Hammock 2.1 User Guide

Test Doubles for Java ME

Carl Meijer

June 2009

1 Mock Objects

1.1 Introduction

In unit testing a programmer writes tests to validate units of production code. A *unit* is, typically, a single method of some class under test (CUT). Normally, though, a class cannot be tested completely in isolation since it needs to collaborate with other classes to do anything useful. Interactions can make testing tricky if a collaborator:

- Is difficult to configure (for example, a data access object, DAO).
- Exhibits behavior that must be tested but is difficult to reproduce (for example, a network error).
- Is slow.
- Is nondeterministic (e.g. a random number generator or date class).
- Doesn't yet have a concrete implementation (e.g. is only specified by an interface or an abstract class).

In any of the above scenarios a *mock object* can usefully stand in for the actual collaborator. The CUT will call methods on the mock object which will return results or throw exceptions as specified in the unit tests. A mock object also allows tests to verify that methods were invoked by the CUT.

Mock objects are one example of the *test double* pattern; other examples of test doubles are test stubs and spy objects.

1.2 Java ME and Mock Objects

Mock objects have found widespread use in Java EE and SE development. Popular frameworks are jMock, EasyMock and Mockito. These frameworks typically rely on, at least, the reflection API and sometimes Java 5 enhancements such as generics. Consequently, mock objects have not been widely used in Java ME which is based on Java 1.2 and doesn't support reflection.

Common characteristics of Java ME applications are that they may:

- Use a phone's connectivity to establish HTTP connections, send and receive SMSs and to communicate with other devices over Bluetooth or infrared.
- Store data persistently either using the `RecordStore` or, possibly, the file system.
- Need to be built for more than one device to work around bugs or to exploit features not present on all devices.

In the above scenarios, mock objects can simplify testing since:

- We can use mocked connections rather than having to hit a network in our tests.
- We can abstract away our persistence layer using mocked persistence classes rather than writing code to set up and tear down record stores.
- We can write tests that mimic the behavior of a particular device.

1.3 Hammock and HammockMaker

Hammock is a test double library that supports the notions of both mock objects and spy objects. Traditionally test double frameworks have supported mock objects but spy objects are gaining favor with the advent of the Mockito framework. This user guide shows how to use mock objects; there is a separate document in this distribution that describes how to use spy objects.

Hammock includes test doubles of many classes from the MIDP packages (including the IO, LCDUI, WMA, PDA API and the location API packages). The Hammock distribution includes a utility, HammockMaker, that allows developers to mock their own interfaces and classes. Unlike Java SE mock object frameworks where the mock objects can be created at run time, Hammock needs source or class files of the objects to be mocked. The Hammock distribution includes a utility, HammockMaker, that can create the source code for a test double given the class file of the class to mock. HammockMaker can be run:

- From the command line.
- As an Ant task.
- As an Eclipse plug-in.

The examples in this guide, run HammockMaker as an Ant task. See the HammockMaker User Guide for details on running the tool from the command line or within Eclipse.

Hammock can be used with any unit test framework for Java ME including J2MEUnit, JMUnit, MoMEUnit and CLDCUnit. The examples in this guide use JMUnit.

1.3.1 Hammock Distributions

There are several Hammock libraries in this distribution (in the `dist` directory):

- The core Hammock libraries for CLDC 1.0 and CLDC 1.1 (`hammock_core_cldc10-2.1.0.jar` and `hammock_core_cldc11-2.1.0.jar`).
- Libraries that include mocks of many MIDP classes for both MIDP 1.0 and MIDP 2.0 (`hammock_midp10_cldc10-2.1.0.jar`, `hammock_midp20_cldc10-2.1.0.jar` and `hammock_midp20_cldc11-2.1.0.jar`).

If possible, you should use the CLDC 1.1 libraries. The CLDC 1.1 build can generate more detailed error messages and catch some testing errors earlier on. The MIDP 2.0 JARs contain mocks of many optional microedition libraries (such as wireless messaging, the PDA API, location API, etc). See the JavaDocs to view all the mocked classes. The examples in this guide use the `hammock_midp20_cldc11-2.1.0.jar` library.

