```
    {
        HttpServer server = new HttpServer();             ❶
        SocketListener listener = new SocketListener();   ❷
        listener.setPort(8080);
        server.addListener(listener);

        HttpContext context = new HttpContext();          ❸
        context.setContextPath("/");
        context.setResourceBase("./");
        context.addHandler(new ResourceHandler());        ❹
        server.addContext(context);                       ❸
        server.start();                                   ❺
    }
}
```

❶ Create the Jetty `HttpServer` object.

❷ Attach a listener on port 8080 to the `HttpServer` object so that it can receive HTTP requests.

❸ Create an `HttpContext` that processes the HTTP requests and passes them to the different handlers. You map the context to the root (/) URL with `setContextPath`. The `setResourceBase` method sets the document root from which to serve resources.

❹ Add a resource handler to the `HttpContext` in order to be able to serve files from the filesystem.

❺ Start the server.

6.3 Stubbing the web server's resources

Now that you know how to easily start and configure Jetty, let's focus on the HTTP connection sample unit test. You will write a first test that verifies you can call a valid URL and get its content.

6.3.1 Setting up the first stub test

To verify that the `WebClient` works with a valid URL, you need to start the Jetty server before the test, which you can implement in a `setUp` method inside a JUnit test case. You can also stop the server in a `tearDown` method. Listing 6.3 shows the code.

Listing 6.3 Skeleton of the first test to verify that the WebClient works with a valid URL

```
package junitbook.coarse.try1;

import java.net.URL;

import junit.framework.TestCase;

public class TestWebClientSkeleton extends TestCase
{
    protected void setUp()
    {
        // Start Jetty and configure it to return "It works" when
        // the http://localhost:8080/testGetContentOk URL is
        // called.
    }

    protected void tearDown()
    {
        // Stop Jetty.
    }

    public void testGetContentOk() throws Exception
    {
        WebClient client = new WebClient();

        String result = client.getContent(new URL(
             "http://localhost:8080/testGetContentOk"));

        assertEquals ("It works", result);
    }

}
```

In order to implement the `setUp` and `tearDown` methods, you have two solutions. You can prepare a static page containing the text *It works* that you put in your document root (controlled by the call to `context.setResourceBase(String)` in listing 6.2). Alternatively, you can configure Jetty to use your own custom `Handler` that returns the string directly instead of getting it from a file in the filesystem. This is a much more powerful technique, because it lets you unit-test the case when the remote HTTP server throws an error code at your `WebClient` client application.

Creating a Jetty handler

Listing 6.4 shows how to create a Jetty `Handler` that returns *It works*.

Listing 6.4 Create a Jetty Handler that returns *It works* when called

```
private class TestGetContentOkHandler extends AbstractHttpHandler      ❶
{
    public void handle(String pathInContext, String pathParams,
        HttpRequest request, HttpResponse response)
        throws IOException
    {
        OutputStream out = response.getOutputStream();        ❷
        ByteArrayISO8859Writer writer =
            new ByteArrayISO8859Writer();
        writer.write("It works");              ❸
        writer.flush();
        response.setIntField(HttpFields.__ContentLength,          ❹
            writer.size());
        writer.writeTo(out);
        out.flush();
        request.setHandled(true);      ❺
    }
```

❶ Handlers are easily created by extending the Jetty `AbstractHttpHandler` class, which defines a single `handle` method you need to implement. This method is called by Jetty when the incoming request is forwarded to your handler.

❷ Use the `ByteArrayISO8859Writer` class that Jetty provides to make it easy to send back your string in the HTTP response (❸).

❹ Set the response content length to be the length of the string you wrote to the output stream (this is required by Jetty), and then send the response.

❺ Tell Jetty that the request has been handled and does not need to be passed to any further handler.

Now that this handler is written, you can tell Jetty to use it by calling `context.addHandler(new TestGetContentOkHandler())`.

You are almost ready to run your test. The last issue to solve is the one involving `setUp/tearDown` methods. The solution shown in listing 6.3 isn't great, because it means the server will be started and stopped for every single test. Although Jetty is very fast, this process is still not necessary. A better solution is to start the server only once for all the tests. Fortunately, JUnit supports doing this with the notion of `TestSetup`, which lets you wrap a whole suite of tests and have global `setUp` and `tearDown` methods for the suite.

Isolating each test vs. performance considerations

In previous chapters, we went to great lengths to explain why each test should run in a clean environment (even to the extent of using a new classloader instance). However, sometimes there are other considerations to take into account. Performance is a typical one. In the present case, even if starting Jetty takes only 1 second, once you have 300 tests, it will add an overtime of 300 seconds (5 minutes). Test suites that take a long time to execute are a handicap; you will be tempted not to execute them often, which negates the regression feature of unit testing. You must be aware of this tradeoff. Depending on the situation, you may choose to have longer-running tests that execute in a clean environment, or instead tune the tests for performance by reusing some parts of the environment. In the example at hand, you use different handlers for different tests, and you can be pretty confident they will not interfere with one another. Thus, going for faster running tests is probably the best option.

Starting and stopping Jetty once per test suite

Listing 6.5 shows the `TestSetup` with the methods to start and stop Jetty. It also configures Jetty with the `TestGetContentOkHandler` from listing 6.4.

Listing 6.5 Extending TestSetup to configure, start, and stop Jetty once per test suite

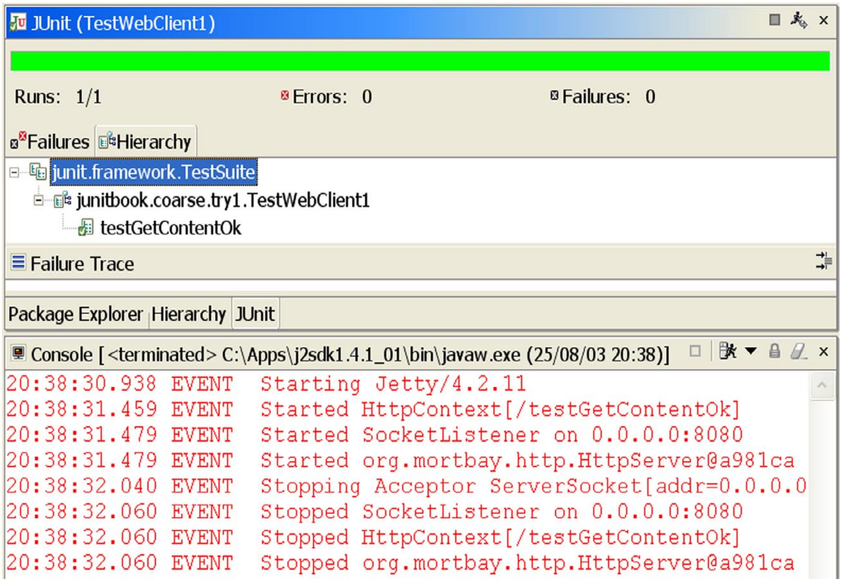
```
package junitbook.coarse.try1;

import java.io.IOException;
import java.io.OutputStream;

import org.mortbay.http.HttpContext;
import org.mortbay.http.HttpFields;
import org.mortbay.http.HttpRequest;
import org.mortbay.http.HttpResponse;
import org.mortbay.http.HttpServer;
import org.mortbay.http.SocketListener;
import org.mortbay.http.handler.AbstractHttpHandler;
import org.mortbay.util.ByteArrayISO8859Writer;

import junit.extensions.TestSetup;
import junit.framework.Test;

public class TestWebClientSetup1 extends TestSetup      ❶ Extend TestSetup to provide global setUp and tearDown
{
    protected static HttpServer server;                ❷ Save Jetty server object for use in tearDown

    public TestWebClientSetup1(Test suite)
    {
        super(suite);
    }

    protected void setUp() throws Exception           ❸ Configure and start Jetty
    {
        server = new HttpServer();
        SocketListener listener = new SocketListener();
        listener.setPort(8080);
        server.addListener(listener);

        HttpContext context1 = new HttpContext();
        context1.setContextPath("/testGetContentOk");
        context1.addHandler(new TestGetContentOkHandler());
        server.addContext(context1);

        server.start();
    }

    protected void tearDown() throws Exception          ❹ Stop Jetty
    {
        server.stop();
    }

    private class TestGetContentOkHandler extends       ❺ Implement Jetty
        AbstractHttpHandler                                Handler as an
    {                                                      inner class
        public void handle(String pathInContext, String pathParams,
            HttpRequest request, HttpResponse response)
            throws IOException
        {
            OutputStream out = response.getOutputStream();
            ByteArrayISO8859Writer writer =
                new ByteArrayISO8859Writer();
            writer.write("It works");
            writer.flush();
            response.setIntField(HttpFields.__ContentLength,
                writer.size());
            writer.writeTo(out);
            out.flush();
            request.setHandled(true);
        }
    }
}
```

❶ By extending `TestSetup`, you need to implement a `setUp` method and a `tearDown` method. They are called at the beginning of the execution of the test suite and at its end.

❸ In the `setUp` method, you start Jetty and save a reference to the Jetty `HttpServer` object (❷) for later use in the `tearDown` method.

❹ In the `tearDown` method, you stop the running server.

❺ Implement the content handler that returns *It works* when called as an inner class of `TestSetup`. (You use an inner class because the handler is strongly linked to the



Console [<terminated> C:\Apps\j2sdk1.4.1_01\bin\javaw.exe (25/08/03 20:38)]

```
20:38:30.938 EVENT  Starting Jetty/4.2.11
20:38:31.459 EVENT  Started HttpContext[/testGetContentOk]
20:38:31.479 EVENT  Started SocketListener on 0.0.0.0:8080
20:38:31.479 EVENT  Started org.mortbay.http.HttpServer@a981ca
20:38:32.040 EVENT  Stopping Acceptor ServerSocket[addr=0.0.0.0
20:38:32.060 EVENT  Stopped SocketListener on 0.0.0.0:8080
20:38:32.060 EVENT  Stopped HttpContext[/testGetContentOk]
20:38:32.060 EVENT  Stopped org.mortbay.http.HttpServer@a981ca
```

Figure 6.4 Result of the first working test using a Jetty stub. Jetty is automatically started before the test and stopped at the end.

