Using Hammock's Spy Objects

Test Doubles for Java ME

Carl Meijer

June 2009

1 Spy Objects

1.1 Introduction

Any unit test consists of the following:

- Set up code (possibly in a `setUp()` fixture for several tests).
- Code that exercises the object under test (OUT).
- Code that verifies that the OUT behaved correctly (typically with `assert()` statements).
- Possibly some `tearDown` code.

When one compares tests using mock objects to those that don't, one will see that the verification code is greatly reduced. Frequently, when using mocks, the verification code may consist of a single call to a verify method. By comparison, testing with real collaborators typically involves far more verification code; for example, there may be several `assert`s to verify that the collaborator states have been changed correctly by the OUT. The reduced verification code when using mocks may come at the expense of more set up code though; expectations have to be specified for the mocks (although configuring real collaborators can also be code-intensive).

The use of *spy objects* rather than mock objects has the benefit of using test doubles but can result in test code that more closely resembles tests using real collaborators. The coding idiom associated with spy objects is sometimes known by the acronym AAA for Arrange-Act-Assert. "Arrange" refers to setting up the test, "Act" refers to exercising the OUT and "Assert" refers to the use of assertions to verify interactions with test doubles.

For testing in Java, Mockito is the dominant test double framework that uses spy objects.

Hammock supports a method invocation handler, `Hamspy`, that provides support for the AAA idiom.

This document doesn't make the claim that spy objects are better or worse than mock objects; the choice of a coding idiom is a matter of preference.

2 A Spy Object Example

The example can be found in the `com.hammingweight.hammockexamples.example5` package of the `examples` directory of the Hammock distribution.

For this example we want to test a class (that extends LCDUI `List`) that must be populated with a list of previously made flight reservations. The `ViewBookingsList` class collaborates with two other classes: A `Logger` class that we use for logging our application's activities and a `FlightBookingDAO` class that allows us to retrieve details about airline reservations.

2.1 The Collaborating Classes

2.1.1 The Logger Interface

Listing 1 defines our `Logger` interface that allows us to log at two levels: debug and production.

Listing 1

```
package com.hammingweight.hammockexamples.example5;

public interface Logger {

    public void debug(String s);

    public void debug(String s, Throwable e);

    public void production(String s);

    public void production(String s, Throwable e);

}
```

As an aside, J2ME Polish has rather nice preprocessor directives for logging. MicroLog (texttthttp://microlog.sourceforge.net looks like it might be very nice as well.

2.1.2 The FlightBookingDAO Interface

Listing 2 defines the `FlightBookingDAO` that allows us to get an array of `FlightBooking`s.

Listing 2

```
package com.hammingweight.hammockexamples.example5;

public interface FlightBookingDAO {

    public FlightBooking[] getAllBookings();

}
```

The `FlightBooking` class is a very simple immutable class as shown in listing 3.

Listing 3

```
package com.hammingweight.hammockexamples.example5;

public class FlightBooking {

    private String flightNum;

    private String dateTime;

    private String reservationCode;

    private String orig;

    private String dest;

    public FlightBooking(String flightNum, String dateTime,
            String reservationCode, String orig, String dest) {
```

```
        this.flightNum = flightNum;
        this.dateTime = dateTime;
        this.reservationCode = reservationCode;
        this.orig = orig;
        this.dest = dest;
    }

    public String getDateTime() {
        return this.dateTime;
    }

    public String getFlightNum() {
        return this.flightNum;
    }

    public String getReservationCode() {
        return this.reservationCode;
    }

    public String getOrig() {
        return this.orig;
    }

    public String getDest() {
        return this.dest;
    }

}
```

2.2 The ViewBookingsList

Class Listing 4 shows the skeleton of the class that we're going to test.

Listing 4

```
package com.hammingweight.hammockexamples.example5;

import javax.microedition.lcdui.List;

public class ViewBookingsList extends List {

    public ViewBookingsList(Logger log, FlightBookingDAO bookingDao) {
        super("Your Bookings", List.IMPLICIT);
    }

}
```

2.3 Spying on an Interaction

We want to verify that the constructor of the `ViewBookingsList` logs, at the debug level, when it is invoked. Listing 5 shows a test case that `assert`s that there is one call to the `debug()` method of our mocked `Logger`.

