Extracted from:

Pragmatic Unit Testing

in Java with JUnit

This PDF file contains pages extracted from Pragmatic Unit Testing, one of the Pragmatic Starter Kit series of books for project teams. For more information, visit `http://www.pragmaticprogrammer.com/starter_kit`.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Chapter 6

Using Mock Objects

The objective of unit testing is to exercise just one method at a time, but what happens when that method depends on other things—hard-to-control things such as the network, or a database, or even a servlet engine?

What if your code depends on other parts of the system—maybe even *many* other parts of the system? If you're not careful, you might find yourself writing tests that end up initializing nearly every system component just to give the tests enough context in which to run. Not only is this time consuming, it also introduces a ridiculous amount of coupling into the testing process: someone goes and 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 will become discouraged after this happens a few times, and eventually may abandon all testing. But there are techniques we can use to help.

In movie and television production, crews will often use *stand-ins* or *doubles* for the real actors. In particular, while the crews are setting up the lights and camera angles, they'll use *lighting doubles*: inexpensive, unimportant people who are about the same height and complexion as the expensive, important actors lounging safely in their luxurious trailers.

The crew then tests their setup with the lighting doubles, measuring the distance from the camera to the stand-in's nose, adjusting the lighting until there are no unwanted shad-

ows, and so on, while the obedient stand-in just stands there and doesn't whine or complain about "lacking motivation" for their character in this scene.

So what we're going to do in unit testing is similar to the use of lighting doubles in the movies: we'll use a cheap stand-in that is kind of close to the real thing, at least superficially, but that will be easier to work with for our purposes.

6.1 Simple Stubs

What we need to do is to stub out all those uncooperative parts of the rest of the real world and replace each of them with a more complicit ally—our own version of a "lighting double." For instance, perhaps we don't want to test against the real database, or with the real, current, wall-clock time. Let's look at a simple example.

Suppose throughout your code you call your own `getTime()` method to return the current time. It might be defined to look something like this:

```
public long getTime() {
  return System.currentTimeMillis();
}
```

(In general, we usually suggest wrapping calls to facilities outside the scope of the application to better encapsulate them—and this is a good example.) Since the concept of current time is wrapped in a method of your own writing, you can easily change it to make debugging a little easier:

```
public long getTime() {
  if (debug) {
    return debug_cur_time;
  } else {
    return System.currentTimeMillis();
  }
}
```

You might then have other debug routines to manipulate the system's idea of "current time" to cause events to happen that you'd have to wait around for otherwise.

This is one way of stubbing out the real functionality, but it's messy. First of all, it only works if the code consistently calls your own `getTime()` and does not call the Java method

`System.currentTimeMillis()` directly. What we need is a slightly cleaner—and more object-oriented—way to accomplish the same thing.

6.2 Mock Objects

Fortunately, there's a testing pattern that can help: *mock objects*. A mock object is simply a debug replacement for a real-world object. There are a number of situations that come up where mock objects can help us. Tim Mackinnon [MFC01] offers the following list:

- The real object has nondeterministic behavior (it produces unpredictable results, like a stock-market quote feed.)
- 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 confirm that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

Using mock objects, we can get around all of these problems. The three key steps to using mock objects for testing are:

1. Use an interface to describe the object
2. Implement the interface for production code
3. Implement the interface in a mock object for testing

The code under test only ever refers to the object by it's interface, so it can remain blissfully ignorant as to whether it is using the real object or the mock. Let's take another look at our time example. We'll start by creating an interface for

a number of real-world environmental things, one of which is the current time:

Environmental.java

```
public interface Environmental {
  public long getTime();
  // Other methods omitted...
}
```

Next, we create the real implementation:

SystemEnvironment.java

```
public class SystemEnvironment implements Environmental {
  public long getTime() {
    return System.currentTimeMillis();
  }
  // other methods ...
}
```

And finally, the mock implementation:

MockSystemEnvironment.java

```
public class MockSystemEnvironment
  implements Environmental {
  public long getTime() {
    return current_time;
  }
  public void setTime(long aTime) {
    current_time = aTime;
  }
  private long current_time;
  // ...
}
```

Note that in the mock implementation, we've added the additional method `setTime()` (and the corresponding private variable) that allows you to control the mock object.