Jetty configuration that is performed in the `setUp` method. However, you can implement it as a separate class, which may become cleaner should the number of handlers increase dramatically.)

Writing the Test class

The `Test` class can now be easily written using this `TestWebClientSetup1` class, as demonstrated in listing 6.6. When you execute it, you now get the result shown in figure 6.4.

Listing 6.6 Putting it all together

```
package junitbook.coarse.try1;

import java.net.URL;

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

public class TestWebClient1 extends TestCase
{
    public static Test suite()
    {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestWebClient1.class);
        return new TestWebClientSetup1(suite);
    }

    public void testGetContentOk() throws Exception
    {
        WebClient client = new WebClient();

        String result = client.getContent(new URL(
            "http://localhost:8080/testGetContentOk"));

        assertEquals("It works", result);
    }

}
```

The test class has become quite simple, and you have delegated all setup work in `TestWebClientSetup1`.

6.3.2 Testing for failure conditions

Now that you have the first test working, let's see how to test for server failure conditions. The `WebClient.getContent(URL)` method returns a `null` value when a failure happens. You need to test for this possibility, too. With the infrastructure you have put in place, you simply need to create a new Jetty `Handler` class that returns an error code and register it in the `TestWebClientSetup1.setUp` method.

Let's add a test for an invalid URL—that is, a URL pointing to a file that does not exist. This case is quite easy, because Jetty already provides a `NotFoundHandler` handler class for that purpose. You only need to modify the `TestWebClientSetup1.setUp` method in the following way (changes are in bold):

```
protected void setUp() throws Exception
{
    server = new HttpServer();
    SocketListenerlistener = new SocketListener();
    listener.setPort(8080);
    server.addListener(listener);

    HttpContext context1 = new HttpContext();
    context1.setContextPath("/testGetContentOk");
    context1.addHandler(new TestGetContentOkHandler());
    server.addContext(context1);

    HttpContext context2 = new HttpContext();
    context2.setContextPath("/testGetContentNotFound");
    context2.addHandler(new NotFoundHandler());
    server.addContext(context2);

    server.start();
}
```

Adding a new test in `TestWebClient1` is also a breeze:

```
public void testGetContentNotFound() throws Exception
{
    WebClient client = new WebClient();

    String result = client.getContent(new URL(
        "http://localhost:8080/testGetContentNotFound"));

    assertNull(result);
}
```

In a similar fashion, you can easily add a test to simulate the server having trouble. Returning a 5*XX* HTTP response code indicates this problem. Write a new Jetty `Handler` class, as follows, and register it in `TestWebClientSetup1.setUp`:

```
public class TestGetContentServerErrorHandler extends
    AbstractHttpHandler
{
    public void handle(String pathInContext, String pathParams,
        HttpRequest request, HttpResponse response)
        throws IOException
    {
        response.sendError(HttpResponse.__503_Service_Unavailable);
    }
}
```

Now, that's quite nice. A test like this would be very difficult to perform if you did not choose an embeddable web server.

6.3.3 Reviewing the first stub test

You have been able to fully unit-test the `getContent` method in isolation by stubbing the web resource. What have you really tested? What kind of test have you achieved? Actually, you have done something quite powerful: You have unit-tested the method, but at the same time you have executed an integration test. In addition, not only have you tested the code logic, you have also tested the connection part that is outside the code (it uses the JDK's `HttpURLConnection` class).

However, this approach has a few drawbacks. The major one is that it is complex. It took me about four hours from start to finish, including getting enough Jetty knowledge to set it up correctly (I had no prior knowledge of Jetty). In some instances, I also had to debug my stubs to get them to work. There is a very dangerous line that you should not cross: The stub must stay simple and not become a full-fledged application that requires tests and maintenance.

Moreover, in the specific example, you need a web server—but another stub will be different and will need a different setup. Experience helps, but different cases usually require different stubbing solutions.

The example tests are nice because you can both unit-test the code and perform some integration tests at the same time. However, this functionality comes at the cost of complexity. There are more lightweight solutions that focus on unit-testing the code but without performing the integration tests. The rationale is that integration tests are very much needed but could be run in a separate suite of tests or as part of functional tests.

In the next section, we will look at another solution that can still be qualified as stubbing. It is simpler in the sense that it does not require you to stub a whole web server. It brings you one step closer to the mock-object strategy, which is described in the following chapter.

6.4 Stubbing the connection

So far, you have stubbed the web server's resources. What about stubbing the HTTP connection, instead? Doing so will prevent you from effectively testing the connection, but that's fine because it isn't your real goal at this point. You are really interested in testing your code logic in isolation. Functional or integration tests will test the connection at a later stage.

When it comes to stubbing the connection without changing your code, you are quite lucky because the JDK's `URL` and `HttpURLConnection` classes let you plug in custom protocol handlers to handle any kind of communication protocol. You can have any call to the `HttpURLConnection` class be redirected to your own test class, which will return whatever is needed for the test.

6.4.1 Producing a custom URL protocol handler

To implement a custom URL protocol handler, you need to call the following JDK method and pass it a custom `URLStreamHandlerFactory` object:

```
java.net.URL.setURLStreamHandlerFactory(
    java.net.URLStreamHandlerFactory);
```

Whenever a `URL.openConnection` method is called, the `URLStreamHandlerFactory` class is called to return a `URLStreamHandler` object. Listing 6.7 shows the code to perform this feat. The idea is to call the static `setURLStreamHandlerFactory` method in the JUnit `setUp` method. (A better implementation would use a `TestSetup` class, as shown in the previous section, so that it is performed only once during the whole test suite execution.)

Listing 6.7 Providing custom stream handler classes for testing

```
package junitbook.coarse.try2;

import junit.framework.TestCase;

import java.net.URL;
import java.net.URLStreamHandlerFactory;
import java.net.URLStreamHandler;
import java.net.URLConnection;
import java.io.IOException;

public class TestWebClient extends TestCase
{
    protected void setUp()
    {
        URL.setURLStreamHandlerFactory(
            new StubStreamHandlerFactory());
    }

    private class StubStreamHandlerFactory implements
        URLStreamHandlerFactory
    {
        public URLStreamHandler createURLStreamHandler(
            String protocol)
        {
            return new StubHttpURLStreamHandler();
        }
    }

    private class StubHttpURLStreamHandler extends
        URLStreamHandler
    {
        protected URLConnection openConnection(URL url)
            throws IOException
        {
            return new StubHttpURLConnection(url);
        }
    }

    public void testGetContentOk() throws Exception
    {
        WebClient client = new WebClient();

        String result = client.getContent(
            new URL("http://localhost"));

        assertEquals("It works", result);
    }
}
```

Tell URL class to use factory to handle connections

1 Route all connections to HTTP handler

2 Provide handler that returns stubbed HttpURL-Connection class

Test that exercises WebClient class

You have to use several inner classes (❶ and ❷) to be able to pass the `StubHttpURLConnection` class. You could also use anonymous inner classes for conciseness, but that approach would make the code more complex to read. Note that you haven't written the `StubHttpURLConnection` class yet, which is the topic of the next section.

6.4.2 Creating a JDK HttpURLConnection stub

The last step is to create a stub implementation of the `HttpURLConnection` class so you can return any value you want for the test. Listing 6.8 shows a simple implementation that returns the string *It works* as a stream to the caller.

Listing 6.8 Stubbed HttpURLConnection class

```
package junitbook.coarse.try2;

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

public class StubHttpURLConnection extends HttpURLConnection
{
    private boolean isInput = true;

    protected StubHttpURLConnection(URL url)
    {
        super(url);
    }

    public InputStream getInputStream() throws IOException
    {
        if (!isInput)      ❶ Stub logic
        {
            throw new ProtocolException(
                "Cannot read from URLConnection"
                + " if doInput=false (call setDoInput(true))");
        }

        ByteArrayInputStream bais = new ByteArrayInputStream(
            new String("It works").getBytes());

        return bais;
    }

    public void disconnect()
    {
    }
```

HttpURLConnection has no interface; extend it and override its methods

Override getInputStream method to return a string

Return expected string as a Stream

```
        public void connect() throws IOException
        {
        }

        public boolean usingProxy()
        {
            return false;
        }
    }
```

HttpURLConnection does not provide an interface, so you have to extend it and override the methods you want to stub. In this stub, you provide an implementation for the getInputStream method because it is the only method used by your code under test. Should the code to test use more APIs from HttpURLConnection, you would also need to stub these additional methods. This is where the code would become more complex—you would need to completely reproduce the same behavior as the real HttpURLConnection. For example, at ❶, you test that if setDoInput(false) has been called in the code under test, then a call to the getInputStream method returns a ProtocolException. (This is the behavior of HttpURLConnection.) Fortunately, in most cases, you only need to stub a few methods and not the whole API.

6.4.3 Running the test

Let's run the TestWebClient test, which uses the StubHttpURLConnection. Figure 6.5 shows the result.

As you can see, it is much easier to stub the connection than to stub the web resource. This approach does not bring the same level of testing (you are not

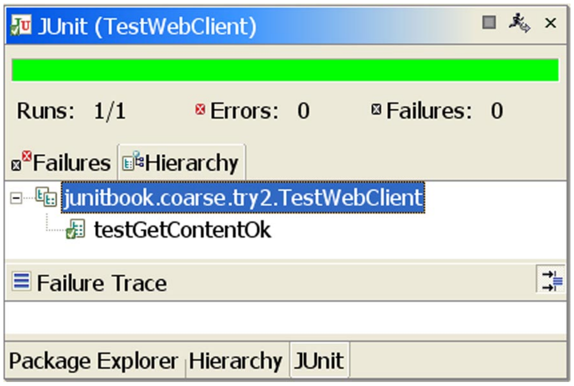


Figure 6.5 Result of executing TestWebClient (which uses the StubHttpURLConnection)

performing integration tests), but it enables you to more easily write a focused unit test for the `WebClient` code logic.