Listing 5

```
package com.hammingweight.hammockexamples.example5;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hamspy;
import com.hammingweight.hammock.MethodInvocation;
import com.hammingweight.hammockexamples.mocks.MockFlightBookingDAO;
import com.hammingweight.hammockexamples.mocks.MockLogger;

public class ViewBookingsListTest extends TestCase {

    // The spy handler
    private Hamspy hamspy;

    // The mock object(s)
    private MockLogger mockLogger;
    private MockFlightBookingDAO mockBookingDao;

    public void setUp() {
        this.hamspy = new Hamspy();
        this.mockLogger = new MockLogger(this.hamspy);
        this.mockBookingDao = new MockFlightBookingDAO(this.hamspy);
    }

    public void testConstructor() {
        // Arrange

        // Act
        ViewBookingsList bookingsList = new ViewBookingsList(this.mockLogger,
                this.mockBookingDao);
```

```
            // Asserts.
            assertEquals(1, this.hamspy
                    .getInvocations(MockLogger.MTHD_DEBUG_$_STRING).length);
        }

        public ViewBookingsListTest() {
            super(1, "ViewBookingsListTest");
        }

        public void test(int testNum) throws Throwable {
            switch (testNum) {
            case 0:
                testConstructor();
                break;

            default:
                fail("No such test.");
            }
        }

    }
```

There are several important points to note in listing 5:

- We use an instance of `Hamspy` as the method invocation handler for our test doubles.
- The `MockLogger` and `MockFlightBookingDAO` are associated with the `Hamspy` instance.
- Our `testConstructor` test case does not set an expectation that the `debug()` method will be invoked on our `Logger`.
- The `Hamspy` class exposes a method `getInvocations()` that returns an array of method invocations. By looking at the length of the array, we can see how many times a method was invoked.

If we run the test it fails with the exception

```
jmunit.framework.cldc10.AssertionFailedException: Assert equals failed.
Expected 1, but was 0.
```

The failure reflects the fact that the `debug()` method was invoked zero times rather than once. It's easy to change the code of listing 4 so that the test passes; see listing 6.

Listing 6

```
package com.hammingweight.hammockexamples.example5;

import javax.microedition.lcdui.List;

public class ViewBookingsList extends List {

    public ViewBookingsList(Logger log, FlightBookingDAO bookingDao) {
        super("Your Bookings", List.IMPLICIT);
        log.debug(null);
    }

}
```

Of course, writing `null` to the log is unhelpful. We modify our test (listing 7) to assert that a descriptive message is logged.

Listing 7

```
public void testConstructor() {
    // Arrange

    // Act
    ViewBookingsList bookingsList = new ViewBookingsList(this.mockLogger,
            this.mockBookingDao);

    // Asserts.
```

```
        MethodInvocation[] mi = this.hamspy
                .getInvocations(MockLogger.MTHD_DEBUG_$_STRING);
        assertEquals(1, mi.length);
        assertEquals("ViewBookingsList constructed.", mi[0]
                .getMethodArguments()[0]);
    }
```

In the previous test we used the `getInvocations()` method to retrieve all invocations of the `debug()` method. The `MethodInvocation` class allows us to get the arguments passed in a method call via `getMethodArguments()`. While we can explicitly use the `getMethodArguments()` to get a method's arguments so that we can check them, the next section will introduce the `InvocationVerifier` class that makes writing these `asserts` easier (especially if a method takes several arguments).

Naturally, our test now fails with an exception:

```
Assert equals failed. Expected ViewBookingsList constructed., but
was null.
```

Listing 8 shows the corrected class under test.

Listing 8

```
package com.hammingweight.hammockexamples.example5;

import javax.microedition.lcdui.List;

public class ViewBookingsList extends List {

    public ViewBookingsList(Logger log, FlightBookingDAO bookingDao) {
        super("Your Bookings", List.IMPLICIT);
        log.debug("ViewBookingsList constructed.");
    }

}
```

2.4 Setting Expectations

We should check that our `List` is correctly populated with flights. We'll assume that our menu should be populated with three flight reservation codes. Listing 9 shows the `asserts` we've added to check the codes.