Some people believe that one should mock only your own classes and that you should not mock third-party classes. If you strongly agree with this sentiment, you should use the core libraries. The core libraries can be used with Java SE (but most people would choose to use a Java SE framework like jMock, EasyMock, Mockito, rMock, etc.).

The distribution also includes JARs containing mocks of some of the kSOAP2 classes built for CLDC 1.0 and CLDC 1.1 (in the `ksoap_mocks` directory).

2 The Build System

While it is possible to create MIDlets using only an IDE, most professional development probably uses Ant or Maven to build the applications. Additionally, it is desirable to run unit (and possibly integration) tests before packaging an application. Only if all tests pass is the MIDlet packaged. This user guide uses Ant for building MIDlets. Some familiarity with Ant is probably helpful.

In the next section, we show how to create unit tests using Hammock and JMUnit. The examples can be found in the `examples` directory of the Hammock distribution. The tests can be compiled and run using the `build.xml` Ant script. You may want to use the build script as a template for your own build scripts. In particular the script shows how to add the `<hammockmaker>` and JMUnit `<testlistener>` tasks to your build script.

The build script:

- Compiles the source to be tested.
- Compiles and packages the unit tests into a test MIDlet.
- Runs the tests in the WTK emulator and creates a test report.
- Halts the build if any unit tests failed.
- Packages the application.

2.1 Ant, Antenna, JMUnit and Hammock

The examples use:

- The tasks provided by Antenna for building, obfuscating and packaging MIDlets.
- HammockMaker to create mock objects for the unit tests.
- JMUnit for running automated unit tests.

JMUnit supports two tasks, `<jmunit>` and `<testlistener>`, for running automated tests. This user guide uses `<testlistener>` for running the tests since the `<jmunit>` task requires more third-party libraries. The test reports prepared by `<testlistener>` are parsed by the `<junitreport>` task in the build script to generate an HTML test report.

2.2 A Note on Maven Support

If you're using Maven for your builds (or, perhaps, Apache Ivy) you may want to retrieve the Hammock JARs (i.e. artifacts) from a Maven repository. Hopefully the Hammock libraries will be added to the Apache Central Repository (a JIRA ticket has been logged), in the interim you can retrieve the artifacts from `http://hammingweight.com/modules/hammock/m2repo/` (add the server URL to your Maven `settings.xml`).

The `groupId` for the artifacts is `net.sf.hammockmocks`. The valid `artifactIds` are:

- `hammock_core_cldc10`
- `hammock_core_cldc11`
- `hammock_midp10_cldc10`
- `hammock_midp20_cldc10`
- `hammock_midp20_cldc11`
- `hammockmaker`

The current version of the artifacts is 2.1.0.

A future release of Hammock *may* add a Maven plug-in for HammockMaker.

3 Examples

3.1 Example 1: Setting and Verifying Expectations

In the first example, we want to test a class that uses an `HttpConnection` to retrieve a name corresponding to an ID number from a web server. Listing 1 shows the skeleton of the class that we're going to create. Notice that we have a `protected` method that allows us to inject an `HttpConnection` into the class; we'll use this method to supply a mocked `HttpConnection` for testing.

The source code can be found in the `examples/src` directory of the Hammock distribution.

Listing 1

```
package com.hammingweight.hammockexamples.example1;

import java.io.IOException;

import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;

public class GetName {

    protected String getNameForId(HttpConnection conn, int id)
            throws IOException {
        return null;
    }

    public String getNameForId(int id) throws IOException {
        return this.getNameForId((HttpConnection) Connector
                .open("http://example.com"), id);
    }
}
```

Listing 2 shows the JMUnit test case, `GetNameTest`, that we'll extend to exercise the `GetName` class. The code can be found in the `examples/test` directory.