6.5 Summary

In this chapter, we have demonstrated how using a stub has helped us unit-test code accessing a remote web server using the JDK's `HttpURLConnection` API. In particular, we have shown how to stub the remote web server by using Jetty. Jetty's embeddable nature lets you concentrate on stubbing only the Jetty HTTP request handler, instead of having to stub the whole container. We also demonstrated a more lightweight stub by writing it for the JDK's `HttpURLConnection` implementation.

The next chapter will demonstrate a technique called *mock objects* that allows fine-grained unit testing, is completely generic, and (best of all) forces you to write good code. Although stubs are very useful in some cases, they're more a vestige of the past, when the general consensus was that tests should be a separate activity and should not modify the code. The new mock-objects strategy not only allows modification of code, it favors it. Using mock objects is a unit-testing strategy, but it is more than that: It is a completely new way of writing code.

7 Testing in isolation with mock objects

This chapter covers

- Introducing and demonstrating mock objects
- Performing different refactorings
- Using mock objects to verify API contracts on collaborating classes
- Practicing on an HTTP connection sample application

The secret of success is sincerity. Once you can fake that you've got it made.
—Jean Giraudoux

Unit-testing each method in isolation from the other methods or the environment is certainly a nice goal. But how do you perform this feat? You saw in chapter 6 how the stubbing technique lets you unit-test portions of code by isolating them from the environment (for example, by stubbing a web server, the filesystem, a database, and so on). But what about fine-grained isolation like being able to isolate a method call to another class? Is that possible? Can you achieve this without deploying huge amounts of energy that would negate the benefits of having tests?

The answer is, "Yes! It is possible." The technique is called *mock objects*. Tim Mackinnon, Steve Freeman, and Philip Craig first presented the mock objects concept at XP2000. The mock-objects strategy allows you to unit-test at the finest possible level and develop method by method, while providing you with unit tests for each method.

7.1 Introducing mock objects

Testing in isolation offers strong benefits, such as the ability to test code that has not yet been written (as long as you at least have an interface to work with). In addition, testing in isolation helps teams unit-test one part of the code without waiting for all the other parts.

But perhaps the biggest advantage is the ability to write focused tests that test only a single method, without side effects resulting from other objects being called from the method under test. Small is beautiful. Writing small, focused tests is a tremendous help; small tests are easy to understand and do not break when other parts of the code are changed. Remember that one of the benefits of having a suite of unit tests is the courage it gives you to refactor mercilessly—the unit tests act as a safeguard against regression. If you have large tests and your refactoring introduces a bug, several tests will fail; that result will tell you that there is a bug somewhere, but you won't know where. With fine-grained tests, potentially fewer tests will be affected, and they will provide precise messages that pinpoint the exact cause of the breakage.

Mock objects (or *mocks* for short) are perfectly suited for testing a portion of code logic in isolation from the rest of the code. Mocks replace the objects with which your methods under test collaborate, thus offering a layer of isolation. In that sense, they are similar to stubs. However, this is where the similarity ends, because mocks do not implement any logic: They are empty shells that provide methods to let the tests control the behavior of all the business methods of the faked class.

DEFINITION *mock object*—A *mock object* (or *mock* for short) is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests.

We will discuss when to use mock objects in section 7.6 at the end of this chapter, after we show them in action on some examples.

7.2 Mock tasting: a simple example

Let's taste our first mock! Imagine a very simple use case where you want to be able to make a bank transfer from one account to another (figure 7.1 and listings 7.1–7.3).

The `AccountService` class offers services related to `Accounts` and uses the `AccountManager` to persist data to the database (using JDBC, for example). The service that interests us is materialized by the `AccountService.transfer` method, which makes the transfer. Without mocks, testing the `AccountService.transfer` behavior would imply setting up a database, presetting it with test data, deploying the code inside the container (J2EE application server, for example), and so forth. Although this process is required to ensure the application works end to end, it is too much work when you want to unit-test only your code logic.

Listing 7.1 presents a very simple `Account` object with two properties: an account ID and a balance.

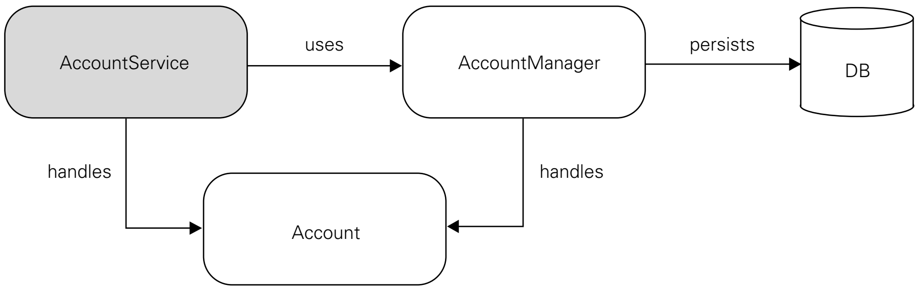


Figure 7.1 In this simple bank account example, you will use a mock object to test an account transfer method.

Listing 7.1 Account.java

```
package junitbook.fine.tasting;

public class Account
{
    private String accountId;
    private long balance;

    public Account(String accountId, long initialBalance)
    {
        this.accountId = accountId;
        this.balance = initialBalance;
    }

    public void debit(long amount)

    {
        this.balance -= amount;
    }

    public void credit(long amount)

    {
        this.balance += amount;
    }

    public long getBalance()

    {
        return this.balance;
    }
}
```

Listing 7.2 introduces the `AccountManager` interface, whose goal is to manage the life cycle and persistence of `Account` objects. (You are limited to finding accounts by ID and updating accounts.)

Listing 7.2 AccountManager.java

```
package junitbook.fine.tasting;

public interface AccountManager
{
    Account findAccountForUser(String userId);

    void updateAccount(Account account);

}
```

Listing 7.3 shows the `transfer` method for transferring money between two accounts. It uses the `AccountManager` interface you previously defined to find the debit and credit accounts by ID and to update them.

Listing 7.3 AccountService.java

```
package junitbook.fine.tasting;

public class AccountService
{
    private AccountManager accountManager;

    public void setAccountManager(AccountManager manager)
    {
        this.accountManager = manager;
    }

    public void transfer(String senderId, String beneficiaryId,
        long amount)
    {
        Account sender =
            this.accountManager.findAccountForUser(senderId);
        Account beneficiary =
            this.accountManager.findAccountForUser(beneficiaryId);

        sender.debit(amount);
        beneficiary.credit(amount);

        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(beneficiary);
    }
}
```

You want to be able to unit-test the `AccountService.transfer` behavior. For that purpose, you use a mock implementation of the `AccountManager` interface (listing 7.4). You do this because the transfer method is using this interface, and you need to test it in isolation.

Listing 7.4 MockAccountManager.java

```
package junitbook.fine.tasting;

import java.util.Hashtable;

public class MockAccountManager implements AccountManager
{
    private Hashtable accounts = new Hashtable();

    public void addAccount(String userId, Account account)           ❶
    {
        this.accounts.put(userId, account);         ❶
    }

    public Account findAccountForUser(String userId)          ❷
    {
        return (Account) this.accounts.get(userId);
    }

    public void updateAccount(Account account)        ❸
    {
        // do nothing
    }
}
```

❶ The `addAccount` method uses an instance variable to hold the values to return. Because you have several account objects that you want to be able to return, you store the `Account` objects to return in a `Hashtable`. This makes the mock generic and able to support different test cases: One test could set up the mock with one account, another test could set it up with two accounts or more, and so forth.

❷ `addAccount` tells the `findAccountForUser` method what to return when called.

❸ The `updateAccount` method updates an account but does not return any value. Thus you simply do nothing. When it is called by the `transfer` method, it will do nothing, as if the account had been correctly updated.

JUnit best practices: don't write business logic in mock objects

The single most important point to consider when writing a mock is that it should not have any business logic. It must be a dumb object that only does what the test tells it to do. In other words, it is purely driven by the tests. This characteristic is exactly the opposite of stubs, which contain all the logic (see chapter 6).

There are two nice corollaries. First, mock objects can be easily generated, as you will see in following chapters. Second, because mock objects are empty shells, they are too simple to break and do not need testing themselves.

You are now ready to write a unit test for `AccountService.transfer`. Listing 7.5 shows a typical test using a mock.

Listing 7.5 Testing transfer with MockAccountManager

```
package junitbook.fine.tasting;

import junit.framework.TestCase;

public class TestAccountService extends TestCase
{
    public void testTransferOk()
    {
        MockAccountManager mockAccountManager =                  ❶
            new MockAccountManager();
        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);

        mockAccountManager.addAccount("1", senderAccount);
        mockAccountManager.addAccount("2", beneficiaryAccount);

        AccountService accountService = new AccountService();

        accountService.setAccountManager(mockAccountManager);

        accountService.transfer("1", "2", 50);       ❷

        assertEquals(150, senderAccount.getBalance());           ❸
        assertEquals(150, beneficiaryAccount.getBalance());
    }
}
```

As usual, a test has three steps: the test setup (❶), the test execution (❷), and the verification of the result (❸). During the test setup, you create the `MockAccountManager` object and define what it should return when called for the two accounts you manipulate (the sender and beneficiary accounts). You have succeeded in testing the `AccountService` code in isolation of the other domain object, `AccountManager`, which in this case did not exist, but which in real life could have been implemented using JDBC.