Listing 9

```
public void testConstructor() {
    // Arrange

    // Act
    ViewBookingsList bookingsList = new ViewBookingsList(this.mockLogger,
            this.mockBookingDao);

    // Asserts.
    MethodInvocation[] mi = this.hamspy
            .getInvocations(MockLogger.MTHD_DEBUG_$_STRING);
    assertEquals(1, mi.length);
    assertEquals("ViewBookingsList constructed.", mi[0]
            .getMethodArguments()[0]);
    assertEquals("ABC123", bookingsList.getString(0));
    assertEquals("DEF456", bookingsList.getString(1));
    assertEquals("GHI789", bookingsList.getString(2));
}
```

Listing 10 shows the code that we implement in the `ViewBookingsList` class to get the reservation codes from the collaborating DAO.

Listing 10

```
package com.hammingweight.hammockexamples.example5;

import javax.microedition.lcdui.List;

public class ViewBookingsList extends List {

    public ViewBookingsList(Logger log, FlightBookingDAO bookingDao) {
        super("Your Bookings", List.IMPLICIT);
```

```
            log.debug("ViewBookingsList constructed.");

            for (int i = 0; i < bookingDao.getAllBookings().length; i++) {
                append(bookingDao.getAllBookings()[i].getReservationCode(), null);
            }
        }

    }
```

Running the test fails with a `NullPointerException`. The reason is that spy objects, while powerful, are not psychic. Our test double of the `FlightBookingDAO` must return an `Object` when the `getAllBookings()` method is invoked. Since the test double doesn't know what value to return, it returns `null`. Hammock's spy objects always return a default value if no expectation has been explicitly set (for primitive numeric types the default value is zero, the default `boolean` value is `false` and the default value for any `Object` is `null`).

To fix our test we need to an expectation or, in the language of the AAA idiom, we need to `arrange` our test. Listing 11 shows an attempt to fix the test:

Listing 11

```
public void testConstructor() {
    // Arrange
    FlightBooking[] flights = {
            new FlightBooking("", "", "ABC123", "", ""),
            new FlightBooking("", "", "DEF456", "", ""),
            new FlightBooking("", "", "GHI789", "", "") };
    this.hamspy.setExpectation(MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS)
            .setReturnValue(flights);

    // Act
    ViewBookingsList bookingsList = new ViewBookingsList(this.mockLogger,
            this.mockBookingDao);

    // Asserts.
    MethodInvocation[] mi = this.hamspy
            .getInvocations(MockLogger.MTHD_DEBUG_$_STRING);
    assertEquals(1, mi.length);
    assertEquals("ViewBookingsList constructed.", mi[0]
            .getMethodArguments()[0]);
    assertEquals("ABC123", bookingsList.getString(0));
    assertEquals("DEF456", bookingsList.getString(1));
    assertEquals("GHI", bookingsList.getString(2));
}
```

Unfortunately our test still fails with a `NullPointerException` since Hammock interprets the `setExpectation()` method to set an expectation that a method will be invoked exactly once. Since the `getAllBookings()` should be invoked seven times, `null` is returned on the second invocation of the method.

We could change our code to set that we expect the method to be invoked seven times as

```
this.hamspy.setExpectation(MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS)
        .setReturnValue(flights).setInvocationCount(7);
```

However, the above is not a good change since we could (and should!) refactor our code so that `getAllBookings()` is invoked only once. Since we want all invocations of the method to respond identically we can set a stub expectation as shown in listing 12. A stub expectation responds identically an arbitrary number of times (possibly zero).

