock objects* or *stubs* to simulate database connections and other complex resources. For more about mock objects and stubs, see chapters 6 and 7, respectively.

A database connection is a good example of why you might need a fixture. If a `TestCase` includes several database tests, they each need a fresh connection to the database. A fixture makes it easy for you to open a new connection for each test without replicating code. For more about testing databases, see chapter 11. You can also use a fixture to generate input files; doing this means you do not have to carry your test files with your tests, and you always have a known state before the test is executed.

JUnit also reuses code through the utility methods provided by the `Assert` interface, as we'll explain in the next section.

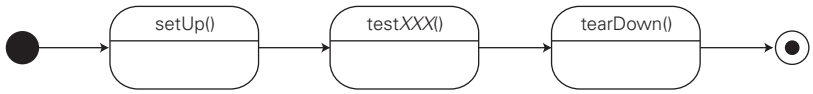

Figure 2.6 The `TestCase` life cycle re-creates your fixture, or scaffolding, for each test method.

JUnit design goals

When it is easy to reuse a fixture between tests, you can write tests more quickly. Each time you reuse the fixture, you decrease the initial investment made when the fixture was created. The `TestCase` fixtures speak to JUnit's third design goal:

The framework must lower the cost of writing tests by reusing code.

2.6.2 Creating unit test methods

Fixtures are a great way to reuse setup code. But there are many other common tasks that many tests perform over and over again. The JUnit framework encapsulates the most common testing tasks with an array of assert methods. The assert methods can make writing unit tests much easier.

The Assert supertype

In listing 2.2, we introduced the `Test` interface that both `TestSuite` and `TestCase` implement. The `Test` interface specifies only two methods, `countTestCases` and `run`. But in writing the `TestCalculator` case, you also used an assert method inherited from the base `TestCase` class.

The assert methods are defined in a utility class named (you guessed it) `Assert`. This class contains many of the nuts and bolts you use to construct tests.

If you look at the Javadoc or the source, you'll see that, with 38 signatures, the `Assert` interface is longer than most. But if you look a little closer, you'll notice that `Assert` has only eight public methods, as shown in table 2.3.

Table 2.3 The eight core methods provided by the `Assert` superclass

Method	Description
`assertTrue`	Asserts that a condition is true. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertFalse`	Asserts that a condition is true. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertEquals`	Asserts that two objects are equal. If they are not, the method throws an `AssertionFailedError` with the given message (if any).
`assertNotNull`	Asserts that an object isn't null. If it is, the method throws an `AssertionFailedError` with the message (if any).

continued on next page

Table 2.3 The eight core methods provided by the `Assert` superclass *(continued)*

Method	Description
`assertNull`	Asserts that an object is null. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertSame`	Asserts that two objects refer to the same object. If they do not, the method throws an `AssertionFailedError` with the given message (if any).
`assertNotSame`	Asserts that two objects do not refer to the same object. If they do, the method throws an `AssertionFailedError` with the given message (if any).
`fail`	Fails a test with the given message.

The Javadoc is bulked up by convenience forms of these eight methods. The convenience forms make it easy to pass whatever type you need in your test. The `assertEquals` method, for example, has 20 forms! Most of the `assertEquals` forms are conveniences that end up calling the core `assertEquals(String message, Object expected, Object actual)` form.

The TestCase members

Along with the methods provided by `Assert`, `TestCase` implements 10 methods of its own. Table 2.4 overviews the 10 `TestCase` methods *not* provided by the `Assert` interface.

Table 2.4 The 10 additional methods provided by `TestCase`

Method	Description
`countTestCases`	Counts the number of `TestCase`s executed by `run(TestResult result)`. (Specified by the `Test` interface.)
`createResult`	Creates a default `TestResult` object.
`getName`	Gets the name of a `TestCase`.
`run`	Runs the `TestCase` and collects the results in `TestResult`. (Specified by the `Test` interface.)
`runBare`	Runs the test sequence without any special features, like automatic discovery of test methods.
`runTest`	Override to run the test and assert its state.
`setName`	Sets the name of a `TestCase`.
`setUp`	Initializes the fixture, for example, to open a network connection. This method is called before a test is executed. (Specified by the `Test` interface.)

continued on next page

Table 2.4 The 10 additional methods provided by `TestCase` *(continued)*

Method	Description
`tearDown`	De-initializes the fixture, for example, to close a network connection. This method is called after a test is executed. (Specified by the `Test` interface.)
`toString`	Returns a string representation of the `TestCase`.

In practice, many `TestCases` use the `setUp` and `tearDown` methods. The other methods in table 2.4 are mainly of interest to developers creating JUnit extensions. Used together, the 18 methods in tables 2.3 and 2.4 provide you with all the functionality you need to write tests with JUnit.

Keeping tests independent

As you begin to write your own tests, remember the first rule: *Each unit test must run independently of all other unit tests.* Unit tests must be able to be run in any order. One test must not depend on some side effect caused by a previous test (for example, a member variable being left in a certain state). If tests begin to depend on one another, you are inviting trouble. Here are some of the problems with co-dependent tests:

- *Not portable*—By default, JUnit finds test methods by reflection. The reflection API does not guarantee an order in which it returns the method names. If your tests depend on an ordering, then your suite may work in one Java Virtual Machine (JVM) but fail in another.
- *Hard to maintain*—When you modify one test, you may find that a number of other tests are affected. If you need to change the other tests, you spend time maintaining tests—time that could have been spent developing code.
- *Not legible*—To understand how co-dependent tests work, you must understand how each one works in turn. Tests become more difficult to read and harder to maintain. A good test must be easy to read and simple to maintain.

2.7 Stepping through TestCalculator

Let's demonstrate how all the core JUnit classes work together, using the `TestCalculator` test from listing 2.1 as an example. This is a very simple test class with a single test method:

```
import junit.framework.TestCase;

public class TestCalculator extends TestCase
```

```
{
  public void testAdd()
  {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    assertEquals(60, result, 0);
  }
}
```

When you start a JUnit test runner by typing **`java junit.swingui.TestRunner TestCalculator`**, the JUnit framework performs the following actions:

- Creates a `TestSuite`
- Creates a `TestResult`
- Executes the test methods (`testAdd` in this case)

We'll present these steps using standard Universal Modeling Language (UML) sequence diagrams.

> ***More about UML and design patterns***
>
> As the sage said, a picture tells a thousand words; a symbol tells ten thousand more. Today many developers rely on UML to represent software components and design patterns to symbolize the deeper meanings of designs. To learn more about UML, we recommend *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, by Martin Fowler and Kendall Scott (Reading, MA: Addison-Wesley, 1999). For more about design patterns, we recommend the classic *Design Patterns,* by Erich Gamma et al (Reading, MA: Addison-Wesley, 1995); and *Patterns of Enterprise Application Architecture*, by Martin Fowler (Boston: Addison-Wesley, 2003). You don't need to understand UML or design patterns to test your code, but both disciplines can help you design code to test.

2.7.1 Creating a TestSuite

NOTE In the UML diagrams that follow, we refer to a `TestRunner` class. Although there is no `TestRunner` interface, the JUnit test runners all extend the `BaseTestRunner` class and are named `TestRunner`: `junit.swingui.TestRunner` for the Swing test runner and `junit.textui.TestRunner` for the text test runner. By extrapolation, we are giving the name `TestRunner` to any class that extends `BaseTestRunner`. This means whenever we mention `TestRunner`, you can mentally replace it with any JUnit test runner.

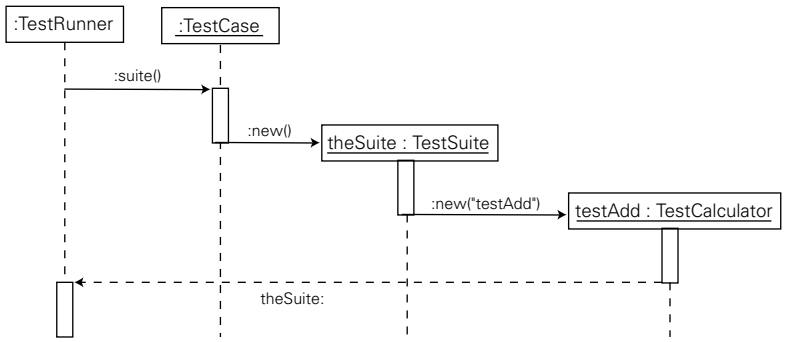

Figure 2.7 JUnit creates an explicit test suite (when the `suite` method is defined in the test case).

The `TestRunner` begins by looking for a `suite` method in the `TestCalculator` class. Had one existed, the `TestRunner` would have called it, as shown in figure 2.7. Then this `suite` method would have created the different `TestCase` classes, adding them to the test suite (see section 2.3).

Because there is no `suite` method in the `TestCalculator` class, the `TestRunner` creates a default `TestSuite` object, as shown in figure 2.8.