JUnit best practices: only test what can possibly break

You may have noticed that you did not mock the `Account` class. The reason is that this data-access object class does not need to be mocked—it does not depend on the environment, and it's very simple. Your other tests use the `Account` object, so they test it indirectly. If it failed to operate correctly, the tests that rely on `Account` would fail and alert you to the problem.

At this point in the chapter, you should have a reasonably good understanding of what a mock is. In the next section, we will show you that writing unit tests with mocks leads to refactoring your code under test—and that this process is a good thing!

7.3 Using mock objects as a refactoring technique

Some people used to say that unit tests should be totally transparent to your code under test, and that you should not change runtime code in order to simplify testing. *This is wrong!* Unit tests are first-class users of the runtime code and deserve the same consideration as any other user. If your code is too inflexible for the tests to use, then you should correct the code.

For example, what do you think of the following piece of code?

```
[...]
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
[...]

public class DefaultAccountManager implements AccountManager
{
    private static final Log LOGGER =
        LogFactory.getLog(AccountManager.class);                  ❶

    public Account findAccountForUser(String userId)
    {
        LOGGER.debug("Getting account for user [" + userId + "]");
        ResourceBundle bundle =
           PropertyResourceBundle.getBundle("technical");              ❷
        String sql = bundle.getString("FIND_ACCOUNT_FOR_USER");

        // Some code logic to load a user account using JDBC
        [...]
    }
[...]
}
```

❶ You get a `Log` object using a `LogFactory` that creates it.

❷ Use a `PropertyResourceBundle` to retrieve an SQL command.

Does the code look fine to you? We can see two issues, both of which relate to code flexibility and the ability to resist change. The first problem is that it is not possible to decide to use a different `Log` object, as it is created inside the class. For testing, for example, you probably want to use a `Log` that does nothing, but you can't.

As a general rule, a class like this should be able to use whatever `Log` it is given. The goal of this class is not to create loggers but to perform some JDBC logic.

The same remark applies to the use of `PropertyResourceBundle`. It may sound OK right now, but what happens if you decide to use XML to store the configuration? Again, it should not be the goal of this class to decide what implementation to use.

An effective design strategy is to pass to an object any other object that is outside its immediate business logic. The choice of peripheral objects can be controlled by someone higher in the calling chain. Ultimately, as you move up in the calling layers, the decision to use a given logger or configuration should be pushed to the top level. This strategy provides the best possible code flexibility and ability to cope with changes. And, as we all know, change is the only constant.

7.3.1 Easy refactoring

Refactoring all code so that domain objects are passed around can be time-consuming. You may not be ready to refactor the whole application just to be able to write a unit test. Fortunately, there is an easy refactoring technique that lets you keep the same interface for your code but allows it to be passed domain objects that it should not create. As a proof, let's see how the refactored `DefaultAccountManager` class could look (modifications are shown in bold):

```
public class DefaultAccountManager implements AccountManager
{
    private Log logger;
    private Configuration configuration;

    public DefaultAccountManager()
    {
        this(LogFactory.getLog(DefaultAccountManager.class),
            new DefaultConfiguration("technical"));
    }

    public DefaultAccountManager(Log logger,
        Configuration configuration)
    {
        this.logger = logger;
        this.configuration = configuration;
    }

    public Account findAccountForUser(String userId)
    {
        this.logger.debug("Getting account for user ["
            + userId + "]");
        this.configuration.getSQL("FIND_ACCOUNT_FOR_USER");

        // Some code logic to load a user account using JDBC
        [...]
    }
[...]
}
```

1 **Log and Configuration are both interfaces**

Notice that at ❶, you swap the `PropertyResourceBundle` class from the previous listing in favor of a new `Configuration` interface. This makes the code more flexible because it introduces an interface (which will be easy to mock), and the implementation of the `Configuration` interface can be anything you want (including using resource bundles). The design is better now because you can use and reuse the `DefaultAccountManager` class with any implementation of the `Log` and `Configuration` interfaces (if you use the constructor that takes two parameters). The class can be controlled from the outside (by its caller). Meanwhile, you have not broken the existing interface, because you have only added a new constructor. You kept the original default constructor that still initializes the `logger` and `configuration` field members with default implementations.

With this refactoring, you have provided a trapdoor for controlling the domain objects from your tests. You retain backward compatibility and pave an easy refactoring path for the future. Calling classes can start using the new constructor at their own pace.

Should you worry about introducing trapdoors to make your code easier to test? Here's how Extreme Programming guru Ron Jeffries explains it:

> *My car has a diagnostic port and an oil dipstick. There is an inspection port on the side of my furnace and on the front of my oven. My pen cartridges are transparent so I can see if there is ink left.*
>
> *And if I find it useful to add a method to a class to enable me to test it, I do so. It happens once in a while, for example in classes with easy interfaces and complex inner function (probably starting to want an Extract Class).*
>
> *I just give the class what I understand of what it wants, and keep an eye on it to see what it wants next.*[1]

7.3.2 Allowing more flexible code

What we have described in section 7.3.1 is a well-known pattern called Inversion of Control (IOC). The main idea is to externalize all domain objects from outside the class/method and pass everything to it. Instead of the class creating object instances, the instances are passed to the class (usually through interfaces).

In the example, it means passing `Log` and `Configuration` objects to the `DefaultAccountManager` class. `DefaultAccountManager` has no clue what instances are passed to it or how they were constructed. It just knows they implement the `Log` and `Configuration` interfaces.

[1] Ron Jeffries, on the TestDrivenDevelopment mailing list: http://groups.yahoo.com/group/testdrivendevelopment/message/3914.

Design patterns in action: Inversion of Control (IOC)

Applying the IOC pattern to a class means removing the creation of all object instances for which this class is not directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It becomes the responsibility of the calling code to correctly set these domain objects on the called class.[2]

IOC makes unit testing a breeze. To prove the point, let's see how easily you can now write a test for the `findAccountByUser` method:

```
    public void testFindAccountByUser()
    {
        MockLog logger = new MockLog();    ❶
        MockConfiguration configuration = new MockConfiguration();
       configuration.setSQL("SELECT * [...]");                      ❷

        DefaultAccountManager am =
            new DefaultAccountManager(logger, configuration);     ❸

        Account account = am.findAccountForUser("1234");

        // Perform asserts here
        [...]
    }
}
```

❶ Use a mock logger that implements the `Log` interface but does nothing.

❷ Create a `MockConfiguration` instance and set it up to return a given SQL query when `Configuration.getSQL` is called.

❸ Create the instance of `DefaultAccountManager` that you will test, passing to it the `Log` and `Configuration` instances.

You have been able to completely control your logging and configuration behavior from outside the code to test, in the test code. As a result, your code is more flexible and allows for any logging and configuration implementation to be used. You will see more of these code refactorings in this chapter and later ones.

One last point to note is that if you write your test first, you will automatically design your code to be flexible. Flexibility is a key point when writing a unit test. If you test first, you will not incur the cost of refactoring your code for flexibility later.

[2] See the Jakarta Avalon framework for a component framework implementing the IOC pattern (http://avalon.apache.org).

7.4 Practicing on an HTTP connection sample

To see how mock objects work in a practical example, let's use a simple application that opens an HTTP connection to a remote server and reads the content of a page. In chapter 6 we tested that application using stubs. Let's now unit-test it using a mock-object approach to simulate the HTTP connection.

In addition, you'll learn how to write mocks for classes that do not have a Java interface (namely, the `HttpURLConnection` class). We will show a full scenario in which you start with an initial testing implementation, improve the implementation as you go, and modify the original code to make it more flexible. We will also show how to test for error conditions using mocks.

As you dive in, you will keep improving both the test code and the sample application, exactly as you might if you were writing the unit tests for the same application. In the process, you will learn how to reach a simple and elegant testing solution while making your application code more flexible and capable of handling change.

Figure 7.2 introduces the sample HTTP application, which consists of a simple `WebClient.getContent` method performing an HTTP connection to a web resource executing on a web server. You want to be able to unit-test the `getContent` method in isolation from the web resource.

7.4.1 Defining the mock object

Figure 7.3 illustrates the definition of a mock object. The `MockURL` class stands in for the real `URL` class, and all calls to the `URL` class in `getContent` are directed to the `MockURL` class. As you can see, the test is the controller: It creates and configures the behavior the mock must have for this test, it (somehow) replaces the real `URL` class with the `MockURL` class, and it runs the test.