Listing 2

```
package com.hammingweight.hammockexamples.example1;

import jmunit.framework.cldc10.TestCase;

public class GetNameTest extends TestCase {


    public GetNameTest() {
        super(0, "GetNameTest");
    }

    public void test(int testNum) throws Throwable {
    }
}
```

3.1.1 Creating a MockHttpConnection

Hammock is distributed with mocks of many MIDP classes. In particular, the distribution includes a `MockHttpConnection` class. A mock object must be associated with a method invocation handler. In this user guide, the handler will always be an instance of the `Hammock` class. Listing 3 shows how we create a `Hammock` handler and a `MockHttpConnection` instance. Notice that the handler is set in the `MockHttpConnection` constructor.

Listing 3

```
package com.hammingweight.hammockexamples.example1;

import java.io.IOException;

import javax.microedition.io.HttpConnection;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hammock;
import com.hammingweight.hammock.mocks.microedition.io.MockHttpConnection;

public class GetNameTest extends TestCase {

    // The Object under Test.
    private GetName getName;

    // The mock object handler.
    private Hammock hammock;

    // The mock object(s).
    private MockHttpConnection mockConn;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockConn = new MockHttpConnection(this.hammock);
        this.getName = new GetName();
    }

    public GetNameTest() {
        super(0, "GetNameTest");
    }

    public void test(int testNum) throws Throwable {
    }
}
```

3.1.2 Setting an Expectation

Testing using mock objects involves three steps:

- Setting expectations that certain methods will be invoked on the mock objects.
- Exercising the object under test (OUT).
- Verifying that the OUT invoked its (mocked) collaborators as expected.

Initially, we'll simply write a test that verifies that the `GetNames` class closes the `HttpConnection` when we invoke the `getNameForId()` method. The `testGetNameForId()` method in Listing 4 shows how we set an expectation that the `close()` method will be invoked on the `MockHttpConnection`. Hammock associates an identifier of the form `MTHD_XXX` with every method exposed by the mock object where `XXX` is the name of the method; e.g. `MTHD_CLOSE` corresponds to the `close()` method.

After setting the expectation that the `close()` method will be invoked, we exercise the class by trying to get the name corresponding to identifier 1234. Notice that we inject our mock object into the method that we're testing. Finally we call the `verify()` method on the `Hammock` handler to verify that our mock object was invoked as expected.

Listing 4

```
public class GetNameTest extends TestCase {

    // The Object under Test.
    private GetName getName;

    // The mock object handler.
    private Hammock hammock;

    // The mock object(s).
    private MockHttpConnection mockConn;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockConn = new MockHttpConnection(this.hammock);
        this.getName = new GetName();
    }

    public void testGetNameForId() throws IOException {
        // Expectations.
        this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

        // Exercise.
        this.getName.getNameForId(this.mockConn, 1234);

        // Verify.
        this.hammock.verify();
    }

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

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

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

If we run the test, an exception is thrown:

```
com.hammingweight.hammock.HammockException: A method was invoked less often
than expected.
        Class:  MockHttpConnection
        Method: MTHD_CLOSE
```

To get the test to pass, we need to close the `HttpConnection` (see listing 5).

Listing 5

```
public class GetName {

    protected String getNameForId(HttpConnection conn, int id)
            throws IOException {
        conn.close();
        return null;
    }

    public String getNameForId(int id) throws IOException {
        return this.getNameForId((HttpConnection) Connector
                .open("http://example.com"), id);
    }
}
```

3.1.3 Passing Arguments to Mock Object Method Calls

The `close()` method is not very exciting since it takes no arguments nor does it return a value. We'll now consider setting an expectation that `setRequestMethod(String method)` will be invoked. Listing 6 shows a test that sets an expectation that the `setRequestMethod()` method will be invoked.

Listing 5

```
public void testGetNameForId() throws IOException {
    // Expectations.
    this.hammock
            .setExpectation(MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING);
    this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

    // Exercise.
    this.getName.getNameForId(this.mockConn, 1234);