The main difference between figures 2.7 and 2.8 is that in figure 2.8, the discovery of the `TestCalculator` test methods is done using Java introspection, whereas in figure 2.7, the `suite` method explicitly defines the `TestCalculator` test methods. For example:

```
public static Test suite()
{
    TestSuite suite = new TestSuite();
    suite.addTest(new TestCalculator("testAdd"));
    return suite;
}
```

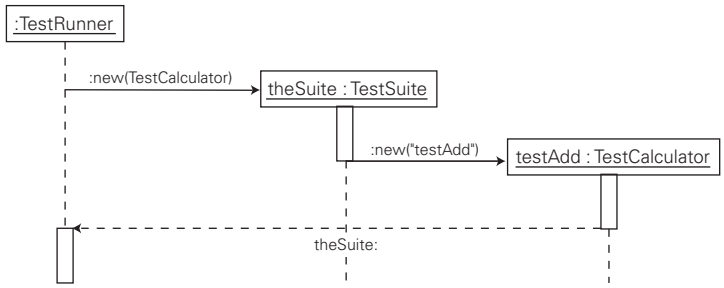

Figure 2.8 JUnit creates an automatic test suite (when the `suite` method is not defined in the test case).

2.7.2 Creating a TestResult

Figure 2.9 demonstrates the steps that JUnit follows to create a `TestResult` object that contains the test results (success or failures or errors).

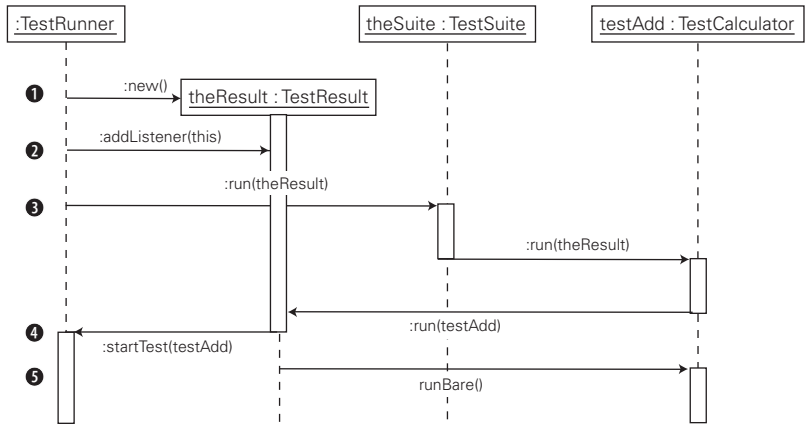

Figure 2.9 JUnit creates a `TestResult` object to collect the test results (good or bad).

These steps are as follows:

1. At ❶ in the figure, the `TestRunner` instantiates the `TestResult` object that will hold the test results as the tests are executed one after another.
2. The `TestRunner` registers against the `TestResult` so that it will receive events that happen during the execution of the tests (❷). This is a typical example of the Observer pattern in action. A `TestResult` advertises these methods:
   - A test has been started (`startTest`; see ❹).
   - A test has failed (`addFailure`; see figure 2.10).
   - A test has thrown an unexpected exception (`addError`; see figure 2.10).
   - A test has ended (`endTest`; see figure 2.10).
3. Knowing about these events allows the `TestRunner` to display a progress bar as the tests progress and to display failures and errors as they happen (instead of having to wait until the end of all the tests).
4. The `TestRunner` starts the tests by calling the `TestSuite`'s `run(TestResult)` method (❸).

5 The `TestSuite` calls the `run(TestResult)` method for each of the `TestCase` instances it holds.

6 The `TestCase` uses the `TestResult` instance that was passed to it to call its `run(Test)` method, passing itself as a parameter so that the `TestResult` can then call it back with `runBare` (❺). The reason is that JUnit needs to give the control to the `TestResult` instance so that it can alert all its listeners that the test has started (❹).

2.7.3 Executing the test methods

You have seen in the previous section that for each `TestCase` in the `TestSuite`, the `runBare` method is called. In this case, because you have a single `testAdd` test in `TestCalculator`, it will be called only once.

Figure 2.10 highlights the steps for executing a single test method. The steps in figure 2.10 are as follows:

1 At ❶ in the figure, the `runBare` method executes the `setUp`, `testAdd`, and `tearDown` methods.

2 If any test failure or test error happens during the execution of any of these three methods, the `TestResult` notifies its listeners by calling `addFailure` (❷) and `addError` (❸), respectively.

3 If any errors occur, the `TestRunner` lists them. Otherwise, the bar turns green—and you know the code is clean.

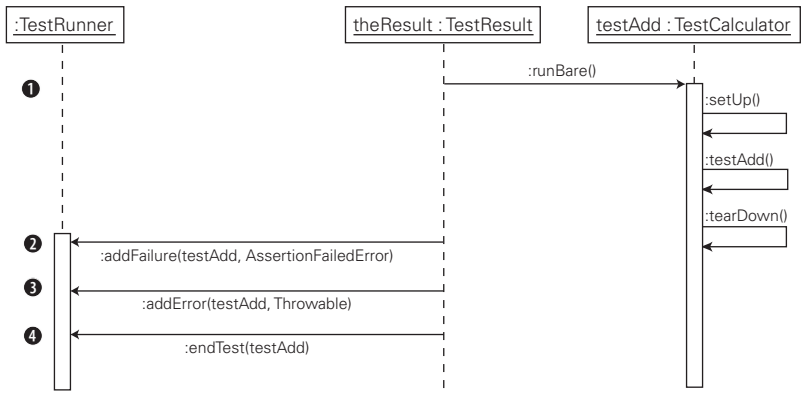

Figure 2.10 JUnit executes a test method and notifies test listeners about failure, errors, and the end of the test.

4 When the `tearDown` method has finished executing, the test is finished. The `TestResult` signals this fact to its listeners by calling the `endTest` method (❹).

2.7.4 Reviewing the full JUnit life cycle

The full JUnit life cycle we have described in the previous sections is shown in figure 2.11.

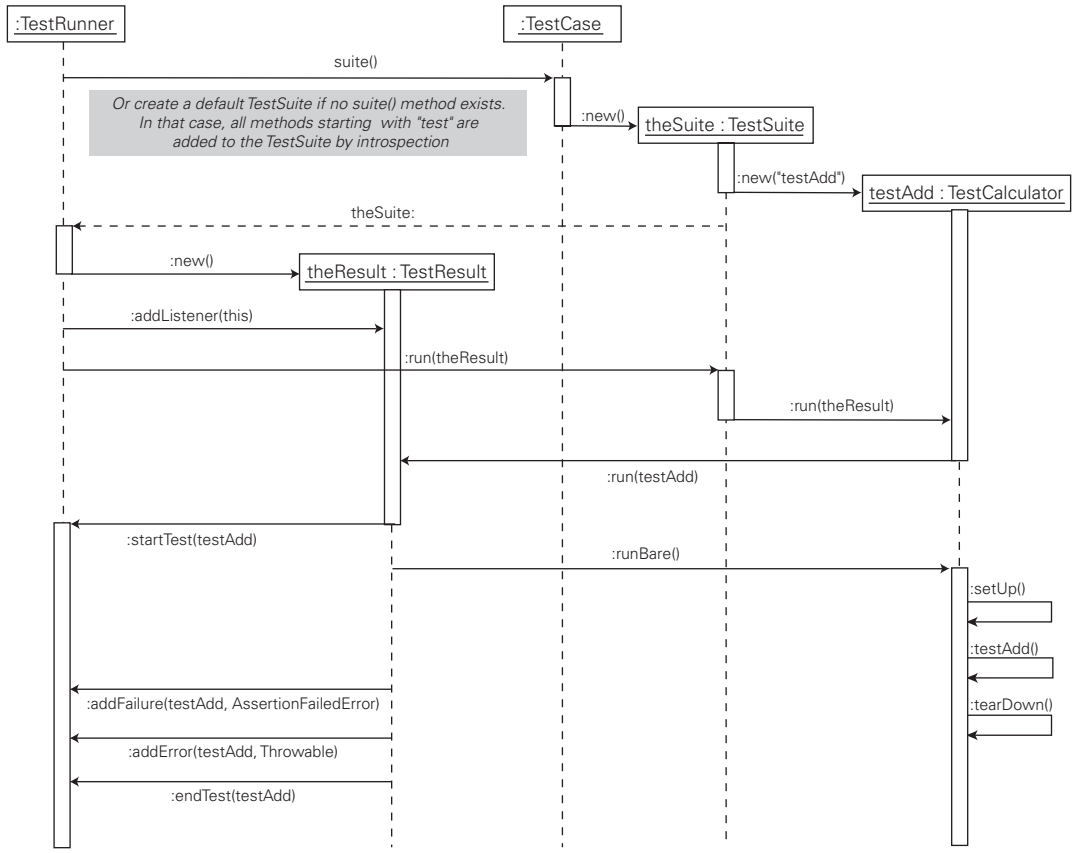

Figure 2.11 The full JUnit life cycle applied on the `TestCalculator` class

2.8 Summary

As you saw in chapter 1, it's not hard to jump in and begin writing JUnit tests for your applications. In this chapter, we zoomed in and took a closer look at how JUnit works under the hood.