Listing 12

```
public void testConstructor() {
    // Arrange
    FlightBooking[] flights = {
            new FlightBooking("", "", "ABC123", "", ""),
            new FlightBooking("", "", "DEF456", "", ""),
            new FlightBooking("", "", "GHI789", "", "") };
    this.hamspy.setStubExpectation(
            MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS).setReturnValue(
            flights);

    // Act
```

```
        ViewBookingsList bookingsList = new ViewBookingsList(this.mockLogger,
                this.mockBookingDao);

        // Asserts.
        MethodInvocation[] mi = this.hamspy
                .getInvocations(MockLogger.MTHD_DEBUG_$_STRING);
        assertEquals(1, mi.length);
        assertEquals("ViewBookingsList constructed.", mi[0]
                .getMethodArguments()[0]);
        assertEquals("ABC123", bookingsList.getString(0));
        assertEquals("DEF456", bookingsList.getString(1));
        assertEquals("GHI789", bookingsList.getString(2));
        assertTrue(this.hamspy
                .getInvocationCount(MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS) >= 1);
    }
```

2.5 Using an InvocationVerifier Class

We can use the `getMethodArguments()` method to retrieve the arguments passed to a method that was spied on and then validate each argument with an `assert`. This can lead to verbose tests; it is better to get an InvocationVerifier instance from the spy handler. The `Hamspy` class exposes a method, `getExpectation()`, that returns an `InvocationVerifier` object that we can use to verify that a mocked collaborator was invoked as expected. The name "getExpectation()" was chosen to indicate a symmetry with mock objects; with mock objects we *set* our expectations using a `setExpectation()` method *before* exercising the OUT; when using spy objects we need to *get* an object for verifying our expectations *after* exercising the OUT.

The test case below illustrates a test that we write to verify that the `debug()` method is invoked on the mock Logger if a DAO throws an exception.

Listing 13

```
    public void testLogException() {
        // Arrange
        this.hamspy.setStubExpectation(
                MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS).setThrowable(
                new RuntimeException());

        // Act
        try {
            new ViewBookingsList(this.mockLogger, this.mockBookingDao);
            fail("RuntimeException should have been thrown.");
        } catch (RuntimeException e) {
            // expected.
        }

        // Asserts.
        this.hamspy
                .getExpectation(
                        MockLogger.MTHD_DEBUG_$_STRING_THROWABLE,
                        new Object[] { "DAO threw exception.",
                                new RuntimeException() }).setArgumentMatcher(1,
                        new ClassArgumentMatcher()).setInvocationCount(1)
                .verify();
    }
```

When we run the test, it fails:

```
com.hammingweight.hammock.HammockException: A method was invoked less often
than expected.
        Class:  MockLogger
        Method: MTHD_DEBUG_$_STRING_THROWABLE
```

Notice in the test that we can pass both arguments to be verified in a single call rather than using two `assert`s. Also notice that we needed to use an argument matcher to validate the second argument (the `ClassArgumentMatcher` regards two arguments as equal if the class of the actual argument can be cast to the class of the expected argument).

If you really like `asserts` and would prefer to see a test failure, rather than an error, you can use the `isVerified()` method instead of the `verify()` method

(see Listing 14). This allows your test to use an `assert`. Note though that the exceptions thrown by `verify` are more informative than the `false` value returned by `isVerified()`. Unless you really think that `assert`s make your code's intention clearer, use `verify()` rather than `isVerified()`.

Listing 14

```
public void testLogException() {
    // Arrange
    this.hamspy.setStubExpectation(
            MockFlightBookingDAO.MTHD_GET_ALL_BOOKINGS).setThrowable(
            new RuntimeException());

    // Act
    try {
        new ViewBookingsList(this.mockLogger, this.mockBookingDao);
        fail("Runtime Exception should have been thrown.");
    } catch (RuntimeException e) {
        // expected.
    }

    // Asserts.
    assertTrue(this.hamspy
            .getExpectation(
                    MockLogger.MTHD_DEBUG_$_STRING_THROWABLE,
                    new Object[] { "DAO threw exception.",
                            new RuntimeException() }).setArgumentMatcher(1,
                    new ClassArgumentMatcher()).setInvocationCount(1)
            .isVerified());
}
```

The `MethodInvocationVerifier` class exposes `setInvocationCount()` methods that allow us to specify how many times we expect a method to be invoked. If we don't explicitly set the invocation count, the verification will succeed as long as the method is invoked at least once.

Finally, listing 15 shows the `ViewBookingsList` class that has been refactored to pass both tests.