    // Verify.
    this.hammock.verify();
}
```

Notice the identifier `MTHD_SET_REQUEST_METHOD_$_STRING`; the '$' indicates that the method takes arguments and the 'STRING' indicates that the first (and only) argument for the method is a `String`.

We've set an expectation that the `setRequestMethod()` method will be invoked but the expectation isn't satisfactory; we should specify whether the request method is GET or POST. Fortunately, the `setExpectation()` method of the Hammock class is overloaded so that we can supply expected arguments. Listing 6 shows the improved test in which we set the expectation that the request method will be POST. Expected arguments are passed as an array of `Objects`.

Listing 6

```
public void testGetNameForId() throws IOException {
    // Expectations.
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING,
            new Object[] { HttpConnection.POST });
    this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

    // Exercise.
    this.getName.getNameForId(this.mockConn, 1234);

    // Verify.
    this.hammock.verify();
}
```

Listing 7 shows the `getNameForId()` method that allows the test in listing 6 to pass.

Listing 7

```
protected String getNameForId(HttpConnection conn, int id)
        throws IOException {
    conn.setRequestMethod(HttpConnection.POST);
    conn.close();
    return null;
}
```

3.1.4 Returning a Value from a Mock Object

The `close()` and `setRequestMethod()` methods do not return values. Frequently we need to specify the value that a mock object must return when a method is invoked; after setting an expectation we can chain a `setReturnValue()` method invocation. Listing 8 sets the expectation that the OUT will open a data output stream and we specify the `DataOutputStream`, `dos`, that must be returned. The code in listing 8 also asserts that the identifier is written to the output stream by checking the state of the output stream.

Listing 8

```
public void testGetNameForId() throws IOException {
    // Expectations.
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING,
            new Object[] { HttpConnection.POST });
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_OPEN_DATA_OUTPUT_STREAM)
            .setReturnValue(dos);
    this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

    // Exercise.
    this.getName.getNameForId(this.mockConn, 1234);
    DataInputStream dis = new DataInputStream(new ByteArrayInputStream(baos
            .toByteArray()));
    assertEquals(1234, dis.readInt());

    // Verify.
    this.hammock.verify();
}
```

Finally, for this first test, we need to read from a `DataInputStream` and return the result. The code is shown in listing 9.

Listing 9

```
public void testGetNameForId() throws IOException {
    // Expectations.
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING,
            new Object[] { HttpConnection.POST });
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_OPEN_DATA_OUTPUT_STREAM)
            .setReturnValue(dos);
    ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
    new DataOutputStream(baos2).writeUTF("Carl␣Meijer");
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_OPEN_DATA_INPUT_STREAM).setReturnValue(
            new DataInputStream(new ByteArrayInputStream(baos2
                    .toByteArray())));
    this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

    // Exercise.
    assertEquals("Carl␣Meijer", this.getName.getNameForId(this.mockConn, 1234));
    DataInputStream dis = new DataInputStream(new ByteArrayInputStream(baos
            .toByteArray()));
    assertEquals(1234, dis.readInt());

    // Verify.
    this.hammock.verify();
}
```

Listing 10 shows the code that we wrote to get this test to pass.

Listing 10

```
protected String getNameForId(HttpConnection conn, int id)
        throws IOException {
    conn.setRequestMethod(HttpConnection.POST);
    conn.openDataOutputStream().writeInt(id);
    String result = conn.openDataInputStream().readUTF();
    conn.close();
    return result;
}
```

Note that there is a significant problem with the code in listing 10: the `DataInputStream` and `DataOutputStreams` are never closed. If we had used mocked streams we could have set expectations that `close()` method is invoked on the streams.

3.1.5 Using MockDataInputStream and MockDataOuputStream

If you look at the classes or JavaDocs distributed with Hammock, you'll notice that the distribution includes mocks of `DataInputStream` and `DataOutputStream`. We could have returned a `MockDataOutputStream` when opening a `DataOutputStream` on the `MockHttpConnection`. However this would have posed a problem since the `writeInt()` method of `DataOutputStream` is `final` and, consequently, the method can't be mocked. If you look at the `MockDataOutputStream` class, you will see that there is no method identifier called `MTHD_WRITE_INT_$_INT`.

The declaration of classes or methods as final can make using mock objects difficult; the fourth example of this section shows how to work around the problems of testing classes with `final` methods.

3.1.6 Throwing an IOException

One of the advantages of using mock objects is that we can get them to exhibit behavior that needs to be tested but that can be difficult to produce in a real collaborator. For example, we should test that even if an `IOException` is thrown that the `HTTPConnection` is closed. Listing 11 sets an expectation that the `setRequestMethod()` will be called and the behavior that an `IOException` will be thrown when this happens. We can specify that a method must throw an exception via the `setThrowable()` method.

Listing 11