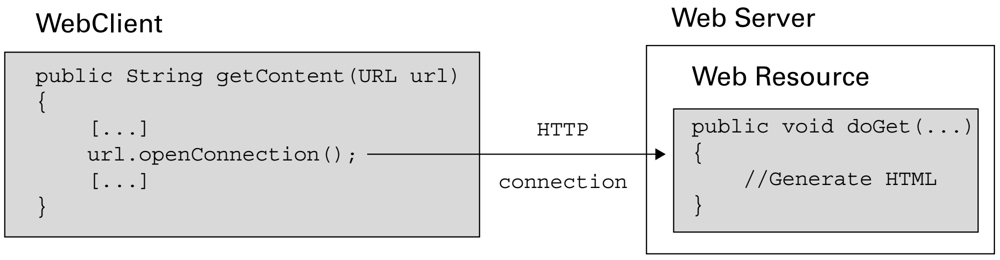
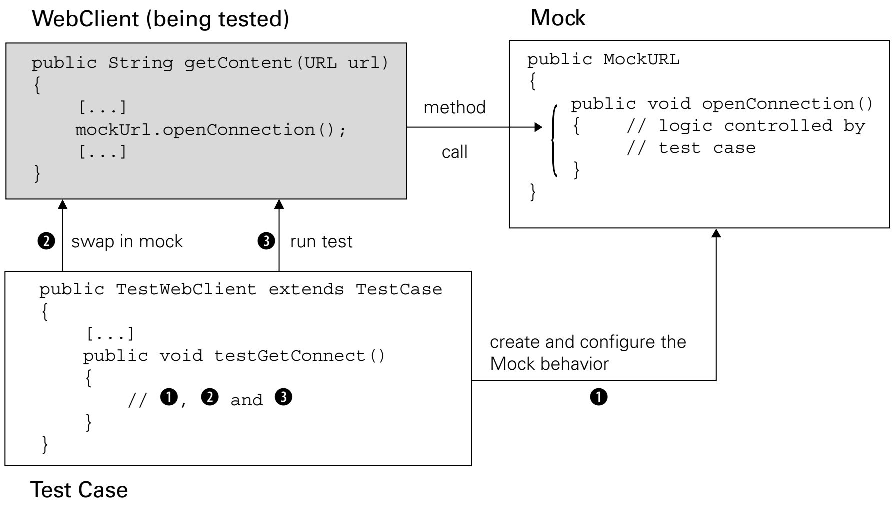


Figure 7.2 The sample HTTP application before introducing the test



```
public String getContent(URL url)
{
    [...]
    mockUrl.openConnection();
    [...]
}
```

```
public MockURL
{
    public void openConnection()
    {    // logic controlled by
         // test case
    }
}
```

```
public TestWebClient extends TestCase
{
    [...]
    public void testGetConnect()
    {
        // ❶, ❷ and ❸
    }
}
```

Figure 7.3 The steps involved in a test using mock objects

Figure 7.3 shows an interesting aspect of the mock-objects strategy: the need to be able to swap-in the mock in the production code. The perceptive reader will have noticed that because the `URL` class is final, it is actually not possible to create a `MockURL` class that extends it. In the coming sections, we will demonstrate how this feat can be performed in a different way (by mocking at another level). In any case, when using the mock-objects strategy, swapping-in the mock instead of the real class is the hard part. This may be viewed as a negative point for mock objects, because you usually need to modify your code to provide a trapdoor. Ironically, modifying code to encourage flexibility is one of the strongest advantages of using mocks, as explained in section 7.3.1.

7.4.2 *Testing a sample method*

The example in listing 7.6 demonstrates a code snippet that opens an HTTP connection to a given URL and reads the content found at that URL. Let's imagine that it's one method of a bigger application that you want to unit-test, and let's unit-test that method.

Listing 7.6 Sample method that opens an HTTP connection

```
package junitbook.fine.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 =                    ❶
                (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);

            InputStream is = connection.getInputStream();  ❷

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

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

❶ ❷ Open an HTTP connection using the `HttpURLConnection` class.

❸ Read the content until there is nothing more to read.

If an error occurs, you return null. Admittedly, this is not the best possible error-handling solution, but it is good enough for the moment. (And your tests will give you the courage to refactor later.)

7.4.3 *Try #1: easy method refactoring technique*

The idea is to be able to test the `getContent` method independently of a real HTTP connection to a web server. If you map the knowledge you acquired in section 7.2, it means writing a mock URL in which the `url.openConnection` method returns a mock `HttpURLConnection`. The `MockHttpURLConnection` class would provide an implementation that lets the test decides what the `getInputStream` method returns. Ideally, you would be able to write the following test:

```
public void testGetContentOk() throws Exception
{
    MockHttpURLConnection mockConnection =
        new MockHttpURLConnection();
    mockConnection.setupGetInputStream(
        new ByteArrayInputStream("It works".getBytes()));       ❶

    MockURL mockURL = new MockURL();
    mockURL.setupOpenConnection(mockConnection);                 ❷

    WebClient client = new WebClient();

    String result = client.getContent(mockURL);      ❸

    assertEquals("It works", result);      ❹
}
```

❶ Create a mock `HttpURLConnection` that you set up to return a stream containing *`It works`* when the `getInputStream` method is called on it.

❷ Do the same for creating a mock `URL` class and set it up to return the `MockURLConnection` when `url.openConnection` is called.

❸ Call the `getContent` method to test, passing to it your `MockURL` instance.

❹ Assert the result.

Unfortunately, this approach does not work! The JDK `URL` class is a final class, and no URL interface is available. So much for extensibility….

You need to find another solution and, potentially, another object to mock. One solution is to stub the `URLStreamHandlerFactory` class. We explored this solution in chapter 6, so let's find a technique that uses mock objects: refactoring the `getContent` method. If you think about it, this method does two things: It gets an `HttpURLConnection` object and then reads the content from it. Refactoring leads to the class shown in listing 7.7 (changes from listing 7.6 are in bold). We have extracted the part that retrieved the `HttpURLConnection` object.

Listing 7.7 Extracting retrieval of the connection object from getContent

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

        try
        {
            HttpURLConnection connection =
                createHttpURLConnection(url);                  ❶
            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();
    }

    protected HttpURLConnection createHttpURLConnection(URL url)     ❶
        throws IOException
    {
        return (HttpURLConnection) url.openConnection();
    }
}
```

❶ Refactoring. You now call the `createHttpURLConnection` method to create the HTTP connection.

How does this solution let you test `getContent` more effectively? You can now apply a useful trick, which consists of writing a test helper class that extends the `WebClient` class and overrides its `createHttpURLConnection` method, as follows:

```
private class TestableWebClient extends WebClient
{
    private HttpURLConnection connection;

    public void setHttpURLConnection(HttpURLConnection connection)
    {
        this.connection = connection;
    }

    public HttpURLConnection createHttpURLConnection(URL url)
        throws IOException
    {
        return this.connection;
    }
}
```

In the test, you can call the `setHttpURLConnection` method, passing it the mock `HttpURLConnection` object. The test now becomes the following (differences are shown in bold):

```
public void testGetContentOk() throws Exception
{
    MockHttpURLConnection mockConnection =
        new MockHttpURLConnection();
    mockConnection.setupGetInputStream(
        new ByteArrayInputStream("It works".getBytes()));

    TestableWebClient client = new TestableWebClient();          ❶
    client.setHttpURLConnection(mockConnection);

    String result = client.getContent(new URL("http://localhost")); ❷

    assertEquals("It works", result);
}
```

❶ Configure `TestableWebClient` so that the `createHttpURLConnection` method returns a mock object.

❷ The `getContent` method accepts a URL as parameter, so you need to pass one. The value is not important, because it will not be used; it will be bypassed by the `MockHttpURLConnection` object.

This is a common refactoring approach called *Method Factory* refactoring, which is especially useful when the class to mock has no interface. The strategy is to extend that class, add some setter methods to control it, and override some of its getter methods to return what you want for the test. In the case at hand, this approach is OK, but it isn't perfect. It's a bit like the Heisenberg Uncertainty Principle: The act of subclassing the class under test changes its behavior, so when you test the subclass, what are you truly testing?

This technique is useful as a means of opening up an object to be more testable, but stopping here means testing something that is similar to (but not exactly) the class you want to test. It isn't as if you're writing tests for a third-party library and can't change the code—you have complete control over the code to test. You can enhance it, and make it more test-friendly in the process.

7.4.4 Try #2: refactoring by using a class factory

Let's apply the Inversion of Control (IOC) pattern, which says that any resource you use needs to be passed to the `getContent` method or `WebClient` class. The only resource you use is the `HttpURLConnection` object. You could change the `WebClient.getContent` signature to

```
public String getContent(URL url, HttpURLConnection connection)
```

This means you are pushing the creation of the `HttpURLConnection` object to the caller of `WebClient`. However, the URL is retrieved from the `HttpURLConnection` class, and the signature does not look very nice. Fortunately, there is a better solution that involves creating a `ConnectionFactory` interface, as shown in listings 7.8 and 7.9. The role of classes implementing the `ConnectionFactory` interface is to return an `InputStream` from a connection, whatever the connection might be (HTTP, TCP/IP, and so on). This refactoring technique is sometimes called a Class Factory refactoring.[3]

Listing 7.8 ConnectionFactory.java

```
package junitbook.fine.try2;

import java.io.InputStream;

public interface ConnectionFactory
{
    InputStream getData() throws Exception;
}
```

The `WebClient` code then becomes as shown in listing 7.9. (Changes from the initial implementation in listing 7.6 are shown in bold.)

Listing 7.9 Refactored WebClient using ConnectionFactory

```
package junitbook.fine.try2;

import java.io.InputStream;

public class WebClient
{
    public String getContent(ConnectionFactory connectionFactory)
    {
        StringBuffer content = new StringBuffer();

        try
        {
            InputStream is = connectionFactory.getData();

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

[3] J. B. Rainsberger calls it Replace Subclasses with Collaborators: http://www.diasparsoftware.com/articles/refactorings/replaceSubclassWithCollaborator.html.

```
        {
            return null;
        }

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

This solution is better because you have made the retrieval of the data content independent of the way you get the connection. The first implementation worked only with URLs using the HTTP protocol. The new implementation can work with any standard protocol (`file://`, `http://`, `ftp://`, `jar://`, and so forth), or even your own custom protocol. For example, listing 7.10 shows the `ConnectionFactory` implementation for the HTTP protocol.

Listing 7.10 HttpURLConnectionFactory.java

```
package junitbook.fine.try2;

import java.io.InputStream;

import java.net.HttpURLConnection;
import java.net.URL;

public class HttpURLConnectionFactory.java implements ConnectionFactory
{
    private URL url;

    public HttpURLConnectionFactory(URL url)
    {
        this.url = url;
    }

    public InputStream getData() throws Exception
    {
        HttpURLConnection connection =
            (HttpURLConnection) this.url.openConnection();
        return connection.getInputStream();
    }
}
```

Now you can easily test the `getContent` method by writing a mock for `Connection-Factory` (see listing 7.11).