A key feature of JUnit is that it provides a convenient spot to hang the scaffolding (or fixture) that you need for a test. Also built into JUnit are several convenient assert methods that make tests quick and easy to build. With the JUnit test runners, unit tests become so convenient that some developers have made testing an integral part of writing code.

With the responsibilities of the JUnit classes defined, we presented a complete UML diagram of the JUnit life cycle. Being able to visualize the JUnit life cycle can be very helpful when you're creating tests for more complex objects.

In chapter 3, we present a more sophisticated use case and walk through the type of tests you will create for a package of related objects.

3 Sampling JUnit

This chapter covers

- Testing larger components
- Project infrastructure

Tests are the Programmer's Stone, transmuting fear into boredom.

—Kent Beck, *Test First Development*

Now that you've had an introduction to JUnit, you're ready to see how it works on a practical application. Let's walk through a case study of testing a single, significant component, like one your team leader might assign to you. We should choose a component that is both useful *and* easy to understand, common to many applications, large enough to give us something to play with, but small enough that we can cover it here. How about a *controller*?

In this chapter, we'll first 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'll create tests for exceptional conditions, to be sure the code behaves well even when things go wrong.

3.1 Introducing the controller component

Core J2EE Patterns describes a controller as a component that "interacts with a client, controlling and managing the handling of each request," and tells us that it is used in both presentation-tier and business-tier patterns.[1]

In general, a controller does the following:

- Accepts requests
- Performs any common computations on the request
- Selects an appropriate request handler
- Routes the request so that the handler can execute the relevant business logic
- May provide a top-level handler for errors and exceptions

A controller is a handy class and can be found in a variety of applications. For example, in a presentation-tier pattern, a web controller accepts HTTP requests and extracts HTTP parameters, cookies, and HTTP headers, perhaps making the HTTP elements easily accessible to the rest of the application. A web controller determines the appropriate business logic component to call based on elements in the request, perhaps with the help of persistent data in the HTTP session, a database, or some other resource. The Apache Struts framework is an example of a web controller.

---

[1] Deepak Alur, John Crupi, and Dan Malks, *Core J2EE Patterns: Best Practices and Design Strategies* (Upper Saddle River, NJ: Prentice Hall, 2001).

Another common use for a controller is to handle applications in a business-tier pattern. Many business applications support several presentation layers. Web applications may be handled through HTTP clients. Desktop applications may be handled through Swing clients. Behind these presentation tiers there is often an application controller, or *state machine*. Many Enterprise JavaBean (EJB) applications are implemented this way. The EJB tier has its own controller, which connects to different presentation tiers through a business façade or *delegate*.

Given the many uses for a controller, it's no surprise that controllers crop up in a number of enterprise architecture patterns, including Page Controller, Front Controller, and Application Controller.[2] The controller you will design here could be the first step in implementing any of these classic patterns.

Let's work through the code for the simple controller, to see how it works, and then try a few tests. If you would like to follow along and run the tests as you go, all the source code for this chapter is available at SourceForge (http://junit-book.sf.net). See appendix A for more about setting up the source code.

3.1.1 Designing the interfaces

Looking over the description of a controller, four objects pop out: the `Request`, the `Response`, the `RequestHandler`, and the `Controller`. The `Controller` accepts a `Request`, dispatches a `RequestHandler`, and returns a `Response` object. With a description in hand, you can code some simple starter interfaces, like those shown in listing 3.1.

Listing 3.1 Request, Response, RequestHandler, and Controller interfaces

```
public interface Request
{
    String getName();      ❶
}

public interface Response      ❷
{
}

public interface RequestHandler
{
    Response process(Request request) throws Exception;      ❸
}

public interface Controller
{
    Response processRequest(Request request);      ❹
```

[2] Martin Fowler, *Patterns of Enterprise Application Architecture* (Boston: Addison-Wesley, 2003).

```
    void addHandler(Request request, RequestHandler requestHandler);      ❺
}
```

❶ Define a `Request` interface with a single `getName` method that returns the request's unique name, just so you can differentiate one request from another. As you develop the component you will need other methods, but you can add those as you go along.

❷ Here you specify an empty interface. To begin coding, you only need to return a `Response` object. What the `Response` encloses is something you can deal with later. For now, you just need a `Response` type you can plug into a signature.

❸ Define a `RequestHandler` that can process a `Request` and return your `Response`. `RequestHandler` is a helper component designed to do most of the dirty work. It may call upon classes that throw any type of exception. So, `Exception` is what you have the `process` method throw.

❹ Define a top-level method for processing an incoming request. After accepting the request, the controller dispatches it to the appropriate `RequestHandler`. Notice that `processRequest` does not declare any exceptions. This method is at the top of the control stack and should catch and cope with any and all errors internally. If it did throw an exception, the error would usually go up to the Java Virtual Machine (JVM) or servlet container. The JVM or container would then present the user with one of those nasty white screens. Better you handle it yourself.

❺ This is a very important design element. The `addHandler` method allows you to extend the `Controller` without modifying the Java source.

> ***Design patterns in action: Inversion of Control***
>
> Registering a handler with the controller is an example of Inversion of Control. This pattern is also known as the *Hollywood Principle*, or "Don't call us, we'll call you." Objects register as handlers for an event. When the event occurs, a hook method on the registered object is invoked. Inversion of Control lets frameworks manage the event life cycle while allowing developers to plug in custom handlers for framework events.[3]

[3] John Earles, "Frameworks! Make Room for Another Silver Bullet": http://www.cbd-hq.com/PDFs/cbdhq_000301je_frameworks.pdf.

3.1.2 Implementing the base classes

Following up on the interfaces in listing 3.1, listing 3.2 shows a first draft of the simple controller class.

Listing 3.2 The generic controller

```
package junitbook.sampling;

import java.util.HashMap;
import java.util.Map;

public class DefaultController implements Controller
{
    private Map requestHandlers = new HashMap();                    (1)

    protected RequestHandler getHandler(Request request)            (2)
    {
        if (!this.requestHandlers.containsKey(request.getName()))
        {
            String message = "Cannot find handler for request name "
                + "[" + request.getName() + "]";
            throw new RuntimeException(message);                    (3)
        }
        return (RequestHandler) this.requestHandlers.get(           (4)
            request.getName());
    }

    public Response processRequest(Request request)                 (5)
    {
        Response response;
        try
        {
            response = getHandler(request).process(request);
        }
        catch (Exception exception)
        {
            response = new ErrorResponse(request, exception);
        }
        return response;
    }

    public void addHandler(Request request,
        RequestHandler requestHandler)
    {
        if (this.requestHandlers.containsKey(request.getName()))
        {
            throw new RuntimeException("A request handler has "     (6)
                + "already been registered for request name "
                + "[" + request.getName() + "]");
        }
        else
```

```
        {
            this.requestHandlers.put(request.getName(),
                requestHandler);
        }
    }
}
```

❶ Declare a `HashMap` (`java.util.HashMap`) to act as the registry for your request handlers.

❷ Add a protected method, `getHandler`, to fetch the `RequestHandler` for a given request.

❸ If a `RequestHandler` has not been registered, you throw a `RuntimeException` (`java.lang.RuntimeException`), because this happenstance represents a programming mistake rather than an issue raised by a user or external system. Java does not require you to declare the `RuntimeException` in the method's signature, but you can still catch it as an exception. An improvement would be to add a specific exception to the controller framework (`NoSuitableRequestHandlerException`, for example).

❹ Your utility method returns the appropriate handler to its caller.

❺ This is the core of the `Controller` class: the `processRequest` method. This method dispatches the appropriate handler for the request and passes back the handler's `Response`. If an exception bubbles up, it is caught in the `ErrorResponse` class, shown in listing 3.3.

❻ Check to see whether the name for the handler has been registered, and throw an exception if it has. Looking at the implementation, note that the signature passes the request object, but you only use its name. This sort of thing often occurs when an interface is defined *before* the code is written. One way to avoid over-designing an interface is to practice Test-Driven Development (see chapter 4).

Listing 3.3 Special response class signaling an error

```
package junitbook.sampling;

public class ErrorResponse implements Response
{
    private Request originalRequest;
    private Exception originalException;

    public ErrorResponse(Request request, Exception exception)
    {
        this.originalRequest = request;
        this.originalException = exception;
```

```
    }

    public Request getOriginalRequest()
    {
        return this.originalRequest;
    }

    public Exception getOriginalException()
    {
        return this.originalException;
    }
}
```

At this point, you have a crude but effective skeleton for the controller. Table 3.1 shows how the requirements at the top of this section relate to the source code.

Table 3.1 Resolving the base requirements for the component

Requirement	Resolution
Accept requests	`public Response processRequest(Request request)`
Select handler	`this.requestHandlers.get(request.getName())`
Route request	`response = getRequestHandler(request).process(request);`
Error-handling	Subclass `ErrorResponse`

The next step for many developers would be to cobble up a stub application to go with the skeleton controller. But not us! As "test-infected" developers, we can write a test suite for the controller without fussing with a stub application. That's the beauty of unit testing! You can write a package and verify that it works, all outside of a conventional Java application.