```
public void testCloseAfterIoException() {
    // Expectations.
    this.hammock.setExpectation(
            MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING)
            .setThrowable(new IOException());
    this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);
    this.hammock.setStrictOrdering();

    // Exercise.
    try {
        this.getName.getNameForId(this.mockConn, 1234);
        fail("IOException␣showld␣have␣been␣thrown");
    } catch (IOException ioe) {
        // Expected.
    }

    // Verify.
    this.hammock.verify();
}
```

The code in listing 11 illustrates the use of the `setStrictOrdering()` method; by default Hammock doesn't care in what order methods are invoked on mock objects. If we specify strict ordering the methods must be invoked in the order specified in the expectations. Mock objects that require that methods be invoked in some order are known as *strict mocks.*

If we run the above test against the code in listing 10, our unit test fails:

```
com.hammingweight.hammock.HammockException: A method was invoked less often
than expected.
        Class:  MockHttpConnection
        Method: MTHD_CLOSE
```

We need to refactor the code of listing 10 so that both tests pass; the result is in listing 12.

Listing 12

```java
protected String getNameForId(HttpConnection conn, int id)
        throws IOException {
    try {
        conn.setRequestMethod(HttpConnection.POST);
        conn.openDataOutputStream().writeInt(id);
        return conn.openDataInputStream().readUTF();
    } finally {
        conn.close();
    }
}
```

3.2 Example 2: Creating Mock Objects

The previous example used the `MockHttpConnection` class that is supplied with Hammock. Often, though, you will want to mock your own classes. For example, if your application has a layered architecture, you may want to mock the layer that the class under test interacts with. Also if a class needs to persist data or retrieve persisted data it can be better to mock your persistence code or your DAO since the `setUp()` and `tearDown()` code for persistence can be substantial, slow and brittle (any change to the DAO may break the `setUp()` code).

In the example in this section we assume that we have a `Persistence` class that can save `Hashtable`s where the keys and values are `String`s. Classes that need to be persisted will have an associated DAO class that can convert an instance of the class to a `Hashtable` representation.

3.2.1 The Persistence Class

Listing 13 shows the skeleton of our `Persistence` class. We might, conceivably, have more than one implementation of this class; for example one implementation that uses the `RecordStore` and another that uses the the file system if the device supports JSR 75 (PDA API).

Listing 13

```java
package com.hammingweight.hammockexamples.example2;

import java.util.Hashtable;

public class Persistence {

    /**
     *
     * @param clazz
     *             The original class of the persisted object.
     * @param id
     *             An identifier (or primary key) for the object.
     * @param data
     *             A representation of the object to be serialized.
     */
    public void persist(Class clazz, int id, Hashtable data) {

    }
}
```

3.2.2 The Person and PersonDao Classes

In this example we'll write code to persist the details of a person. For our purposes, a person is simply somebody who has a first and last name as shown in listing 14.

Listing 14

```
package com.hammingweight.hammockexamples.example2;

public class Person {

    private String firstName;

    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

}
```

The `PersonDao` class that interacts with the persistence layer is shown in listing 15. We need to implement the `persist()` method. Note the dependency on a `Random` instance; the `PersonDao` will use the `Random` object for generating an identifier when persisting the `Person` instance.

Listing 15