Listing 7.11 MockConnectionFactory.java

```
package junitbook.fine.try2;

import java.io.InputStream;

public class MockConnectionFactory implements ConnectionFactory
{
    private InputStream inputStream;

    public void setData(InputStream stream)
    {
        this.inputStream = stream;
    }

    public InputStream getData() throws Exception
    {
        return this.inputStream;
    }
}
```

As usual, the mock does not contain any logic and is completely controllable from the outside (by calling the `setData` method). You can now easily rewrite the test to use `MockConnectionFactory` as demonstrated in listing 7.12.

Listing 7.12 Refactored WebClient test using MockConnectionFactory

```
package junitbook.fine.try2;

import java.io.ByteArrayInputStream;

import junit.framework.TestCase;

public class TestWebClient extends TestCase
{
    public void testGetContentOk() throws Exception
    {
        MockConnectionFactory mockConnectionFactory =
            new MockConnectionFactory();

        mockConnectionFactory.setData(
            new ByteArrayInputStream("It works".getBytes()));

        WebClient client = new WebClient();

        String result = client.getContent(mockConnectionFactory);

        assertEquals("It works", result);
    }
}
```

You have achieved your initial goal: to unit-test the code logic of the `WebClient.getContent` method. In the process you had to refactor it for the test, which led to a more extensible implementation that is better able to cope with change.

7.5 Using mocks as Trojan horses

Mock objects are Trojan horses, but they are not malicious. Mocks replace real objects from the inside, without the calling classes being aware of it. Mocks have access to internal information about the class, making them quite powerful. In the examples so far, you have only used them to emulate real behaviors, but you haven't mined all the information they can provide.

It is possible to use mocks as probes by letting them monitor the method calls the object under test makes. Let's take the HTTP connection example. One of the interesting calls you could monitor is the `close` method on the `InputStream`. You have not been using a mock object for `InputStream` so far, but you can easily create one and provide a `verify` method to ensure that `close` has been called.

Then, you can call the `verify` method at the end of the test to verify that all methods that should have been called, were called (see listing 7.13). You may also want to verify that `close` has been called exactly once, and raise an exception if it was called more than once or not at all. These kinds of verifications are often called *expectations.*

DEFINITION *expectation*—When we're talking about mock objects, an *expectation* is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. For example, a database connection mock could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

Listing 7.13 Mock InputStream with an expectation on close

```
package junitbook.fine.expectation;

import java.io.IOException;
import java.io.InputStream;

import junit.framework.AssertionFailedError;

public class MockInputStream extends InputStream
{
    private String buffer;                      ❶
    private int position = 0;
    private int closeCount = 0;          ❷
    public void setBuffer(String buffer)                ❶
    {
        this.buffer = buffer;
    }

    public int read() throws IOException
    {
        if (position == this.buffer.length())
        {
            return -1;
        }

        return this.buffer.charAt(this.position++);
    }

    public void close() throws IOException
    {
        closeCount++;        ❷
        super.close();
    }

    public void verify() throws AssertionFailedError                   ❸
    {
        if (closeCount != 1)
        {
            throw new AssertionFailedError("close() should "
                + "have been called once and once only");
        }
    }
}
```

❶ Tell the mock what the `read` method should return.

❷ Count the number of times `close` is called.

❸ Verify that the expectations are met.

In the case of the `MockInputStream` class, the expectation for `close` is simple: You always want it to be called once. However, most of the time, the expectation for `closeCount` depends on the code under test. A mock usually has a method like `setExpectedCloseCalls` so that the test can tell the mock what to expect.

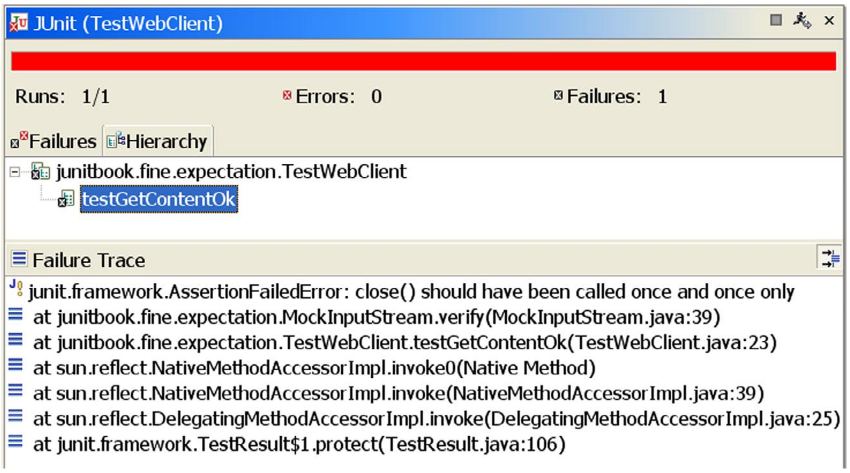
Let's modify the `TestWebClient.testGetContentOk` test method to use the new `MockInputStream`:

```
package junitbook.fine.expectation;

import junit.framework.TestCase;
import junitbook.fine.try2.MockConnectionFactory;

public class TestWebClient extends TestCase
{
    public void testGetContentOk() throws Exception
```

```
        {
            MockConnectionFactory mockConnectionFactory =
                new MockConnectionFactory();
            MockInputStream mockStream = new MockInputStream();
            mockStream.setBuffer("It works");

            mockConnectionFactory.setData(mockStream);

            WebClient client = new WebClient();

            String result = client.getContent(mockConnectionFactory);

            assertEquals("It works", result);
            mockStream.verify();
        }
    }
```

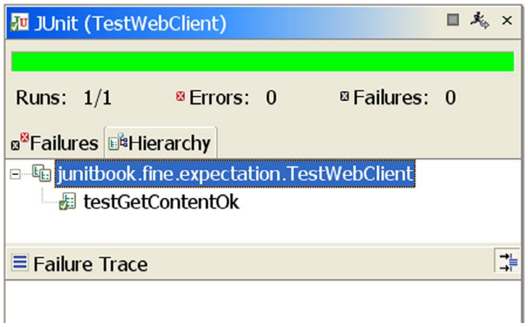
Instead of using a real `ByteArrayInputStream` as in previous tests, you now use the `MockInputStream`. Note that you call the `verify` method of `MockInputStream` at the end of the test to ensure that all expectations are met. The result of running the test is shown in figure 7.4.

The test fails with the message *close() should have been called once and once only*. Why? Because you have not closed the input stream in the `WebClient.getContent` method. The same error would be raised if you were closing it twice or more, because the test verifies that it is called once and only once. Let's correct the code under test (see listing 7.14). You now get a nice green bar (figure 7.5).



```
junit.framework.AssertionFailedError: close() should have been called once and once only
at junitbook.fine.expectation.MockInputStream.verify(MockInputStream.java:39)
at junitbook.fine.expectation.TestWebClient.testGetContentOk(TestWebClient.java:23)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at junit.framework.TestResult$1.protect(TestResult.java:106)
```

Figure 7.4 Running `TestWebClient` with the new `close` expectation


JUnit (TestWebClient)

Runs: 1/1 Errors: 0 Failures: 0

Failures Hierarchy

junitbook.fine.expectation.TestWebClient
testGetContentOk

Failure Trace

Figure 7.5 Working `WebClient` that closes the input stream

Listing 7.14 WebClient closing the stream

```
public class WebClient
{
    public String getContent(ConnectionFactory connectionFactory)
        throws IOException
    {
        String result;

        StringBuffer content = new StringBuffer();
        InputStream is = null;
        try
        {
            is = connectionFactory.getData();

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

            result = content.toString();
        }
        catch (Exception e)
        {
            result = null;
        }

        // Close the stream
        if (is != null)                                  ❶
        {
            try
            {
                is.close();
            }
            catch (IOException e)
            {
                result = null;                    ❶
            }
        }

        return result;
    }
}
```

❶ Close the stream and return null if an error occurs when you're closing it.

There are other handy uses for expectations. For example, if you have a component manager calling different methods of your component life cycle, you might expect them to be called in a given order. Or, you might expect a given value to be passed as a parameter to the mock. The general idea is that, aside from behaving the way you want during a test, your mock can also provide useful feedback on its usage.

NOTE The MockObjects project (http://www.mockobjects.com) provides some ready-made mock objects for standard JDK APIs. These mocks usually have expectations built in. In addition, the MockObjects project contains some reusable expectation classes that you can use in your own mocks.

7.6 Deciding when to use mock objects

You have seen how to create mocks, but we haven't talked about when to use them. For example, in your tests, you have sometimes used the real objects (when you used `ByteArrayInputStream` in listing 7.11, for example) and sometimes mocked them.

Here are some cases in which mocks provide useful advantages over the real objects. (This list can be found on the C2 Wiki at http://c2.com/cgi/wiki?Mock-Object.) This should help you decide when to use a mock:

- Real object has non-deterministic behavior
- Real object is difficult to set up
- Real object has behavior that is hard to cause (such as a network error)
- Real object is slow
- Real object has (or is) a UI

- Test needs to query the object, but the queries are not available in the real object (for example, "was this callback called?")
- Real object does not yet exist

7.7 Summary

This chapter has described a technique called mock objects that lets you unit-test code in isolation from other domain objects and from the environment. When it comes to writing fine-grained unit tests, one of the main obstacles is to abstract yourself from the executing environment. We have often heard the following remark: "I haven't tested this method because it's too difficult to simulate a real environment." Well, not any longer!

In most cases, writing mock-object tests has a nice side effect: It forces you to rewrite some of the code under test. In practice, code is often not written well. You hard-code unnecessary couplings between the classes and the environment. It's easy to write code that is hard to reuse in a different context, and a little nudge can have a big effect on other classes in the system (similar to the domino effect). With mock objects, you must think differently about the code and apply better design patterns, like Interfaces and Inversion of Control (IOC).