Listing 15

```
package com.hammingweight.hammockexamples.example5;

import javax.microedition.lcdui.List;

public class ViewBookingsList extends List {

    public ViewBookingsList(Logger log, FlightBookingDAO bookingDao) {
        super("Your Bookings", List.IMPLICIT);
        log.debug("ViewBookingsList constructed.");

        FlightBooking[] bookings = null;
        try {
            bookings = bookingDao.getAllBookings();
        } catch (RuntimeException e) {
            log.debug("DAO threw exception.", e);
            throw e;
        }
        for (int i = 0; i < bookings.length; i++) {
            append(bookings[i].getReservationCode(), null);
        }
    }

}
```

3 Spying on Concrete Classes

In the example of the previous section, the test doubles that we used implemented an interface. Sometimes though we will want to spy on a class with non-abstract methods. This raises an interesting question: Should the spy object delegate invocations to the parent class or should it respond with default responses as is the case when spying on an (abstract) interface. Listing 16, in which we spy on a `Vector` instance, should make the question clearer.

Listing 16

```
package com.hammingweight.hammockexamples.example6;
```

```
import java.util.Vector;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hamspy;
import com.hammingweight.hammock.mocks.util.MockVector;

public class ConcreteSpyTest extends TestCase {

    public void testDelegateToParent() {
        Vector spyVector = new MockVector(new Hamspy());
        spyVector.addElement("foo");
        assertEquals(1, spyVector.size());
        assertEquals("foo", spyVector.elementAt(0));
    }

    public void testReturnDefaultValues() {
        Vector spyVector = new MockVector(new Hamspy());
        spyVector.addElement("foo");
        assertEquals(0, spyVector.size());
        assertNull(spyVector.elementAt(0));
    }

    public ConcreteSpyTest() {
        super(2, "ConcreteSpyTest");
    }

    public void test(int testNum) throws Throwable {
        switch (testNum) {
        case 0:
            testDelegateToParent();
            break;

        case 1:
            testReturnDefaultValues();
            break;

        default:
            fail("No such test.");
        }
    }

}
```

The question is: Does the first or the second test pass? The answer is: The first; the spy object behaves identically to its parent class. The second test fails with the exception:

```
Assert equals failed. Expected 0, but was 1.
```

However, you might prefer it if the behavior allowed the second test to pass instead. In fact, Mockito's test doubles of concrete classes adopt the second idiom: always return default values even when mocking a non-abstract method. Fortunately, Hammock allows you to override the default behavior by using an overloaded constructor that takes a `boolean` argument; if the argument is `true` the test doubles will return default values (unless an expectation has been explicitly set). Listing 17 shows the corrected test.

Listing 17

```
package com.hammingweight.hammockexamples.example6;

import java.util.Vector;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hamspy;
import com.hammingweight.hammock.mocks.util.MockVector;

public class ConcreteSpyTest extends TestCase {

    public void testDelegateToParent() {
        Vector spyVector = new MockVector(new Hamspy());
        spyVector.addElement("foo");
        assertEquals(1, spyVector.size());
        assertEquals("foo", spyVector.elementAt(0));
    }

    public void testReturnDefaultValues() {
        Hamspy hamspy = new Hamspy(true);
        Vector spyVector = new MockVector(hamspy);
        hamspy.setStubExpectation(MockVector.MTHD_ELEMENT_AT_$_INT,
                new Object[] { new Integer(100) }).setReturnValue("bar");
        spyVector.addElement("foo");
        assertEquals(0, spyVector.size());
```

```
        assertNull(spyVector.elementAt(0));
        assertEquals("bar", spyVector.elementAt(100));
    }

    public ConcreteSpyTest() {
        super(2, "ConcreteSpyTest");
    }

    public void test(int testNum) throws Throwable {
        switch (testNum) {
        case 0:
            testDelegateToParent();
            break;

        case 1:
            testReturnDefaultValues();
            break;

        default:
            fail("No such test.");
        }
    }

}
```

4 Conclusions

This document showed how the `Hamspy` handler allows test doubles to behave like a spy objects. If we use spy objects, we do not necessarily need to set expectations for the interactions with a test double. Instead, we spy on the interactions and use `asserts` to verify that the OUT interacted as expected after we have exercised the OUT.

When spying on test doubles of concrete classes, the `Hamspy` invocation handler allows the test doubles to return default values or to delegate the call to the mocked class.