```
package com.hammingweight.hammockexamples.example2;

import java.util.Random;

public class PersonDao {

    private Persistence persistence;

    private Random random;

    public void setPersistence(Persistence persistence) {
        this.persistence = persistence;
    }

    public void setRandom(Random random) {
        this.random = random;
    }

    public void persist(Person person) {
    }
}
```

3.2.3 Mocking the Persistence Class

Before we can test the `PersonDao` class, we will need to mock the `Persistence` class. The `build.xml` script includes the following lines that create a mock of the `Persistence` class using the `<hammockmaker>` task:

```
<!-- Create mock objects for our tests. -->
<hammockmaker usecldc11="true" dir="${test.dir}"
        package="com.hammingweight.hammockexamples.mocks" classpath="${src.build.dir}">
        <mock class="com.hammingweight.hammockexamples.example2.Persistence" />
</hammockmaker>
```

The mock object source code is created in the `com.hammingweight.hammockexamples.mocks` package as specified in the `package` attribute of the `hammockmaker` element. The name of the mocked class is `MockPersistence`; HammockMaker prepends the word “Mock” to the mocked class name. The `useCldc11` attribute of the `<hammockmaker>` task determines whether the source code should be compatible with CLDC 1.1 or with CLDC 1.0; code generated for CLDC 1.1 is larger but produces more useful error messages when a mock object is unexpectedly invoked. The `dir` attribute of the task specifies the directory where the source code for the mock object will be written.

Notice that the mock object that we created is a mock of a concrete class; HammockMaker can mock interfaces and non-final classes. You can only set expectations for methods that are non-final when mocking a class; where possible it is better to mock interfaces rather than classes. However in Java ME it is common to avoid interfaces since every interface adds to the size of the deployed application.

3.2.4 Testing the PersonDao Class

Listing 16 tests the `persist()` method of the `PersonDao` class. There are several things to note in the test:

- The test uses two mock objects but needs only one instance of a `Hammock`.
- We are setting an expectation that a random ID will be generated for the persisted object.
- We are setting a delay of 500 milliseconds before the `MockPersistence` class responds to a method invocation.
- We are specifying that the the `Random` class will be invoked at least once and at most three times.
- When returning an `int` value from a method, we need to wrap it in an `Integer`.

Setting the delay (using `setDelay()`) actually serves no purpose here but can be useful when testing multithreaded applications where you want to ensure that one action completes before another. Similarly, specifying that the `Random` class may be invoked more than once also serves no purpose but simply illustrates some functionality that Hammock provides.

Listing 16

```
package com.hammingweight.hammockexamples.example2;

import java.util.Hashtable;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hammock;
import com.hammingweight.hammock.mocks.util.MockRandom;
import com.hammingweight.hammockexamples.example2.Person;
import com.hammingweight.hammockexamples.example2.PersonDao;
import com.hammingweight.hammockexamples.mocks.MockPersistence;

public class PersonDaoTest extends TestCase {

    // The OUT.
    private PersonDao personDao;

    // The handler and mock(s)
    private Hammock hammock;
    private MockPersistence mockPersistence;
    private MockRandom mockRandom;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockPersistence = new MockPersistence(this.hammock);
        this.mockRandom = new MockRandom(this.hammock);
        this.personDao = new PersonDao();
        this.personDao.setPersistence(this.mockPersistence);
        this.personDao.setRandom(this.mockRandom);
    }

    public void testPersist() {
        // Expectations.
        // The Random object should be asked to generate a random ID.
        // Our random value is -42.
        this.hammock.setExpectation(MockRandom.MTHD_NEXT_INT).setReturnValue(
            new Integer(-42)).setInvocationCount(1, 3);
        // A Hashtable with the first and last names should be persisted.
        Hashtable h = new Hashtable();
        h.put("fname", "Carl");
        h.put("lname", "Meijer");
        this.hammock.setExpectation(
                MockPersistence.MTHD_PERSIST_$_CLASS_INT_HASHTABLE,
                new Object[] { new Person("", "").getClass(), new Integer(-42),
                    h }).setDelay(500).setArgumentMatcher(2,
                new HashtableMatcher());