Mock objects should be viewed not only as a unit-testing technique but also as a design technique. A new rising star among methodologies called Test-Driven Development advocates writing tests before writing code. With TDD, you don't have to refactor your code to enable unit testing: The code is already under test! (For a full treatment of the TDD approach, see Kent Beck's book *Test Driven Development.*[4] For a brief introduction, see chapter 4.)

Although writing mock objects is easy, it can become tiresome when you need to mock hundreds of objects. In the following chapters we will present several open source frameworks that automatically generate ready-to-use mocks for your classes, making it a pleasure to use the mock-objects strategy.

[4] Kent Beck, *Test Driven Development: By Example* (Boston: Addison-Wesley, 2003).

8 In-container testing with Cactus

This chapter covers

- Drawbacks of mock objects when unit-testing components
- Introducing testing inside the container with Cactus
- How Cactus works

Good design at good times. Make it run, make it run right.
—Kent Beck, *Test Driven Development: By Example*

Starting with this chapter, we'll study how to unit-test J2EE components. Unit-testing components is more difficult than just testing plain Java classes. The components interact with their container, and the container services are available only when the container is running. JUnit is not designed to run inside a container like a J2EE component. So, how can we unit-test components?

This chapter examines one approach to unit-testing J2EE components: *in-container unit testing*, or *integration unit testing*. (These concepts and terminologies were introduced by the Cactus framework.) More specifically, we'll discuss the pros and cons of running J2EE tests inside the container using the Cactus framework (http://jakarta.apache.org/cactus/). We'll show what can be achieved using the mock-objects approach introduced in chapter 7, where it comes up short, and how the in-container testing approach enables you to write integration unit tests.

8.1 The problem with unit-testing components

Imagine you have a web application that uses servlets, and that you wish to unit-test the `isAuthenticated` method in listing 8.1 from a `SampleServlet` servlet.

Listing 8.1 Sample of a servlet method to unit-test

```
package junitbook.container;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class SampleServlet extends HttpServlet
{
    public boolean isAuthenticated(HttpServletRequest request)
    {
        HttpSession session = request.getSession(false);

        if (session == null)
        {
            return false;
        }

        String authenticationAttribute =
            (String) session.getAttribute("authenticated");

        return Boolean.valueOf(
            authenticationAttribute).booleanValue();
    }
}
```

In order to be able to test this method, you need to get hold of a valid `HttpServletRequest` object. Unfortunately, it is not possible to call `new HttpServletRequest` to create a usable request. The life cycle of `HttpServletRequest` is managed by the container. JUnit alone is not enough to write a test for the `isAuthenticated` method.

DEFINITION *component/container*—A component is a piece of code that executes inside a container. A container is a receptacle that offers useful services for the components it is hosting, such as life cycle, security, transaction, distribution, and so forth.

Fortunately, several solutions are available; we'll cover them in the following sections. Overall, we're presenting two core solutions: *outside-the-container testing* with mock objects and *in-container testing* using Cactus.

A variation on these two approaches would be to use a stubbed container, such as the ServletUnit module in HttpUnit (http://httpunit.sourceforge.net/). Unfortunately, we haven't found a full-blown implementation of a stub J2EE container. ServletUnit implements only a portion of the Servlet specification. (For more about stubs, see chapter 6.)

Mock objects and Cactus are both valid approaches. In chapter 7, we discussed the advantages and drawbacks of the mock objects outside-the-container strategy. In this chapter, we focus on the Cactus in-container strategy to show its relative advantages and drawbacks. Chapter 9 turns the focus to unit-testing servlets. In that chapter, we discuss when to use mock objects and when to use Cactus tests when you're testing servlets.

8.2 Testing components using mock objects

Let's try using mock objects to test a servlet and then discuss the benefits and drawbacks of this approach. In chapter 7, you built your own mock objects. Fortunately, several frameworks are available that automate the generation of mock objects. In this chapter you'll use *EasyMock* (http://www.easymock.org/—version 1.0 or above).

EasyMock is one of the best-known mock-object generators today. We'll demonstrate how to use some other generators in later chapters. EasyMock is implemented using Java *dynamic proxies* so that it can transparently generate mock objects at runtime, as shown in the next section.

8.2.1 Testing the servlet sample using EasyMock

The solution that comes to mind to unit-test the `isAuthenticated` method (listing 8.1) is to mock the `HttpServletRequest` class using the mock objects approach described in chapter 7. This approach would work, but you would need to write quite a few lines of code. The result can be easily achieved using the EasyMock framework as demonstrated in listing 8.2.

Listing 8.2 Using EasyMock to unit-test servlet code

```
package junitbook.container;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import org.easymock.MockControl;

import junit.framework.TestCase;

public class TestSampleServlet extends TestCase
{
    private SampleServlet servlet;

    private MockControl controlHttpServlet;                      ❶
    private HttpServletRequest mockHttpServletRequest;

    private MockControl controlHttpSession;
    private HttpSession mockHttpSession;

    protected void setUp()
    {
        servlet = new SampleServlet();

        controlHttpServlet = MockControl.createControl(                ❶
            HttpServletRequest.class);
        mockHttpServletRequest =
            (HttpServletRequest) controlHttpServlet.getMock();

        controlHttpSession = MockControl.createControl(
            HttpSession.class);
        mockHttpSession =
            (HttpSession) controlHttpSession.getMock();
    }

    protected void tearDown()                               ❷
    {
        controlHttpServlet.verify();
        controlHttpSession.verify();
    }

    public void testIsAuthenticatedAuthenticated()
    {
        mockHttpServletRequest.getSession(false);                    ❸
        controlHttpServlet.setReturnValue(mockHttpSession);

        mockHttpSession.getAttribute("authenticated");               ❹
        controlHttpSession.setReturnValue("true");

        controlHttpServlet.replay();                                 ❺
        controlHttpSession.replay();

        assertTrue(servlet.isAuthenticated(mockHttpServletRequest));
    }

    public void testIsAuthenticatedNotAuthenticated()
    {
        mockHttpServletRequest.getSession(false);                    ❸
        controlHttpServlet.setReturnValue(mockHttpSession);

        mockHttpSession.getAttribute("authenticated");               ❻
        controlHttpSession.setReturnValue(null);

        controlHttpServlet.replay();                                 ❺
        controlHttpSession.replay();

        assertFalse(
            servlet.isAuthenticated(mockHttpServletRequest));
    }

    public void testIsAuthenticatedNoSession()
    {
        mockHttpServletRequest.getSession(false);                    ❼
        controlHttpServlet.setReturnValue(null);
        controlHttpServlet.replay();                                 ❺
        controlHttpSession.replay();

        assertFalse(
            servlet.isAuthenticated(mockHttpServletRequest));
    }
}
```

❶ Create two mocks, one of `HttpServletRequest` and one of `HttpSession`. EasyMock works by creating a dynamic proxy for the mock, which is controlled through a control object (`MockControl`).

❷ Tell EasyMock to verify that the calls to the mocked methods have effectively happened. EasyMock reports a failure if you have defined a mocked method that is not called in the test. It also reports a failure if a method for which you have not defined any behavior is called. You do this in JUnit's `tearDown` method so that the verification is done automatically for all tests. For this factorization to work for all the tests, you have to activate the `HttpSession` mock in `testIsAuthenticatedNoSession`, even though you don't use it in that test. With EasyMock, a mock can only be verified if it has been activated.

❸ Using the control object, tell the `HttpServletRequest` mock that when `getSession(false)` is called, the method should return the `HttpSession` mock.

❹ ❻ ❼ Do the same thing as in the previous step, telling the mock how to behave when such-and-such method is called.

❸ ❹ ❻ ❼ Notice that in order to tell the mocks how to behave, you call their methods and then tell the control object what the method should return. This is the EasyMock *training mode.*

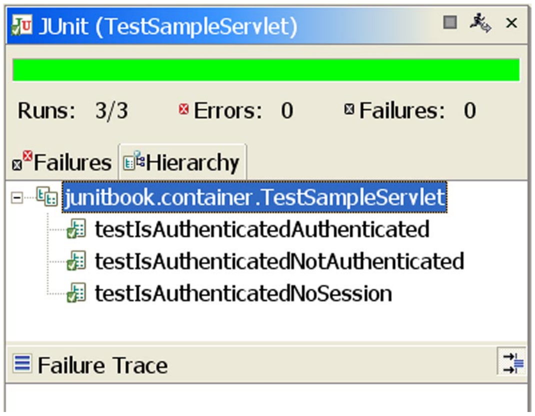
❺ Once you activate the mocks here, you step out of the training mode, and calls to the mock methods return the expected results.

The result of executing this `TestSampleServlet TestCase` in Eclipse is shown in figure 8.1. (Of course, other IDEs would yield a similar result.)

8.2.2 Pros and cons of using mock objects to test components

The biggest advantage of the mock-object approach is that it does not require a running container in order to execute the tests. The tests can be set up quickly, and they run fast. The main drawback is that the components are not tested in the container in which they will be deployed. You don't get to test the interactions with the container, which is an important part of the code. You're also not testing the interactions between the different components when they are run inside the container.

You still need a way to perform integration testing. Writing and running functional tests could achieve this. The problem with functional tests is that they are coarse-grained and test only a full use case—you lose the benefits of fine-grained



```
Runs:  3/3     Errors:  0     Failures:  0
junitbook.container.TestSampleServlet
    testIsAuthenticatedAuthenticated
    testIsAuthenticatedNotAuthenticated
    testIsAuthenticatedNoSession
```

Figure 8.1
Result of executing the mock object `TestCase` for testing `isAuthenticated`

unit testing. You also know that you won't be able to test as many different cases with functional tests as you can with unit tests.