3.2 Let's test it!

A fit of inspiration has led us to code the four interfaces shown in listing 3.1 and the two starter classes shown in listings 3.2 and 3.3. If we don't write an automatic test now, the Bureau of Extreme Programming will be asking for our membership cards back!

Listings 3.2 and 3.3 began with the simplest implementations possible. So, let's do the same with the new set of unit tests. What's the simplest possible test case we can explore?

3.2.1 Testing the DefaultController

How about a test case that instantiates the `DefaultController` class? The first step in doing anything useful with the controller is to construct it, so let's start there.

Listing 3.4 shows the bootstrap test code. It constructs the `DefaultController` object and sets up a framework for writing tests.

Listing 3.4 TestDefaultController—a bootstrap iteration

```
package junitbook.sampling;

import junit.framework.TestCase;

public class TestDefaultController extends TestCase      ❶
{
    private DefaultController controller;

    protected void setUp() throws Exception      ❷
    {
        controller = new DefaultController();
    }

    public void testMethod()      ❸
    {
        throw new RuntimeException("implement me");      ❹
    }
}
```

❶ Start the name of the test case class with the prefix *Test*. Doing so marks the class as a test case so that you can easily recognize test classes and possibly filter them in build scripts.

❷ Use the default `setUp` method to instantiate `DefaultController`. This is a built-in extension point that the JUnit framework calls between test methods.

❸ Here you insert a dummy test method, just so you have something to run. As soon as you are sure the test infrastructure is working, you can begin adding real test methods. Of course, although this test runs, it also fails. The next step will be to fix the test!

❹ Use a "best practice" by throwing an exception for test code that has not yet been implemented. This prevents the test from passing and reminds you that you must implement this code.

3.2.2 Adding a handler

Now that you have a bootstrap test, the next step is to decide what to test first. We started the test case with the `DefaultController` object, because that's the point of

this exercise: to create a controller. You wrote some code and made sure it compiled. But how can you test to see if it works?

The purpose of the controller is to process a request and return a response. But before you process a request, the design calls for adding a `RequestHandler` to do the actual processing. So, first things first: You should test whether you can add a `RequestHandler`.

The tests you ran in chapter 1 returned a known result. To see if the test succeeded, you compared the result you expected with whatever result the object you were testing returned. The signature for `addHandler` is

```
void addHandler(Request request, RequestHandler requestHandler)
```

To add a `RequestHandler`, you need a `Request` with a known name. To check to see if adding it worked, you can use the `getHandler` method from `DefaultController`, which uses this signature:

```
RequestHandler getHandler(Request request)
```

This is possible because the `getHandler` method is protected, and the test classes are located in the same package as the classes they are testing.

For the first test, it looks like you can do the following:

1. Add a `RequestHandler`, referencing a `Request`.
2. Get a `RequestHandler` and pass the same `Request`.
3. Check to see if you get the same `RequestHandler` back.

Where do tests come from?

Now you know what objects you need. The next question is, where do these objects come from? Should you go ahead and write some of the objects you will use in the application, like a logon request?

The point of unit testing is to test one object at a time. In an object-oriented environment like Java, objects are designed to interact with other objects. To create a unit test, it follows that you need two flavors of objects: the *domain object* you are testing and *test objects* to interact with the object under test.

DEFINITION *domain object*—In the context of unit testing, the term *domain object* is used to contrast and compare the objects you use *in* your application with the objects that you use to *test* your application (*test objects*). Any object under test is considered to be a domain object.

If you used another domain object, like a logon request, and a test failed, it would be hard to identify the culprit. You might not be able to tell if the problem was with the controller or the request. So, in the first series of tests, the only class you will use in production is `DefaultController`. Everything else should be a special test class.

> ***JUnit best practices: unit-test one object at a time***
>
> A vital aspect of unit tests is that they are finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If more than one object is put under test, you cannot predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects.
>
> Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types of tests.

Where do test classes live?

Where do you put the test classes? Java provides several alternatives. For starters, you could do one of the following:

- Make them public classes in your package
- Make them inner classes within your test case class

If the classes are simple and likely to stay that way, then it is easiest to code them as inner classes. The classes in this example are pretty simple.

Listing 3.5 shows the inner classes you can add to the `TestDefaultController` class.

Listing 3.5 Test classes as inner classes

```
public class TestDefaultController extends TestCase
{
[...]
    private class TestRequest implements Request      ❶
    {
        public String getName()
        {
            return "Test";
        }
    }

    private class TestHandler implements RequestHandler      ❷
    {
        public Response process(Request request) throws Exception
```

```
        {
            return new TestResponse();
        }
    }

    private class TestResponse implements Response          ❸
    {
        // empty
    }
[...]
```

❶ Set up a request object that returns a known name (`Test`).

❷ Implement a `TestHandler`. The interface calls for a `process` method, so you have to code that, too. You're not testing the `process` method right now, so you have it return a `TestResponse` object to satisfy the signature.

❸ Go ahead and define an empty `TestResponse` just so you have something to instantiate.

With the scaffolding from listing 3.5 in place, let's look at listing 3.6, which shows the test for adding a `RequestHandler`.

Listing 3.6 TestDefaultController.testAddHandler

```
public class TestDefaultController extends TestCase
{
[...]
    public void testAddHandler()          ❶
    {
        Request request = new TestRequest();                    ❷
        RequestHandler handler = new TestHandler();

        controller.addHandler(request, handler);          ❸

        RequestHandler handler2 = controller.getHandler(request);          ❹

        assertSame(handler2, handler);          ❺
    }
}
```

❶ Pick an obvious name for the test method.

❷ Instantiate your test objects.

❸ This code gets to the point of the test: `controller` (the object under test) adds the test handler. Note that the `DefaultController` object is instantiated by the `setUp` method (see listing 3.4).

❹ Read back the handler under a new variable name.
❺ Check to see if you get back the same object you put in.

> ***JUnit best practices: choose meaningful test method names***
>
> You must be able to understand what a method is testing by reading the name. A good rule is to start with the `test`*`Xxx`* naming scheme, where *`Xxx`* is the name of the method to test. As you add other tests against the same method, move to the `test`*`XxxYyy`* scheme, where *`Yyy`* describes how the tests differ.

Although it's very simple, this unit test confirms the key premise that the mechanism for storing and retrieving `RequestHandler` is alive and well. If `addHandler` or `getRequest` fails in the future, the test will quickly detect the problem.

As you create more tests like this, you will notice that you follow a pattern of steps:

1. Set up the test by placing the environment in a known state (create objects, acquire resources). The pre-test state is referred to as the *test fixture*.
2. Invoke the method under test.
3. Confirm the result, usually by calling one or more assert methods.

3.2.3 Processing a request

Let's look at testing the core purpose of the controller, processing a request. Because you know the routine, we'll just present the test in listing 3.7 and review it.

Listing 3.7 testProcessRequest

```
public class TestDefaultController extends TestCase
{
[...]
    public void testProcessRequest()                    ❶
    {
        Request request = new TestRequest();                    ❷
        RequestHandler handler = new TestHandler();
        controller.addHandler(request, handler);

        Response response = controller.processRequest(request);    ❸
        assertNotNull("Must not return a null response", response);    ❹

        assertEquals(TestResponse.class, response.getClass());    ❺
    }
}
```

❶ First give the test a simple, uniform name.
❷ Set up the test objects and add the test handler.
❸ Here the code diverges from listing 3.6 and calls the `processRequest` method.
❹ You verify that the returned `Response` object is not null. This is important because in ❺ you call the `getClass` method on the `Response` object. It will fail with a dreaded `NullPointerException` if the `Response` object is null. You use the `assertNotNull(String, Object)` signature so that if the test fails, the error displayed is meaningful and easy to understand. If you had used the `assertNotNull(Object)` signature, the JUnit runner would have displayed a stack trace showing an `AssertionFailedError` exception with no message, which would be more difficult to diagnose.
❺ Once again, compare the result of the test against the expected `TestResponse` class.

> ***JUnit best practices: explain the failure reason in assert calls***
>
> Whenever you use the `assertTrue`, `assertNotNull`, `assertNull`, and `assertFalse` methods, make sure you use the signature that takes a `String` as the first parameter. This parameter lets you provide a meaningful textual description that is displayed in the JUnit test runner if the assert fails. Not using this parameter makes it difficult to understand the reason for a failure when it happens.

Factorizing setup logic

Because both tests do the same type of setup, you can try moving that code into the JUnit `setUp` method. As you add more test methods, you may need to adjust what you do in the standard `setUp` method. For now, eliminating duplicate code as soon as possible helps you write more tests more quickly. Listing 3.8 shows the new and improved `TestDefaultController` class (changes are shown in bold).

