JUnit IN ACTION

Vincent Massol
with Ted Husted

MANNING

JUnit in Action

by Vincent Massol
with Ted Husted

Chapter 7

Copyright 2003 Manning Publications

contents

PART I JUNIT DISTILLED

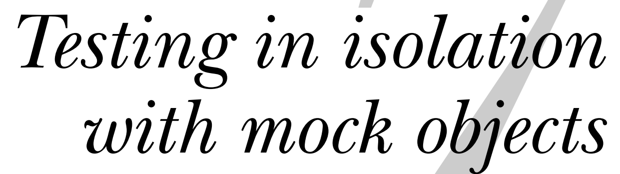
Chapter 1 ■ JUnit jumpstart
Chapter 2 ■ Exploring JUnit
Chapter 3 ■ Sampling JUnit
Chapter 4 ■ Examining software tests
Chapter 5 ■ Automating JUnit

PART II TESTING STRATEGIES

Chapter 6 ■ Coarse-grained testing with stubs
Chapter 7 ■ Testing in isolation with mock objects
Chapter 8 ■ In-container testing with Cactus

PART III TESTING COMPONENTS

Chapter 9 ■ Unit testing servlets and filters
Chapter 10 ■ Unit testing JSPs and taglibs
Chapter 11 ■ Unit testing database applications
Chapter 12 ■ Unit testing EJBs



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.

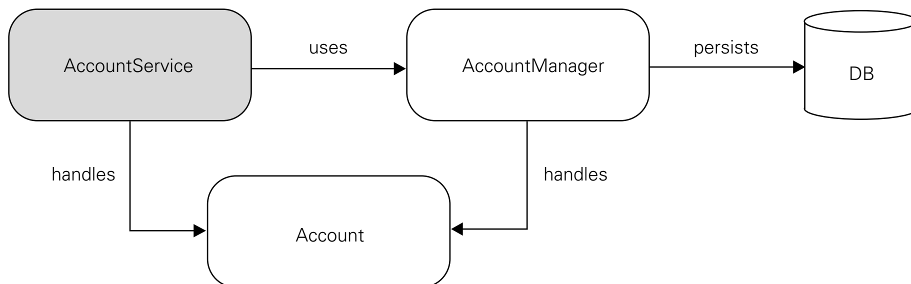


Figure 7.1 In this simple bank account example, you will use a mock object to test an account transfer method.

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.

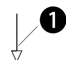
Listing 7.3 AccountService.java

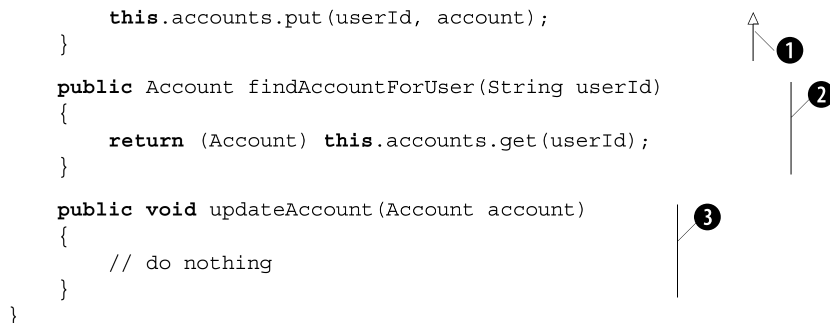
```
package junitbook.fine.tasting;

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)
    {
```





```
        this.accounts.put(userId, account);
    }

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

    public void updateAccount(Account account)
    {
        // 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 =                    (1)
            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);          (2)

        assertEquals(150, senderAccount.getBalance());              (3)
        assertEquals(150, beneficiaryAccount.getBalance());
    }
}
```

As usual, a test has three steps: the test setup (1), the test execution (2), and the verification of the result (3). 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);                  (1)

    public Account findAccountForUser(String userId)
    {
        LOGGER.debug("Getting account for user [" + userId + "]");
        ResourceBundle bundle =
           PropertyResourceBundle.getBundle("technical");              (2)
        String sql = bundle.getString("FIND_ACCOUNT_FOR_USER");