Now suppose we've written a new method that depends on the `getTime()` method. Some details are omitted, but the part we're interested in looks like this:

```
Line 1  import java.util.Calendar;
-
-       public class Checker {
-
5         public Checker(Environmental anEnv) {
-           env = anEnv;
-         }
-
-         /**
10        * After 5 o'clock, remind people to go home
-         * by playing a whistle
-         */
-         public void reminder() {
-           Calendar cal = Calendar.getInstance();
15         cal.setTimeInMillis(env.getTime());
-           int hour = cal.get(Calendar.HOUR_OF_DAY);
-
-           if (hour >= 17) { // 5:00PM
```

```
        env.playWavFile("quit_whistle.wav");
      }

    }

    // ...

    private Environmental env;
  }
```

Checker.java

In the production environment—the real world code that gets shipped to customers—an object of this class would be initialized by passing in a real `SystemEnvironment`. The test code, on the other hand, uses a `MockSystemEnvironment`.

The code under test that uses `env.getTime()` doesn't know the difference between a test environment and the real environment, as they both implement the same interface. You can now write tests that exploit the mock object by setting the time to known values and checking for the expected behavior.

In addition to the `getTime()` call that we've shown, the `Environmental` interface also supports a `playWavFile()` method call (used on line 19 in `Checker.java` above). With a bit of extra support code in our mock object, we can also add tests to see if the `playWavFile()` method was called without having to listen to the computer's speaker.

```
public void playWavFile(String filename) {
  playedWav = true;
}
public boolean wavWasPlayed() {
  return playedWav;
}
public void resetWav() {
  playedWav = false;
}
private boolean playedWav = false;
```

MockSystemEnvironment.java

Putting all of this together, a test using this setup would go something like this:

```
Line 1  import junit.framework.*;
        import java.util.Calendar;

        public class TestChecker extends TestCase {
    5
          public void testQuittingTime() {

            MockSystemEnvironment env =
              new MockSystemEnvironment();
   10
```

```
        // Set up a target test time
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.YEAR, 2004);
        cal.set(Calendar.MONTH, 10);
        cal.set(Calendar.DAY_OF_MONTH, 1);
        cal.set(Calendar.HOUR_OF_DAY, 16);
        cal.set(Calendar.MINUTE, 55);
        long t1 = cal.getTimeInMillis();

        env.setTime(t1);

        Checker checker = new Checker(env);

        // Run the checker
        checker.reminder();

        // Nothing should have been played yet
        assertFalse(env.wavWasPlayed());

        // Advance the time by 5 minutes
        t1 += (5 * 60 * 1000);
        env.setTime(t1);

        // Now run the checker
        checker.reminder();

        // Should have played now
        assertTrue(env.wavWasPlayed());

        // Reset the flag so we can try again
        env.resetWav();

        // Advance the time by 2 hours and check
        t1 += 2 * 60 * 60 * 1000;
        env.setTime(t1);

        checker.reminder();
        assertTrue(env.wavWasPlayed());
    }
}
```

TestChecker.java

The code creates a mock version of the application environment at line 9. Lines 12 though 20 set up the fake time that we'll use, and then sets that in the mock environment object.

By line 25 we can run the `reminder()` call, which will (unwittingly) use the mock environment. The `assert` on line 28 makes sure that the `.wav` file has *not* been played yet, as it is not yet quitting time in the mock object environment. But we'll fix that in short order; line 32 puts the mock time exactly equal to quitting time (a good boundary condition, by the way). The `assert` on line 38 makes sure that the `.wav` file did play this time around.

Finally, we'll reset the mock environment's `.wav` file flag at line 41 and test a time two hours later. Notice how easy it is to alter and check conditions in the mock environment—you don't have to bend over and listen to the PC's speaker, or reset

the clock, or pull wires, or anything like that.

Because we've got an established interface to all system functions, people will (hopefully) be more likely to use it instead of calling methods such as `System.currentTimeMillis()` directly, and we now have control over the behavior behind that interface.

And that's all there is to mock objects: faking out parts of the real world so you can concentrate on testing your own code easily. Let's look at a more complicated example next.

6.3 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,[1] 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.

The listing below shows part of the source of a trivial servlet that converts temperatures from Fahrenheit to Celsius. Let's step quickly through its operation.

```
Line 1   public void doGet(HttpServletRequest req,
                         HttpServletResponse res)
           throws ServletException, IOException
         {
           String str_f = req.getParameter("Fahrenheit");

           res.setContentType("text/html");
           PrintWriter out = res.getWriter();

         10  try {
               int temp_f = Integer.parseInt(str_f);
               double temp_c = (temp_f - 32)*5.0 /9.0;
               out.println("Fahrenheit: " + temp_f +
                           ", Celsius: " + temp_c);
         15  } catch (NumberFormatException e) {
               out.println("Invalid temperature: " + str_f);
             }
         }
```

TemperatureServlet.java

When the servlet container receives the request, it automatically invokes the servlet method `doGet()`, passing in two parameters: a request and a response.