Listing 3.8 TestDefaultController after some refactoring

```
package junitbook.sampling;

import junit.framework.TestCase;

public class TestDefaultController extends TestCase
{
    private DefaultController controller;
    private Request request;
    private RequestHandler handler;
```

```
    protected void setUp() throws Exception
    {
        controller = new DefaultController();
        request = new TestRequest();                          ❶
        handler = new TestHandler();
        controller.addHandler(request, handler);
    }

    private class TestRequest implements Request
    {
        // Same as in listing 3.5
    }

    private class TestHandler implements RequestHandler
    {
        // Same as in listing 3.5
     }

    private class TestResponse implements Response
    {
        // Same as in listing 3.5
    }

    public void testAddHandler()      ❷
    {
        RequestHandler handler2 = controller.getHandler(request);
        assertSame(handler2, handler);
    }

    public void testProcessRequest()      ❸
    {
        Response response = controller.processRequest(request);
        assertNotNull("Must not return a null response", response);
        assertEquals(TestResponse.class, response.getClass());
    }
}
```

❶ The instantiation of the test `Request` and `RequestHandler` objects is moved to `setUp`. This saves you repeating the same code in `testAddHandler` ❷ and `testProcessRequest` ❸.

DEFINITION *refactor*—To improve the design of existing code. For more about refactoring, see Martin Fowler's already-classic book.[4]

Note that you do *not* try to share the setup code by testing more than one operation in a test method, as shown in listing 3.9 (an anti-example).

[4] Martin Fowler, *Refactoring: Improving the Design of Existing Code* (Reading, MA: Addison-Wesley, 1999).

Listing 3.9 Do *not* combine test methods this way.

```
public class TestDefaultController extends TestCase
{
[...]
    public void testAddAndProcess()
    {
        Request request = new TestRequest();
        RequestHandler handler = new TestHandler();
        controller.addHandler(request, handler);

        RequestHandler handler2 = controller.getHandler(request);
        assertEquals(handler2,handler);

        // DO NOT COMBINE TEST METHODS THIS WAY
        Response response = controller.processRequest(request);
        assertNotNull("Must not return a null response", response);
        assertEquals(TestResponse.class, response.getClass());
    }
}
```

JUnit best practices: one unit test equals one testMethod

Do not try to cram several tests into one method. The result will be more complex test methods, which will become increasingly difficult to read and understand. Worse, the more logic you write in your test methods, the more risk there is that it will not work and will need debugging. This is a slippery slope that can end with writing tests to test your tests!

Unit tests give you confidence in a program by alerting you when something that had worked now fails. If you put more than one unit test in a method, it makes it more difficult to zoom in on exactly what went wrong. When tests share the same method, a failing test may leave the fixture in an unpredictable state. Other tests embedded in the method may not run, or may not run properly. Your picture of the test results will often be incomplete or even misleading.

Because all the test methods in a `TestCase` share the same fixture, and JUnit can now generate an automatic test suite (see chapter 2), it's really just as easy to place each unit test in its own method. If you need to use the same block of code in more than one test, extract it into a utility method that each test method can call. Better yet, if all methods can share the code, put it into the fixture.

For best results, your test methods should be as concise and focused as your domain methods.

Each test method must be as clear and focused as possible. This is why JUnit provides a `setUp` method: so you can share fixtures between tests without combining test methods.

3.2.4 Improving testProcessRequest

When we wrote the `testProcessRequest` method in listing 3.7, we wanted to confirm that the response returned is the expected response. The implementation confirms that the object returned is the object that we expected. But what we would really like to know is whether the response returned equals the expected response. The response could be a different class. What's important is whether the class identifies itself as the correct response.

The `assertSame` method confirms that both references are to the same object. The `assertEquals` method utilizes the equals method, inherited from the base Object class. To see if two different objects have the same identity, you need to provide your own definition of identity. For an object like a response, you can assign each response its own command token (or name).

The empty implementation of `TestResponse` didn't have a `name` property you can test. To get the test you want, you have to implement a little more of the `Response` class first. Listing 3.10 shows the enhanced `TestResponse` class.

Listing 3.10 A refactored TestResponse

```
public class TestDefaultController extends TestCase
{
[...]
    private class TestResponse implements Response
    {
        private static final String NAME = "Test";

        public String getName()
        {
            return NAME;
        }

        public boolean equals(Object object)
        {
            boolean result = false;
            if (object instanceof TestResponse)
            {
                result = ((TestResponse) object).getName().equals(
                    getName());
            }
            return result;
        }
```

```
        public int hashCode()
        {
            return NAME.hashCode();
        }
    }
[...]
```

Now that `TestResponse` has an identity (represented by `getName()`) and its own `equals` method, you can amend the test method:

```
    public void testProcessRequest()
    {
        Response response = controller.processRequest(request);
        assertNotNull("Must not return a null response", response);
        assertEquals(new TestResponse(), response);
    }
```

We have introduced the concept of identity in the `TestResponse` class for the purpose of the test. However, the tests are really telling you that this should have existed in the proper `Response` class. Thus you need to modify the `Response` interface as follows:

```
public interface Response
{
   String getName();
}
```

3.3 Testing exception-handling

So far, your tests have followed the main path of execution. If the behavior of one of your objects under test changes in an unexpected way, this type of test points to the root of the problem. In essence, you have been writing *diagnostic tests* that monitor the application's health.

But sometimes, bad things happen to healthy programs. Say an application needs to connect to a database. Your diagnostics may test whether you are following the database's API. If you open a connection but don't close it, a diagnostic can note that you have failed to meet the expectation that all connections are closed after use.

But what if a connection is not available? Maybe the connection pool is tapped out. Or, perhaps the database server is down. If the database server is configured properly and you have all the resources you need, this may never happen. But all resources are finite, and someday, instead of a connection, you may be

handed an exception. "Anything that can go wrong, will" (http://www.geocities.com/murphylawsite/).

If you are testing an application by hand, one way to test for this sort of thing is to turn off the database while the application is running. Forcing actual error conditions is an excellent way to test your disaster-recovery capability. Creating error conditions is also very time-consuming. Most of us cannot afford to do this several times a day—or even once a day. And many other error conditions are not easy to create by hand.

Testing the main path of execution is a good thing, and it needs to be done. But testing exception-handling can be even more important. If the main path does not work, your application will not work either (a condition you are likely to notice).

JUnit best practices: test anything that could possibly fail

Unit tests help ensure that your methods are keeping their API contracts with other methods. If the contract is based solely on other components' keeping their contracts, then there *may* not be any useful behavior for you to test. But if the method changes the parameter's or field's value in any way, then you are providing unique behavior that you should test. The method is no longer a simple go-between—it's a filtering or munging method with its own behavior that future changes could conceivably break. If a method is changed so it is not so simple anymore, then you should add a test *when that change takes place,* but not before. As the JUnit FAQ puts it, "The general philosophy is this: if it can't break *on its own*, it's too simple to break."

But what about things like JavaBean getters and setters? Well, that depends. If you are coding them by hand in a text editor, then yes, you might want to test them. It's surprisingly easy to miscode a setter in a way that the compiler won't catch. But if you are using an IDE that watches for such things, then your team might decide not to test simple JavaBean properties.

We are all too human, and often we tend to be sloppy when it comes to exception cases. Even textbooks scrimp on error-handling so as to simplify the examples. As a result, many otherwise great programs are not error-proofed before they go into production. If properly tested, an application should not expose a screen of death but should trap, log, and explain all errors gracefully.

3.3.1 Simulating exceptional conditions

The exceptional test case is where unit tests really shine. Unit tests can simulate exceptional conditions as easily as normal conditions. Other types of tests, like

functional and acceptance tests, work at the production level. Whether these tests encounter systemic errors is often a matter of happenstance. A unit test can produce exceptional conditions on demand.

During our original fit of inspired coding, we had the foresight to code an error handler into the base classes. As you saw back in listing 3.2, the `processRequest` method traps all exceptions and passes back a special error response instead:

```
try
{
    response = getHandler(request).process(request);
}
catch (Exception exception)
{
    response = new ErrorResponse(request, exception);
}
```

How do you simulate an exception to test whether your error handler works? To test handling a normal request, you created a `TestRequestHandler` that returned a `TestRequest` (see listing 3.5). To test the handling of error conditions, you can create a `TestExceptionHandler` that throws an exception instead, as shown in listing 3.11.

Listing 3.11 RequestHandler for exception cases

```
public class TestDefaultController extends TestCase
{
[...]
    private class TestExceptionHandler implements RequestHandler
    {
        public Response process(Request request) throws Exception
        {
            throw new Exception("error processing request");
        }
    }
}
```

This just leaves creating a test method that registers the handler and tries processing a request—for example, like the one shown in listing 3.12.

Listing 3.12 testProcessRequestAnswersErrorResponse, first iteration

```
public class TestDefaultController extends TestCase
{
[...]
    public void testProcessRequestAnswersErrorResponse()
    {
```

```
        TestRequest request = new TestRequest();                          ❶
        TestExceptionHandler handler = new TestExceptionHandler();
        controller.addHandler(request, handler);      ❷
        Response response = controller.processRequest(request);
        assertNotNull("Must not return a null response", response);       ❸
        assertEquals(ErrorResponse.class, response.getClass());
    }
}
```

❶ Create the request and handler objects.

❷ You reuse the `controller` object created by the default fixture (see listing 3.8).

❸ Test the outcome against your expectations.

But if you run this test through JUnit, you get a red bar! (See figure 3.1.) A quick look at the message tells you two things. First, you need to use a different name for the test request, because there is already a request named `Test` in the fixture. Second, you may need to add more exception-handling to the class so that a `RuntimeException` is not thrown in production.

