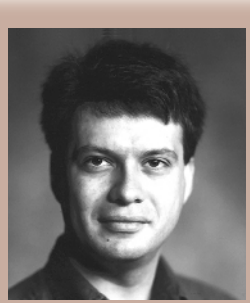
Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

Mock Objects

Dave Thomas and Andy Hunt

Yet sit and see; Minding true things by what their mockeries be. —Shakespeare, Henry V

One thing that makes unit-testing code so hard is the way the real world keeps intruding. If all we had to do was code up tests for methods that sort arrays or generate Fibonacci series, life would be easy. But in the real world we have to test code that uses databases, communications devices, user interfaces, and external applications. We might



have to interface to devices that aren't yet available or simulate network errors that are impossible to generate locally. This all conspires to stop our unit tests from being neat, self-contained (and orthogonal) chunks of code. Instead, if we're not careful, we find ourselves writing tests that end up initializing nearly every system component just to give the tests enough context to run. Not only is this time consuming, it also introduces a ridiculous amount of coupling into the testing process: someone changes an interface or a database table, and suddenly the setup code for your poor little unit test dies mysteriously. Even the best-intentioned developers become discouraged after this happens a few times. Eventually, testing starts to drop off, and we all know where that leads.

Fortunately there's a testing pattern that can help. Using mock objects, you can test code in splendid isolation, simulating all those messy real-world things that would otherwise make automated testing impossible. And, as with many other testing practices, the discipline of using mock objects can improve your code's structure.

An example: Testing a servlet

Servlets are chunks of code that a Web server manages. Requests to certain URLs are forwarded to a servlet container (or manager) such as Jakarta Tomcat (http://jakarta.apache.org/tomcat), which in turn invokes the servlet code. The servlet then builds a response that it sends back to the requesting browser. From the end user's perspective, it's just like accessing any other page.

Figure 1 shows part of the source of a trivial servlet that converts temperatures from Fahrenheit to Celsius. Let's quickly step through its operation. When the servlet container receives the request, it automatically invokes the servlet method `doGet()`, passing in two parameters, a request and a response. (These are important for our testing later). The request parameter contains information about the request; the servlet's job is to fill in the response. The servlet's body gets the contents of the field `"Fahrenheit"` from the request, converts it to Celsius, and writes the results back to the user. The writing is done via a PrintWriter object, which a factory method in the response object provides. If an error occurs converting the number (perhaps the user typed "boo!"

0740-7459/02/$17.00 © 2002 IEEE

into the form's temperature field), we catch the exception and report the error in the response.

Having written this code (or before writing it, for those in the Extreme Programming tribe), we'll want a unit test to verify it. This is where things start looking difficult. This snippet of code runs in a fairly complex environment (a Web server and a servlet container), and it requires a user sitting at a browser to interact with it. This is hardly the basis of a good automated unit test.

But let's look at our servlet code again. Its interface is pretty simple: as we mentioned before, it receives two parameters, a request and a response. The request object must be able to provide a reasonable string when its `getParameter()` method is called, and the response object must support `setContentType()` and `getWriter()`. It's starting to look as if we might be able to write some *stubs*: objects that pretend to be real request and response objects but that contain just enough logic to let us run our code. In principle this is easy: both `HttpServletRequest` and `HttpServletResponse` are interfaces, so all we have to do is whip up a couple of classes that implement the interfaces and we're set. Unfortunately, when we look at the interface, we discover that we'll need to implement dozens of methods just to get the thing to compile. Fortunately, other folks have already done the work for us.

```
 1 public void doGet(HttpServletRequest req,
 2                    HttpServletResponse res)
 3    throws ServletException, IOException
 4 {
 5    String str_f = req.getParameter("Fahrenheit");
 6
 7    res.setContentType("text/html");
 8    PrintWriter out = res.getWriter();
 9
10    try {
11       int    temp_f = Integer.parseInt(str_f);
12       double temp_c = (temp_f - 32) * 5.0 / 9.0;
13       out.println("Fahrenheit: " + temp_f +
                     ", Celsius: " + temp_c);
14    }
15    catch (NumberFormatException e) {
16       out.println("Invalid temperature: " + str_f);
17    }
18 }
```

Figure 1. A trivial servlet that converts temperatures from Fahrenheit to Celsius.

Mock objects

Tim Mackinnon, Steve Freeman, and Philip Craig introduced the concept of mock objects in their paper "Endo-Testing: Unit Testing with Mock Objects" (www.cs.ualberta.ca/~hoover/cmput401/XP-Notes/xp-conf/Papers/4_4_MacKinnon.pdf), which they presented at XP2000. Their idea is a natural extension of the ad hoc stubbing that testers have been doing all along. The difference is that they describe a framework to make writing mock objects and incorporating them into unit testing easier.

Their paper lists seven good reasons to use a mock object (paraphrased slightly here):

- The real object has nondeterministic behavior.
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about *how* it was used (for example, a test might need to check to see that a callback function was actually called).
- The real object does not yet exist.

Mackinnon, Freeman, and Craig also developed the code for a mock object framework for Java programmers (available at www.mockobjects.com). Let's use that code to test our servlet.