[1] `http://jakarta.apache.org/tomcat`

The request parameter contains information about the request. The servlet uses this parameter to get the contents of the field Fahrenheit. It then converts the value to Celsius before writing the result back to the user. (The response object contains a factory method that returns a `PrintWriter` object, which does the actual writing.) If an error occurs converting the number (perhaps the user typed "boo!" instead of a temperature into the form's temperature field), we catch the exception and report the error in the response.

This snippet of code runs in a fairly complex environment: it needs 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. Mock objects to the rescue!

The interface to the servlet code 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()`.

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 code to compile—it's not as easy as the slightly contrived time/wav-file example above. Fortunately, other folks have already done the hard work for us.

Mackinnon, Freeman, and Craig [MFC01] introduced the formalization of mock objects and have also developed the code for a mock object framework for Java programmers.[2] In addition to the basic framework code that makes it easier to develop mock objects, the mock objects package comes with a number of application-level mock objects.

You'll find mock output objects (including `PrintStream`, and `PrintWriter`), objects that mock the `java.sql` library, and

[2] http://www.mockobjects.com

classes for testing in a servlet environment. In particular, it 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 their package to rig the tests, much as we faked out setting the time in the earlier example:

```
Line 1   import junit.framework.*;
  -      import com.mockobjects.servlet.*;
  -
  -      public class TestTempServlet extends TestCase {
  5
  -        public void test_bad_parameter() throws Exception {
  -          TemperatureServlet s = new TemperatureServlet();
  -          MockHttpServletRequest request =
  -            new MockHttpServletRequest();
 10         MockHttpServletResponse response =
  -            new MockHttpServletResponse();
  -
  -          request.setupAddParameter("Fahrenheit", "boo!");
  -          response.setExpectedContentType("text/html");
 15         s.doGet(request,response);
  -          response.verify();
  -          assertEquals("Invalid temperature: boo!\n",
  -                       response.getOutputStreamContents());
  -        }
 20
  -        public void test_boil() throws Exception {
  -          TemperatureServlet s = new TemperatureServlet();
  -          MockHttpServletRequest request =
  -            new MockHttpServletRequest();
 25         MockHttpServletResponse response =
  -            new MockHttpServletResponse();
  -
  -          request.setupAddParameter("Fahrenheit", "212");
  -          response.setExpectedContentType("text/html");
 30         s.doGet(request,response);
  -          response.verify();
  -          assertEquals("Fahrenheit: 212, Celsius: 100.0\n",
  -                       response.getOutputStreamContents());
  -        }
 35
  -      }
```

TestTempServlet.java

We use a `MockHttpServletRequest` object to set up the context in which to run the test. On line 13 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 in the browser; our mock object eliminates the need for human input when the test runs.

On line 14, we tell the response object that we expect the method under test to set the response’s content type to be `text/html`. Then, on lines 16 and 31, 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, and they can check that methods have been called the correct number of times.

Mock objects can also record the data that was given to them. In our case, the response object receives the text that our servlet wants to display on the browser. We can query this value (lines 18 and 33) to check that we're returning the text we were expecting.

6.4 Easy Mock Objects

If the thought of writing all these mock object classes is intimidating, you might want to take a look at Easy-Mock,[3] a convenient Java API for creating mock objects dynamically.

Easy-Mock uses a very interesting method to specify which method calls to a mocked-out interface are allowed and what their return values should be: you specify which method calls should exist by calling them! The mock control object lets you specify a *record mode* and a *replay mode* for the corresponding mock object. While the object is in record mode, you go ahead and call the methods you are interested in and set the desired return values. Once that's finished, you switch over to replay mode. Now when you call those methods, you'll get the return values you specified.

Any remaining methods will throw a runtime exception if they are called—but the nice part is that you don't have to define any of that.