As to the first item, you can try using the `request` object in the fixture instead of your own, but that fails with the same error. (Moral: Once you have a test, use it to explore alternative coding strategies.) You consider changing the fixture. If you remove from the fixture the code that registers a default `TestRequest` and `TestHandler`, you introduce duplication into the other test methods. Not good. Better to fix the `TestRequest` so it can be instantiated under different names. Listing 3.13 is the refactored result (changes from listing 3.11 and 3.12 are in bold).

Figure 3.1 Oops, red bar—time to add exception-handling!

Listing 3.13 testProcessRequestExceptionInHandler, fixed and refactored

```
public class TestDefaultController extends TestCase
{
[...]
    private class TestRequest implements Request
    {
        private static final String DEFAULT_NAME = "Test";      ❶
        private String name;

        public TestRequest(String name)      ❷
        {
            this.name = name;
        }

        public TestRequest()      ❸
        {
            this(DEFAULT_NAME);
        }

        public String getName()
        {
            return this.name;
        }
    }
[...]
    public void testProcessRequestAnswersErrorResponse()
    {
        TestRequest request = new TestRequest("testError");      ❹
        TestExceptionHandler handler = new TestExceptionHandler();
        controller.addHandler(request, handler);
        Response response = controller.processRequest(request);
        assertNotNull("Must not return a null response", response);
        assertEquals(ErrorResponse.class, response.getClass());
    }
}
```

❶ Introduce a member field to hold the request's name and set it to the previous version's default.

❷ Introduce a new constructor that lets you pass a name to the request, to override the default.

❸ Here you introduce an empty constructor, so existing calls will continue to work.

❹ Call the new constructor instead, so the exceptional request object does not conflict with the fixture.

Of course, if you added another test method that also used the exception handler, you might move its instantiation to the `setUp` method, to eliminate duplication.

> ***JUnit best practices: let the test improve the code***
>
> Writing unit tests often helps you write better code. The reason is simple: A test case is a user of your code. And, it is only when using code that you find its shortcomings. Thus, do not hesitate to listen to your tests and refactor your code so that it is easier to use. The practice of *Test-Driven Development (TDD)* relies on this principle. By writing the tests first, you develop your classes from the point of view of a user of your code. See chapter 4 for more about TDD.

But because the duplication hasn't happened yet, let's resist the urge to anticipate change, and let it stand. *("Don't anticipate, navigator!" the captain barked.)*

3.3.2 Testing for exceptions

During testing, you found that `addHandler` throws an undocumented `RuntimeException` if you try to register a request with a duplicate name. (By *undocumented*, we mean that it doesn't appear in the signature.) Looking at the code, you see that `getHandler` throws a `RuntimeException` if the request hasn't been registered.

Whether you *should* throw undocumented `RuntimeException` exceptions is a larger design issue. (You can make that a to-do for later study.) For now, let's write some tests that prove the methods will behave as designed.

Listing 3.14 shows two test methods that prove `addHandler` and `getHandler` will throw runtime exceptions when expected.

Listing 3.14 Testing methods that throw an exception

```
public class TestDefaultController extends TestCase
{
[...]
    public void testGetHandlerNotDefined()      ❶
    {
        TestRequest request = new TestRequest("testNotDefined");      ❷
        try
        {
            controller.getHandler(request);      ❸
            fail("An exception should be raised if the requested "
                + "handler has not been registered");      ❹
        }
        catch (RuntimeException expected)      ❺
        {
            assertTrue(true);      ❻
        }
    }

    public void testAddRequestDuplicateName()      ❼
```

```
    {
        TestRequest request = new TestRequest();
        TestHandler handler = new TestHandler();
        try
        {
            controller.addHandler(request, handler);      ❽
            fail("An exception should be raised if the default "
                + "TestRequest has already been registered");
        }
        catch (RuntimeException expected)
        {
            assertTrue(true);
        }
    }
}
```

❶ Give the test an obvious name. Because this test represents an exceptional case, append `NotDefined` to the standard `testGetHandler` prefix. Doing so keeps all the `getHandler` tests together and documents the purpose of each derivation.

❷ You create the `request` object for the test, also giving it an obvious name.

❸ Pass the (unregistered) request to the default `getHandler` method.

❹ Introduce the `fail` method, inherited from the `TestCase` superclass. If a test ever reaches a `fail` statement, the test (unsurprisingly) will fail, just as if an assertion had failed (essentially, `assertTrue(false)`). If the `getHandler` statement throws an exception, as you expect it will, the `fail` statement will not be reached.

❺ Execution proceeds to the `catch` statement, and the test is deemed a success.

❻ You clearly state that this is the expected success condition. Although this line is not necessary (because it always evaluates to true), we have found that it makes the test easier to read. For the same reason, at ❺ you name the exception variable `expected`.

❼ In the second test, you again use a descriptive name. (Also note that you do not combine tests, but write a separate test for each case.)

❽ You follow the same pattern as the first method:

1. Insert a statement that should throw an exception.
2. Follow it with a `fail` statement (in case the exception isn't thrown).
3. Catch the exception you expect, naming the exception `expected` so the reader can easily guess that the exception is expected!
4. Proceed normally.

JUnit best practices: make exception tests easy to read

Name the exception variable in the `catch` block `expected`. Doing so clearly tells readers that an exception is expected to make the test pass. It also helps to add an `assertTrue(true)` statement in the `catch` block to stress even further that this is the correct path.

The controller class is by no means done, but you have a respectable first iteration and a test suite proving that it works. Now you can commit the controller package, along with its tests, to the project's code repository and move on to the next task on your list.

JUnit best practices: let the test improve the code

An easy way to identify exceptional paths is to examine the different branches in the code you're testing. By *branches*, we mean the outcome of `if` clauses, `switch` statements, and `try/catch` blocks. When you start following these branches, sometimes you may find that testing each alternative is painful. If code is difficult to test, it is usually just as difficult to use. When testing indicates a poor design (called a *code smell*, http://c2.com/cgi/wiki?CodeSmell), you should stop and refactor the domain code.

In the case of too many branches, the solution is usually to split a larger method into several smaller methods.[5] Or, you may need to modify the class hierarchy to better represent the problem domain.[6] Other situations would call for different refactorings.

A test is your code's first "customer," and, as the maxim goes, "the customer is always right."

3.4 Setting up a project for testing

Because this chapter covers testing a fairly realistic component, let's finish up by looking at how you set up the controller package as part of a larger project. In chapter 1, you kept all the Java domain code and test code in the same folder. They were introductory tests on an example class, so this approach seemed simplest for everyone. In this chapter, you've begun to build real classes with real

[5] Fowler, *Refactoring*, "Extract Method."

[6] Ibid., "Extract Hierarchy."

tests, as you would for one of your own projects. Accordingly, you've set up the source code repository just like you would for a real project.

So far, you have only one test case. Mixing this in with the domain classes would not have been a big deal. But, experience tells us that soon you will have at least as many test classes as you have domain classes. Placing all of them in the same directory will begin to create file-management issues. It will become difficult to find the class you want to edit next.

Meanwhile, you want the test classes to be able to unit-test protected methods, so you want to keep everything in the same Java package. The solution? One package, two folders. Figure 3.2 shows a snapshot of how the directory structure looks in a popular integrated development environment (IDE).

This is the code for the "sampling" chapter, so we used `sampling` for the top-level project directory name (see appendix A). The IDE shows it as `junitbook-sampling`, because this is how we named the project. Under the `sampling` directory we created separate `java` and `test` folders. Under each of these, the actual package structure begins.

In this case, all of the code falls under the `junitbook.sampling` package. The working interfaces and classes go under `src/java/junitbook/sampling`; the classes we write for testing only go under the `src/test/junitbook/sampling` directory.

Beyond eliminating clutter, a "separate but equal" directory structure yields several other benefits. Right now, the only test class has the convenient `Test` prefix. Later you may need other helper classes to create more sophisticated tests.

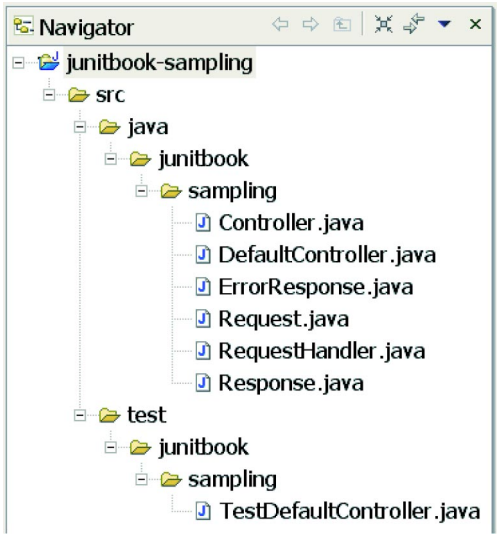

Figure 3.2
A "separate but equal" filing system keeps tests in the same package but in different directories.

These might include stubs, mock objects, and other helpers. It may not be convenient to prefix all of these classes with `Test`, and it becomes harder to tell the domain classes from the test classes.