The good news is that in addition to the underlying framework code, the `mockobjects` package comes with a number of mocked-up application-level objects. You'll find mock output objects (`OutputStream`, `PrintStream`, and `PrintWriter`), objects that mock the `java.sql` library, and classes for faking out a servlet environment. In particular, the package provides mocked-up versions of `HttpServletRequest` and `HttpServletResponse`, which by an incredible coincidence are the types of the parameters of the method we want to test.

We can use mock objects in two distinct ways. First, we can use them to set up an environment in which our test code runs: we can initialize values in the objects that the method under test uses. Figure 2 shows a typical set of tests using the JUnit testing framework, which is available at www.junit.org. We use a `MockHttpServletRequest` object to set up the context in which to run the test. On line six of the code, we set the parameter `"Fahrenheit"` to the value `"boo!"` in the request object. This is equivalent to the user entering "boo!" in the corresponding form field; our mock object eliminates the need for human input when the test runs.

Mock objects can also verify that actions were taken. On line seven of Figure 2, we tell the response object that we expect the method under test to set the response's content type to `text/html`. Then, on lines 9 and 22, after the method under test has run, we tell the response object to verify that this happened. Here, the mock object eliminates the need for a human to check the result visually. This example shows a pretty trivial verification: in reality, mock objects can verify that fairly complex sequences of actions have been performed.

Mock objects can also record the data that was given to them. In our case, the response object receives the

```
 1 public void test_bad_parameter() throws Exception {
 2    TemperatureServlet s = new TemperatureServlet();
 3    MockHttpServletRequest request =
        new MockHttpServletRequest();
 4    MockHttpServletResponse response =
        new MockHttpServletResponse();
 5
 6    request.setupAddParameter("Fahrenheit", "boo!");
 7    response.setExpectedContentType("text/html");
 8    s.doGet(request, response);
 9    response.verify();
10    assertEquals("Invalid temperature: boo!\r\n",
11                 response.getOutputStreamContents());
12 }
13
14 public void test_boil() throws Exception {
15    TemperatureServlet s = new TemperatureServlet();
16    MockHttpServletRequest request =
        new MockHttpServletRequest();
17    MockHttpServletResponse response =
        new MockHttpServletResponse();
18
19    request.setupAddParameter("Fahrenheit", "212");
20    response.setExpectedContentType("text/html");
21    s.doGet(request, response);
22    response.verify();
23    assertEquals("Fahrenheit: 212, Celsius: 100.0\r\n",
24                 response.getOutputStreamContents());
25 }
```

Figure 2. Mock objects in action—a typical set of tests using the JUnit testing framework.

text that our servlet wants to display on the browser. We can query this value (lines 10 and 23) to check that we're returning the text we were expecting.

Mock objects

It's likely that we've all been using mock objects for years without knowing it. However, it's also likely that we've used them only on an ad hoc basis, coding up stubs when we needed them. However, we personally have recently started benefiting from adopting a more systematic approach to creating mock objects. Even things as simple as consistent naming schemes have helped make our tests more readable and the mock objects themselves more portable from project to project.

There are several mock object frameworks to choose from. Three for Java are at www.c2.com/cgi/wiki?MockObject, and a fine implementation for Ruby is at www.b13media.com/dev/ruby/mock.html. If the thought of writing all the mock object classes you might need is intimidating, look at EasyMock (www.easymock.org), a convenient Java API for creating mock objects dynamically. All these implementations are a starting point; you'll probably need to add new mock object implementations to stub out real objects in your environment.

There are also alternatives to mock objects in the servlet environment. In particular, the Jakarta Cactus system (http://jakarta.apache.org/cactus) is a heavier-weight framework for testing server-side components. Compared to the mock-objects approach, Cactus runs your tests in the actual target environment and tends to produce less fine-grained tests. Depending on your needs, this might or might not be a good thing.

A funny thing happens when you start using mock objects. As with other low-level testing practices, you might find that your code becomes not only better tested but also better designed and easier to understand (Nat Pryce discusses this in the sidebar). Mock objects won't solve all your development problems, but they are exceptionally sharp tools to have in your toolbox.

Practical Experience with Mock Objects
Nat Pryce

My experience is that using mock objects and a test-first methodology forces you to think about design differently from traditional object-oriented design techniques. First, you end up with many small, decoupled classes that are used through composition, rather than inheritance. These classes implement interfaces more than they inherit state and behavior. Second, you think about object interfaces in terms of the services that an object both provides and *requires* from its environment, making an object's requirements explicit. This is different from traditional OO design methods that concentrate only on an object's provided services and try to hide an object's requirements through encapsulation of private-member variables. Third, you end up thinking more explicitly in terms of interobject protocols. Those protocols are often defined using interface definitions and tested using mock objects before a concrete implementation is produced.

Nat Pryce is the technical director at B13media Ltd. Contact him at nat.pryce@b13media.com; www.b13media.com.

Dave Thomas and **Andy Hunt** are partners in The Pragmatic Programmers, LLC. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. They also offer training in modern development techniques to programmers and their management. They are coauthors of *The Pragmatic Programmer* and *Programming Ruby*, both from Addison-Wesley. Contact them via www.pragmaticprogrammer.com.