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

1. You get a `Log` object using a `LogFactory` that creates it.
2. 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
        [...]
    }
[...]
}
```

❶ 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();                                    (1)
    MockConfiguration configuration = new MockConfiguration();         (2)
   configuration.setSQL("SELECT * [...]");

    DefaultAccountManager am =                                         (3)
        new DefaultAccountManager(logger, configuration);

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

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

1. Use a mock logger that implements the `Log` interface but does nothing.
2. Create a `MockConfiguration` instance and set it up to return a given SQL query when `Configuration.getSQL` is called.
3. 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.

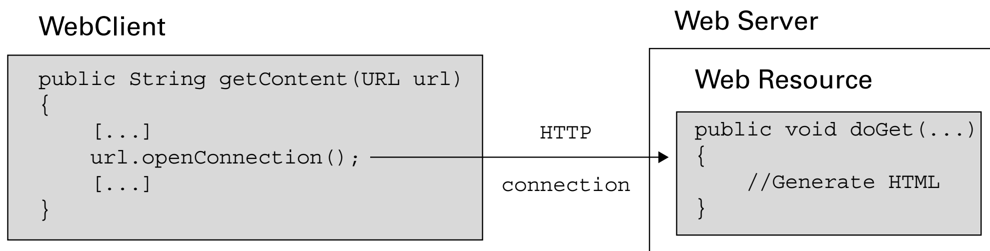


Figure 7.2 The sample HTTP application before introducing the test

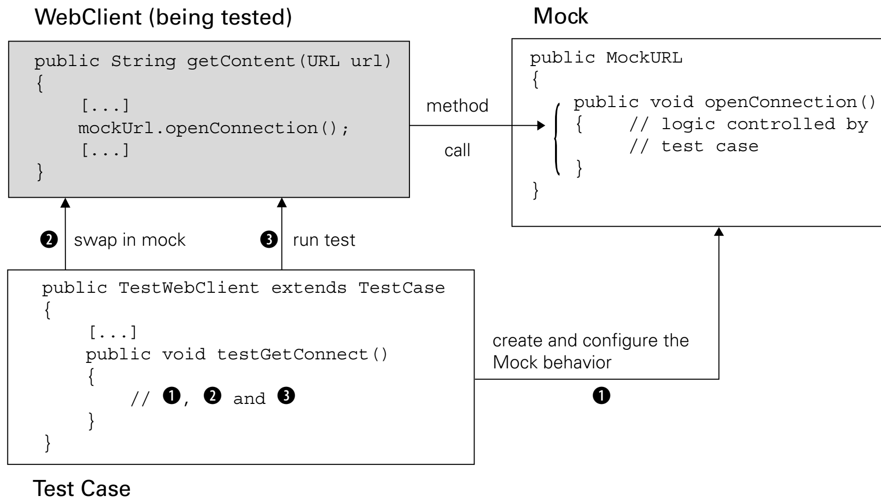


Figure 7.3 The steps involved in a test using mock objects

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)    (1)
        throws IOException
    {
        return (HttpURLConnection) url.openConnection();
    }
}
```

(1) 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();          (1)
    client.setHttpURLConnection(mockConnection);

    String result = client.getContent(new URL("http://localhost")); (2)

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

(1) Configure `TestableWebClient` so that the `createHttpURLConnection` method returns a mock object.

(2) 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)                          (1)
    {
        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++;          (2)
        super.close();
    }