Using a separate test folder also makes it easy to deliver a runtime jar with only the domain classes. And, it simplifies running all the tests automatically.

JUnit best practices: same package, separate directories

Put test classes in the same package as the class they test but in a parallel directory structure. You need tests in the same package to allow access to protected methods. You want tests in a separate directory to simplify file management and to clearly delineate test and domain classes.

3.5 Summary

In this chapter, we created a test case for a simple but complete application controller. Rather than test a single component, the test case examined how several components worked together. We started with a bootstrap test case that could be used with any class. Then we added new tests to `TestCase` one by one until all of the original components were under test.

We expect this package to grow, so we created a second source code directory for the test classes. Because the test and domain source directories are part of the same package, we can still test protected and package default members.

Knowing that even the best-laid plans go astray, we were careful to test the exception- and error-handling as thoroughly as the main path of execution. Along the way, we let the tests help us improve our initial design. At the end, we had a good start on the `Controller` class, and the tests to prove it!

In the next chapter, we will put unit testing in perspective with other types of tests that you need to perform to fully test your applications. We will also talk about how unit testing fits in the development life cycle.

4 Examining software tests

This chapter covers

- The need for unit tests
- Understanding the different types of tests
- Test coverage: how much is enough?
- Practicing Test-Driven Development
- Testing in the development cycle

A crash is when your competitor's program dies. When your program dies, it is an 'idiosyncrasy'. Frequently, crashes are followed with a message like 'ID 02'. 'ID' is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had.

—Guy Kawasaki

Earlier chapters in this book took a very pragmatic approach to designing and deploying unit tests. This chapter steps back and looks at the various types of software tests, the role they play in the application's life cycle, how to design for testability, and how to practice test-first development.

Why would you need to know all this? Because performing unit testing is not just something you do out of the blue. In order to be a good developer, you have to understand why you are doing it and why you are writing unit tests instead of (or to complement) functional, integration, or other kinds of tests. Once you understand why you are writing unit tests, then you need to know how far you should go and when you have enough tests. Testing is not an end goal.

Finally, we'll show you how Test-Driven Development (TDD) can substantially improve the quality and design of your application by placing unit tests at the center of the development process.

4.1 The need for unit tests

Writing unit tests is good. Understanding why you write them is even better! The main goal is to verify that your application works and to try to catch bugs early. Functional testing does that; however, unit tests are extremely powerful and versatile beasts that offer much more than simply verifying that the application works:

- They allow greater test coverage than functional tests. u
- They enable teamwork.
- They prevent regression and limit the need for debugging.
- They give us the courage to refactor.
- They improve the implementation design.
- They serve as the developer's documentation.
- They're fun!

4.1.1 Allowing greater test coverage

Functional tests are the first type of tests any application should have. If you had to choose between writing unit tests or functional tests, you should choose the latter. In our experience, functional tests are able to cover about 70% of the application code. If you wish to go further and provide more test coverage (see section 4.3), you need to write unit tests.

Unit tests can easily simulate error conditions, which is extremely difficult to do with functional tests (it's impossible in some instances). However, making a decision about your need for unit tests based solely on test coverage criteria is a mistake. Unit tests provide much more than just testing, as explained in the following sections.

4.1.2 Enabling teamwork

Imagine you are part of a team, working on some part of the overall application. Unit tests allow you to deliver quality code (tested code) without having to wait for all the other parts to be complete (you wouldn't dream of delivering code without it being tested, right?). On the other hand, functional tests are more coarse-grained and need the full application (or a good part of it) to be ready before you can test it.

4.1.3 Preventing regression and limiting debugging

A good unit-test suite gives you confidence that your code works. It also gives you the courage to modify your existing code, either for refactoring purposes or to add/modify new features. As a developer, there's no better feeling than knowing that someone is watching your back and will warn you if you break something.

A corollary is that a good suite of unit tests reduces the need to debug an application to find out what's failing. Whereas a functional test will tell you that a bug exists somewhere in the implementation of a use case, a unit test will tell you that a specific method is failing for a specific reason. You no longer need to spend hours trying to find the error.

JUnit best practices: refactor

Throughout the history of computer science, many great teachers have advocated iterative development. Nikolas Wirth, for example, who gave us the now-ancient languages Algol and Pascal, championed techniques like *stepwise refinement*.

For a time, these techniques seemed difficult to apply to larger, layered applications. Small changes can reverberate throughout a system. Project managers looked to up-front planning as a way to minimize change, but productivity remained low.

The rise of the xUnit framework has fueled the popularity of *agile methodologies* that once again advocate iterative development. Agile methodologists favor writing code in vertical slices to produce a fully working use case, as opposed to writing code in horizontal slices to provide services layer by layer.

When you design and write code for a single use case or functional chain, your design may be adequate for this feature, but it may not be adequate for the next feature. To retain a design across features, agile methodologies encourage *refactoring* to adapt the code base as needed.

But how do you ensure that refactoring, or improving the design of existing code, does not break the existing code? Answer: unit tests that tell you when code breaks. In short, unit tests give you the courage to refactor.

The agile methodologies try to lower project risks by providing the ability to cope with change. They allow and embrace change by standardizing on quick iterations and applying principles like YAGNI (You Ain't Gonna Need It) and The Simplest Thing That Could Possibly Work. But the foundation upon which all these principles rest is a solid bed of unit tests.

4.1.4 Enabling refactoring

Without unit tests, it is difficult to justify refactoring, because there is always a relatively high chance that you may break something. Why would you risk spending hours of debugging time (and putting the delivery at risk) only to improve the implementation design, change a variable name, and so on? Unit tests provide a safety net that gives you the courage to refactor.

4.1.5 Improving implementation design

Unit tests are a first-rate client of the code they test. They force the API under test to be flexible and to be unit-testable in isolation. You usually have to refactor your code under test to make it unit-testable (or use the TDD approach, which by definition spawns code that can be unit-tested; see section 4.4).

It is important to listen to your unit tests and be tuned to the melody they are singing. If a unit test is too long and unwieldy, it usually means the code under test has a design smell and should be refactored. If the code cannot be easily tested in isolation (see chapter 7), it usually means the code isn't flexible enough and should be refactored. Modifying runtime code so that it can be tested is absolutely normal.

4.1.6 Serving as developer documentation

Imagine you're trying to learn a new API. On one side is a 300-page document describing the API, and on the other side are some simple examples showing how to use it. Which would you choose to learn the API?

The power of examples is well known and doesn't need to be demonstrated. Unit tests are exactly this: samples that show how to use the API and how it behaves. As such, they make excellent developer documentation. Because unit tests must be kept in synch with the working code, unlike other forms of documentation, they must always be up to date.

Listing 4.1 illustrates how unit tests can help provide documentation. The `testTransferWithoutEnoughFunds()` method shows that an `AccountInsufficientFundsException` is thrown when an account transfer is performed without enough funds.

Listing 4.1 Unit tests as automatic documentation

```
import junit.framework.TestCase;

public class TestAccount extends TestCase
{
[...]
    public void testTranferWithoutEnoughFunds()
    {
        long balance = 1000;
        long amountToTransfer = 2000;
        Account credit = new Account(balance);
        Account debit = new Account();
        try
        {
            credit.transfer(debit, amountToTransfer);
```

```
                fail("The debited account doesn't have enough funds"
                    + " and an exception should have been thrown");
            }
            catch (AccountInsufficientFundsException expected)
            {
                assertTrue(true); // We get the exception as expected
            }
        }
    }
```

4.1.7 Having fun

Unit tests are addictive. Once you get hooked on the JUnit green bar, it's hard to do without. It gives you peace of mind: "The bar is green, and I'm happy because I know my code is OK." Alternating between red bar and green bar becomes a challenging, productive pastime. Like any good contest, there's a prize at the end: automated tests that, at the push of a button, tell you if everything still works.

You may be skeptical at first, but you once you are test-infected, it becomes difficult to write any piece of code without writing a test. (Kent Beck, king of test infection, said, "Any program feature without an automated test simply doesn't exist."[1]) The test runner's green bar, shown in figure 4.1, becomes a familiar friend. You miss it when it's gone. For a programmer, there's no greater pleasure than *knowing your code is of good quality and doing what it is supposed to do.*

To help keep the fun going, you can add metrics, such as showing the progress in a number of unit tests across iterations. Test-coverage reports (see section 4.3) showing the part of the code being exercised are another good strategy to keep up the momentum.

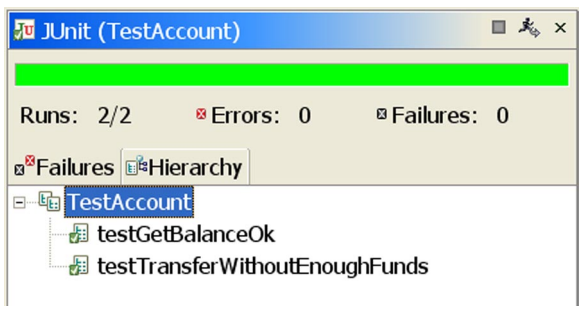

Figure 4.1
The famous green bar that appears when all tests pass

[1] Kent Beck, *Extreme Programming Explained: Embrace Change* (Reading, MA: Addison-Wesley, 1999), p. 57.