There are also other possible disadvantages of using the mock-objects approach. For example, you may have a lot of mock objects to set up; this may prove to be a non-negligible overhead. Obviously, the cleaner the code is (especially with small and focused methods), the easier tests are to set up.

Another important drawback with the mock-objects approach is that in order to set up the test, you usually must know exactly how the API being mocked behaves. It's easy to know that behavior for your own API, but not so easy for an external API, like the Servlet API.

Even though they all implement the same API, all containers do not behave in the same way. For example, take the following piece of code:

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response)
{
    response.setContentType("text/xml");
}
```

Seems simple enough, but, if you run it in Tomcat 4.x (http://jakarta.apache.org/tomcat/) and in Orion 1.6.0 (http://www.orionserver.com/), you'll notice different behaviors. In Tomcat, the returned content type is `text/xml`, but in Orion it's `text/html`.

This may seem like an extreme example, and it is. All servlet containers manage to implement the API in pretty much the same way. But the situation is far worse for the J2EE APIs—especially the EJB APIs. A specification is a document, and sometimes it's possible to interpret a document in different ways. A specification can even be inconsistent, making it impossible to implement it to the letter. (Try talking to servlet container implementers about the Servlet Filters API!) And, of course, containers have bugs. (The example we have shown may be a bug in Orion 1.6.0.) It's an unfortunate fact of life that you have to live with bugs when you're writing an application.

To summarize, the drawbacks of using mock objects for unit-testing J2EE components are that they:

- Do not test interactions with the container or between the components
- Do not test the deployment part of the components
- Need excellent knowledge of the API being called in order to mock it, which can be difficult (especially for external libraries)
- Do not provide confidence that the code will run in the target container

Whereas this section has dwelled on the mock-objects approach, the next section introduces the other approach: integration unit testing.

8.3 What are integration unit tests?

Running unit tests inside the container gives you the best of both worlds: all the benefits of unit tests *and* confirmation that your code will run correctly inside its target container. *Integration unit tests* stand between *logic unit tests* (testing individual methods) and *functional tests* (testing interactions between methods). Figure 8.2 depicts the relationship between integration, logic, and functional unit tests.

Viewed from another angle, integration unit-testing is a generalization of logic unit-testing and functional testing. If you set the test-start slider to be in a given method and the test-end slider to be the same method, you get a logic unit test. If you set the start and end sliders to be one of the application entry points, you get a functional test.

The main idea is that nothing is completely white or black. Sometimes you need more flexibility in how you manage your tests, and you would like to perform testing between logic unit tests and functional tests. With integration unit tests, you can decide where the test starts and where it ends. It can begin in a given method, span several method calls, and stop in another method further down the chain. It can begin in a given method and go all the way to the end (up

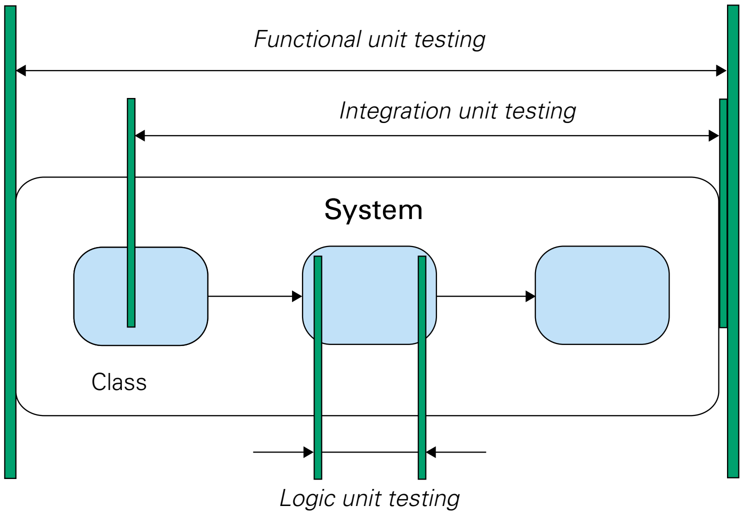


Figure 8.2 Integration unit tests versus logic unit tests versus functional tests

to a database). Or, it can begin at an entry point of the application and stop somewhere in the calling chain (say, by using mock objects or stubs to cut the chain).

In the next section we introduce Jakarta Cactus, a unit-testing framework specializing in integration unit-testing for server-side Java components.

8.4 Introducing Cactus

Cactus is an open source framework (http://jakarta.apache.org/cactus/) for unit-testing server-side Java code (mostly J2EE components in the current version of Cactus). More specifically, it is geared toward integration unit-testing (also called *in-container unit-testing*, a term coined by the Cactus team).

The Cactus vision goes something like this: In the past few years we have seen a move from Java code toward metadata, such as deployment descriptors. Containers offer more services for their components in an increasingly transparent fashion for the component writer. The latest trend is to use Aspect-Oriented Programming (AOP) to decouple programming aspects (security, logging, and so forth) from the business code. The main business code is shrinking, whereas configuration data and metadata are expanding. Integration is becoming more important to verify that an application works. Cactus takes a proactive approach to integration by moving it into the development cycle and providing a solution for fine-grained testing.

Let's see Cactus in action. In later sections, we'll explain in more detail how it works.

8.5 Testing components using Cactus

Let's write unit tests for the `isAuthenticated` method (listing 8.1). The tests in listing 8.3 are exactly the same tests that you wrote using the mock-objects approach in listing 8.2, but this time you use Cactus.

Listing 8.3 Using Cactus to unit-test servlet code

```
package junitbook.container;

import org.apache.cactus.ServletTestCase;
import org.apache.cactus.WebRequest;

public class TestSampleServletIntegration extends ServletTestCase
{
    private SampleServlet servlet;

    protected void setUp()
    {
        servlet = new SampleServlet();
    }

    public void testIsAuthenticatedAuthenticated()
    {
        session.setAttribute("authenticated", "true");      ❶

        assertTrue(servlet.isAuthenticated(request));
    }

    public void testIsAuthenticatedNotAuthenticated()
    {
        assertFalse(servlet.isAuthenticated(request));      ❶
    }

    public void beginIsAuthenticatedNoSession(WebRequest request)
    {
        request.setAutomaticSession(false);
    }

    public void testIsAuthenticatedNoSession()
    {
        assertFalse(servlet.isAuthenticated(request));      ❶
    }

}
```

❶ The Cactus framework exposes the container objects (in this case the `HttpServletRequest` and `HttpSession` objects) to your tests, making it easy and quick to write unit tests.

8.5.1 Running Cactus tests

Let's run the Cactus tests. Doing so is more complex than running a pure JUnit test, because you need to package your code, deploy it into a container, start the container, and then start the tests using a JUnit test runner. Fortunately, Cactus hides much of the complexity and has several runners (a.k.a. *front ends*) that simplify the execution of the tests by automatically performing most, if not all, of these different steps. We'll demonstrate how to run Cactus tests using the most advanced runners: Ant integration, the Maven plugin, and Jetty integration. Figure 8.3 shows the full list (as of this writing) of available Cactus runners. Many of these are written by the Cactus team: Ant integration, browser integration, Eclipse plugin, Jetty integration, and the Maven plugin.

8.5.2 Executing the tests using Cactus/Jetty integration

Cactus/Jetty integration makes it easy to run the Cactus tests from any IDE, in much the same way you ran mock objects with Jetty (but this time, the tests will

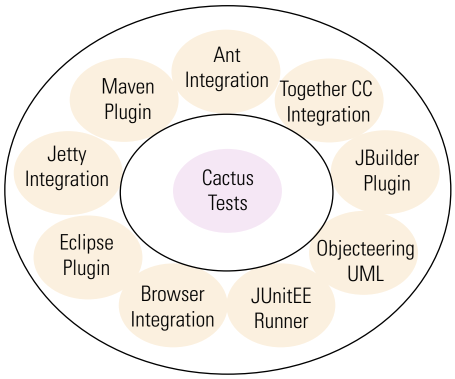


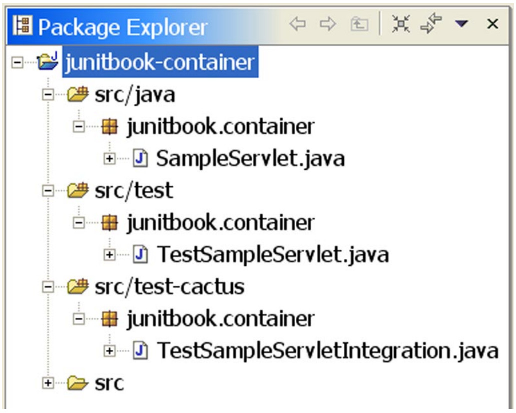
Figure 8.3
The Cactus runners, which are used to easily execute Cactus tests

run inside the container). For these examples, we'll use two of our favorite products, Jetty and Eclipse. (For more about Jetty, see chapter 6. For more about Eclipse, see chapter 5 and appendix B.)

Cactus/Jetty integration yields several benefits: It works with any IDE, it is very fast, and you can set breakpoints for debugging. (Jetty runs in the same JVM as your tests.) As we mentioned in the previous section, Cactus boasts an Eclipse plugin that integrates it even more tightly with Eclipse. However, our goal here is to be tool-agnostic and show you techniques that will work in any IDE.

Figure 8.4 shows what the project looks like in Eclipse. The mock object tests are in `src/test` and the Cactus tests are in `src/test-cactus`.

In order to run a Cactus test with Jetty, you need to create a JUnit test suite containing the tests and wrapped by the special `JettyTestSetup` class provided by Cactus (found in the Cactus jar). `JettyTestSetup` extends the JUnit `TestSetup`



```
junitbook-container
  src/java
    junitbook.container
      SampleServlet.java
  src/test
    junitbook.container
      TestSampleServlet.java
  src/test-cactus
    junitbook.container
      TestSampleServletIntegration.java
  src
```

Figure 8.4
The project structure in Eclipse