For example, suppose we have an interface for a Jukebox hardware controller that has over a dozen methods, but we're only interested in one for this demonstration. (Note that we're not really testing anything here, but are just showing how you set up and use an EasyMock object. Obviously, the whole

[3] `http://www.easymock.org`

point of using a mock object is to allow you to test something that depends on the object you're mocking up.)

```
Line 1   import junit.framework.*;
         import org.easymock.MockControl;

         public class TestJukebox extends TestCase {
       5
           private Jukebox     mockJukebox;
           private MockControl mockJukebox_control;

           protected void setUp() {
         10  // Create a control handle to the Mock object
             mockJukebox_control =
               MockControl.createControl(Jukebox.class);

             // And create the Mock object itself
         15  mockJukebox =
               (Jukebox) mockJukebox_control.getMock();
           }

           public void testEasyMockDemo() {
         20
             // Set up the mock object by calling
             // methods you want to exist
             mockJukebox.getCurrentSong();
             mockJukebox_control.setReturnValue(
         25                      "King Crimson -- Epitaph");

             // You don't have to worry about the other dozen
             // methods defined in Jukebox...

         30  // Switch from record to playback
             mockJukebox_control.replay();

             // Now it's ready to use:
             assertEquals("King Crimson -- Epitaph",
         35                mockJukebox.getCurrentSong());
           }

         }
```

TestJukebox.java

The code in the `setUp()` method for this test creates an empty mock object for the `Jukebox` interface, along with its control. The control starts off in record mode, so the call on line 23 will create a mock stub for the `getCurrentSong()` call that we want to use. The next line sets the return value for that method—that's all it takes.

Now we switch the mock object from record to replay mode (line 31), and finally the call on line 35 returns the value we just set up.

There are many other options; you can specify how many times a method should return a particular value, you can verify that methods which return void were actually called, and so on.

There are also alternatives to mock objects, particularly in the servlet environment. The Jakarta Cactus system[4] 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.

Exercises

7. Come up with a simple mock object (by hand) for an MP3 player control panel with the following methods:

Answer on 144

```
import java.util.ArrayList;
public interface Mp3Player {
  /**
   * Begin playing the filename at the top of the
   * play list, or do nothing if playlist
   * is empty.
   */
  public void play();
  /**
   * Pause playing. Play will resume at this spot.
   */
  public void pause();
  /**
   * Stop playing. The current song remains at the
   * top of the playlist, but rewinds to the
   * beginning of the song.
   */
  public void stop();
  /** Returns the number of seconds into
   * the current song.
   */
  public double currentPosition();
  /**
   * Returns the currently playing file name.
   */
  public String currentSong();
  /**
   * Advance to the next song in the playlist
   * and begin playing it.
   */
  public void next();
  /**
   * Go back to the previous song in the playlist
   * and begin playing it.
   */
  public void prev();
```

[4] http://jakarta.apache.org/cactus

```
  /**
   * Returns true if a song is currently
   * being played.
   */
  public boolean isPlaying();
  /**
   * Load filenames into the playlist.
   */
  public void loadSongs(ArrayList names);
}
```

Mp3Player.java

It should pass the following unit test:

```
import junit.framework.*;
import java.util.ArrayList;
public class TestMp3Player extends TestCase {
  protected Mp3Player mp3;
  protected ArrayList list = new ArrayList();
  public void setUp() {
    mp3 = new MockMp3Player();
    list = new ArrayList();
    list.add("Bill Chase -- Open Up Wide");
    list.add("Jethro Tull -- Locomotive Breath");
    list.add("The Boomtown Rats -- Monday");
    list.add("Carl Orff -- O Fortuna");
  }
  public void testPlay() {
    mp3.loadSongs(list);
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertTrue(mp3.isPlaying());
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.pause();
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.stop();
    assertEquals(mp3.currentPosition(), 0.0, 0.1);
  }
  public void testPlayNoList() {
    // Don't set the list up
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertFalse(mp3.isPlaying());
    assertEquals(mp3.currentPosition(), 0.0, 0.1);
    mp3.pause();
    assertEquals(mp3.currentPosition(), 0.0, 0.1);
    assertFalse(mp3.isPlaying());
    mp3.stop();
    assertEquals(mp3.currentPosition(), 0.0, 0.1);
    assertFalse(mp3.isPlaying());
  }
  public void testAdvance() {
    mp3.loadSongs(list);
    mp3.play();
    assertTrue(mp3.isPlaying());
```

```
        mp3.prev();
        assertEquals(mp3.currentSong(), list.get(0));
        assertTrue(mp3.isPlaying());
        mp3.next();
        assertEquals(mp3.currentSong(), list.get(1));
        mp3.next();
        assertEquals(mp3.currentSong(), list.get(2));
        mp3.prev();
        assertEquals(mp3.currentSong(), list.get(1));
        mp3.next();
        assertEquals(mp3.currentSong(), list.get(2));
        mp3.next();
        assertEquals(mp3.currentSong(), list.get(3));
        mp3.next();
        assertEquals(mp3.currentSong(), list.get(3));
        assertTrue(mp3.isPlaying());
    }
}
```

TestMp3Player.java