4.2 Different kinds of tests

Figure 4.2 outlines our five categories of software tests. There are other ways of categorizing software tests, but we find these most useful for the purposes of this text. Please note that this section is discussing *software tests in general*, not just the automated unit tests covered elsewhere in the book.

In figure 4.2, the outermost software tests are broadest in scope. The innermost software tests are narrowest in scope. As you move from the inner boxes to the outer boxes, the software tests get more and more functional and require that more and more of the application already be built.

Here, we'll first look at the general software test types. Then, we'll focus on the flavors of unit tests.

4.2.1 The four flavors of software tests

We've mentioned that unit tests focus on each distinct unit of work. But what about testing what happens when different units of work are combined into a workflow? Will the end result of the workflow do what you expect? How well will the application work when many people are using it at once? Will the application meet everyone's needs?

Each of these questions is answered by a different "flavor" of software test. For the purposes of this discussion, we can categorize software tests into four varieties:

- Integration tests
- Functional tests
- Stress/load tests
- Acceptance tests

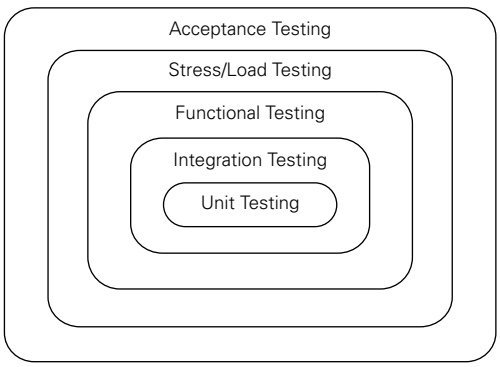

Figure 4.2
The five flavors of tests within the application life cycle

Let's look at each of the test types encompassing software testing, starting with the innermost after unit testing and working our way out.

Integration software testing

Individual unit tests are an essential quality control, but what happens when different units of work are combined into a workflow? Once you have the tests for a class up and running, the next step is to hook up with other methods and with other services. Examining the interaction between components, possibly when they run in their target environment, is the stuff of integration testing.

Because there are many things with which you might want to integrate, the word *integration* can mean different things under different circumstances. Some of these circumstances are laid out in table 4.1.

Table 4.1 Testing how objects, services, and subsystems interact

Circumstance	Description
How objects interact	The test instantiates one or more objects and calls methods on one object from another.
How services interact	Tests are run while the application is hosted within a servlet or EJB container, connected to a live database, or attached to any other external resource or device.
How subsystems interact	A layered application may have a front-end subsystem to handle the presentation and a back-end subsystem to execute the business logic. Tests can verify that a request passes through the front end and returns an appropriate response from the back end.

Just as more traffic collisions occur at intersections, the points where units interact are major contributors of software accidents. Ideally, integration tests should be defined before the units are coded. Being able to code to the test dramatically increases a programmer's ability to write well-behaved objects.

Functional software testing

Testing interactions between objects is essential. But will the end result be what you expect?

Functional tests examine the code at the boundary of its public API. In general, this amounts to testing the application use cases.

Functional tests are often combined with integration tests. You may have a secure web page that should only be accessed by authorized clients. If the client is not logged in, then trying to access the page should result in a redirect to the

login page. A functional unit test can examine this case by sending an HTTP request to the page to see whether a redirect (302) response code comes back.

Depending on the application, you can use several types of functional testing, as shown in table 4.2.

Table 4.2 Testing frameworks, GUIs, and subsystems

Circumstance	Description
Whether the application being built uses a framework	Functional testing within a framework focuses on testing the framework API that is used by users of the framework (end users or service providers).
Whether the application has a GUI	Functional testing of a GUI is about verifying that all features can be accessed and provide expected results. The tests are exercised directly on the GUI (which may in turn call several other components or a back end).
Whether the application is made of subsystems	A layered system tries to separate systems by roles. There may be a presentation subsystem, a business logic subsystem, and a data subsystem. Layering provides flexibility and the ability to access the back end with several different front ends. Each layer defines an API for other layers to use. Functional tests verify that the API contract is enforced.

Stress/load testing

It's important to have an application that functions correctly, but how well will the application perform when many people are using it at once? Most stress tests examine whether the application can process a large number of requests within a short period of time. Usually, this is done with special software like JMeter (http://jakarta.apache.org/jmeter), which can automatically send preprogrammed requests and track how quickly the application responds. Whether the response is correct is not usually tested. (That's why we have the other tests.) Figure 4.3 shows a JMeter throughput graph.

The stress tests are usually performed in a separate environment. A stress-test environment is generally more controlled than a typical development environment. It must be as close as possible to the production environment. If the production and stress-test environments are not very similar, then the tests are meaningless.

Other types of performance tests can be performed within the development environment. A profiler can look for bottlenecks in an application, which the developer can try to optimize. A number of Java profilers are available, and a search on Google will bring up several candidates.

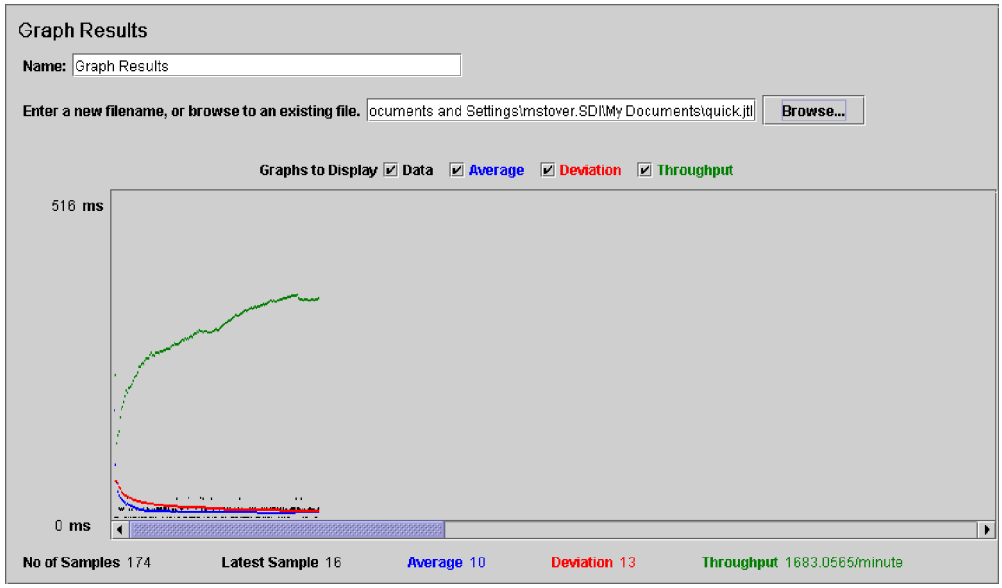

Figure 4.3 A JMeter throughput graph

From a testing perspective, using a profiler helps you to be proactive and can identify potential issues before the actual stress/load tests are performed. First you must be able to prove that a specific bottleneck exists, and then you must be able to prove that the bottleneck has been eliminated. A profiler is an essential tool in either case.

Unit tests can also help you profile an application as a natural part of development. JUnit extensions like JUnitPerf (http://www.clarkware.com/software/JUnitPerf.html) are available to help you create a suite of performance tests to match your unit tests. You might want to assert that a critical method never takes too long to execute. You can specify a threshold duration as part of the performance unit test. As the application is refactored, you will be alerted if any change causes the critical method to cross your threshold.

Listing 4.2 shows the source for a timed test, taken from the JUnitPerf distribution examples. This test ensures that the `ExampleTestCase` never takes more than one second to run.

Listing 4.2 Timed test example

```
package com.clarkware.junitperf;

import junit.framework.Test;
import junit.framework.TestSuite;

public class ExampleTimedTest {

    public static final long toleranceInMillis = 100;

    public static Test suite() {

        long maxElapsedTimeInMillis = 1000 + toleranceInMillis;

        Test testCase =
            new ExampleTestCase("testOneSecondResponse");
        Test timedTest =
            new TimedTest(testCase, maxElapsedTimeInMillis);

        TestSuite suite = new TestSuite();
        suite.addTest(timedTest);

        return suite;
    }

    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Acceptance software testing

It's important that an application perform well; but, in the end, the only question that matters is *Does the application meet the customer's needs?* Acceptance tests are the final sphere of tests. These tests are usually conducted directly by the customer or someone acting as the customer's proxy. Acceptance tests ensure that the application has met whatever goals the customer or stakeholder defined.

Acceptance tests are a superset of all other tests. Usually they start as functional and performance tests, but they may include subjective criteria like "ease of use" and "look and feel." Sometimes, the acceptance suite may include a subset of the tests run by the developers, the difference being that this time the tests are run by the customer or a QA team.

For more about using acceptance tests with an agile software methodology, visit the Wiki site regarding Ward Cunningham's *fit* framework (http://fit.c2.com/).

4.2.2 The three flavors of unit tests

The focus of this book is automatic unit tests used by programmers. *Unit testing*, as we use the term, concentrates on testing the code from the inside (*white box*