    public void verify() throws AssertionFailedError                  (3)
    {
        if (closeCount != 1)
        {
            throw new AssertionFailedError("close() should "
                + "have been called once and once only");
        }
    }
}
```

(1) Tell the mock what the `read` method should return.

(2) Count the number of times `close` is called.

(3) 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).

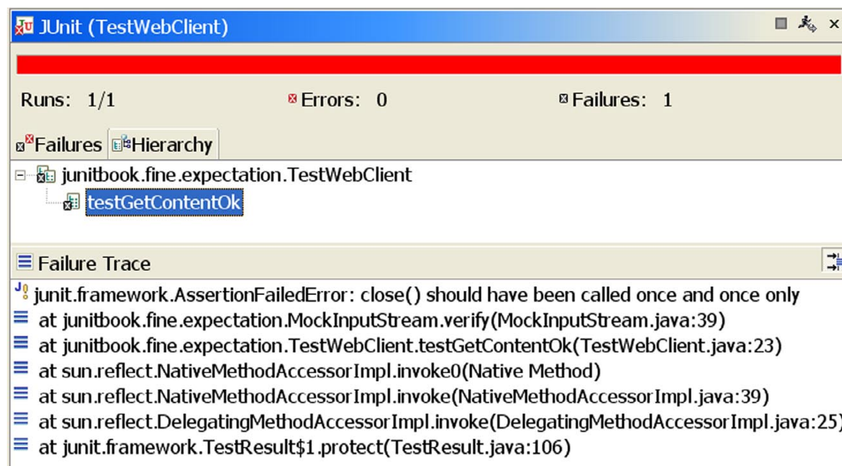


Figure 7.4 Running `TestWebClient` with the new `close` expectation

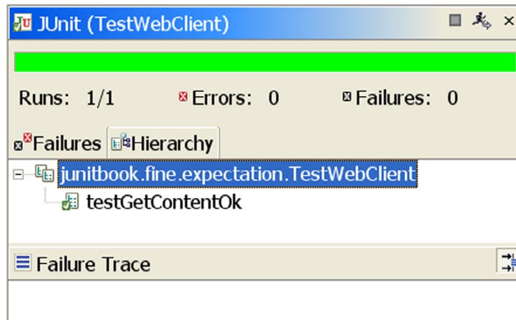
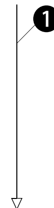
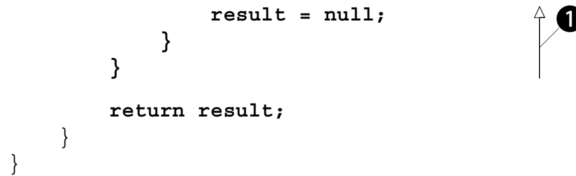


Figure 7.5 Working `WebClient` that closes the input stream

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)
        {
            try
            {
                is.close();
            }
            catch (IOException e)
            {
```





```
                result = null;
            }
        }

        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?MockObject.) 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).

JAVA

Green Bar

JUnit IN ACTION

Vincent Massol with Ted Husted

Developers in the know are switching to a new testing strategy—unit testing—in which coding is interleaved with testing. This powerful approach results in better-designed software with fewer defects and faster delivery cycles. Unit testing is reputed to give developers a kind of "high"—whenever they take a new programming step, their confidence is boosted by the knowledge that every previous step has been confirmed to be correct.

JUnit in Action will get you coding the new way in a hurry. As inevitable errors are continually introduced into your code, you'll want to spot them as quickly as they arise. You can do this using unit tests, and using them often. Rich in real-life examples, this book is a discussion of practical testing techniques by a recognized expert. It shows you how to write automated tests, the advantages of testing a code segment in isolation from the rest of your code, and how to decide when an integration test is needed. It provides a valuable—and unique—discussion of how to test complete J2EE applications.

What's Inside

- Testing in isolation with mock objects
- In-container testing with Cactus
- Automated builds with Ant and Maven
- Testing from within Eclipse
- Unit testing
  - Java apps
  - Servlets
  - JSP
  - Taglibs
  - Filters
  - EJB
  - DB apps

Vincent Massol is the creator of the Jakarta Cactus testing framework and an active member of the Maven and MockObjects development teams. He is CTO of Pivolis, a specialist in agile offshore software development. Vince lives in the City of Light—Paris, France.

The *keep the bar green* frog logo is a trademark of Object Mentor, Inc. in the United States and other countries.

"... captures best practices for effective JUnit and in particular J2EE testing. Don't unit test your J2EE applications without it!"

—Erich Gamma, IBM OTI Labs
Co-author of JUnit

"Outstanding job... It rocks—a joy to read! I recommend it wholeheartedly."

—Erik Hatcher, co-author of
Java Development with Ant

"Brings the mass of information out there under one coherent umbrella."

—J. B. Rainsberger, leader in
the JUnit community, author

"Doesn't shy from tough cases ... Vince really stepped up, rather than side-stepping the real problems people face."

—Scott Stirling, BEA

www.manning.com/massol

Author responds to reader questions

Ebook edition available

ISBN 1-930110-99-5

MANNING $39.95 US/$59.95 Canada