        // Exercise.
        this.personDao.persist(new Person("Carl", "Meijer"));

        // Verify.
        this.hammock.verify();
    }

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

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

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

After writing the test, we write the code to pass the test. Listing 17 shows our implementation of the `persist()` method:

Listing 17

```
public void persist(Person person) {
    Hashtable h = new Hashtable();
    h.put("fname", person.getFirstName());
    h.put("lname", person.getLastName());
    this.persistence.persist(person.getClass(), this.random.nextInt(), h);
}
```

Unfortunately, when we run our test it fails:

```
com.hammingweight.hammock.HammockException: A method was unexpectedly invoked.
Hint: Check the method, the expected number of invocations,
the method arguments and the mock object.
        Class:  MockPersistence
        Method: MTHD_PERSIST_$_CLASS_INT_HASHTABLE
```

The reason for the failure is that HammockMaker compares the expected and actual `Hashtable` arguments using the `equals()` method. Since the actual and expected `Hashtable`s reference different instances they are not regarded as equal. The simplest way to 'fix' this problem is simply to ignore the third argument in the expectation as follows:

```
this.hammock.setExpectation(
                MockPersistence.MTHD_PERSIST_$_CLASS_INT_HASHTABLE,
                new Object[] { Person.class, new Integer(-42), h }).setDelay(
                500).ignoreArgument(2);
```

(Note that arguments are indexed from zero, so 2 corresponds to the method's third argument.)

A better way to fix the problem is to check whether the `Hashtable`s contain the same data. This can be done by implementing an `IArgumentMatcher`. Listing 18 shows a class that tests whether two `Hashtable`s are equal for our purposes.

Listing 18

```java
package com.hammingweight.hammockexamples.example2;

import java.util.Enumeration;
import java.util.Hashtable;

import com.hammingweight.hammock.IArgumentMatcher;

public class HashtableMatcher implements IArgumentMatcher {

    public boolean areArgumentsEqual(Object expected, Object actual) {
        if (!(actual instanceof Hashtable)) {
            return false;
        }

        Hashtable e = (Hashtable) expected;
        Hashtable a = (Hashtable) actual;

        if (e.size() != a.size()) {
            return false;
        }

        Enumeration en = e.keys();
        while (en.hasMoreElements()) {
            Object key = en.nextElement();
            if (!e.get(key).equals(a.get(key))) {
                return false;
            }
        }
        return true;
    }

}
```

We can use our `HashtableMatcher` class to check that our `MockPersistence` class is invoked as expected with the following code change:

```java
this.hammock.setExpectation(
                MockPersistence.MTHD_PERSIST_$_CLASS_INT_HASHTABLE,
                new Object[] { Person.class, new Integer(-42), h }).setDelay(
                500).setArgumentMatcher(2, new HashtableMatcher());
```

3.3 Example 3: Modifying Arguments in Mocked Methods

A mocked method can return a value specified using the `setReturnValue()` method. Sometimes though a mutable `Object` will be passed to a method and the method must change the state of the `Object`. The next example shows how to achieve this in a mocked object.

A *stream cipher* is an encryption scheme where the bytes of a message are XORed with a pseudorandom sequence to produce ciphertext. Decryption works exactly the same way; the bytes of the ciphertext are XORed with the pseudorandom byte stream to produce the original plaintext. We'll test a `StreamCipher` class that collaborates with a `PseudoRandomByteStream`.

Listing 19 shows the interface for the `PseudoRandomByteStream`. The important point to note is that we pass a `byte` array to the `generateBytes()` method which will populate the array with pseudorandom bytes. If we use a mocked `PseudoRandomByteStream` we need some way of populating the array.

Listing 19

```
package com.hammingweight.hammockexamples.example3;

public interface PseudoRandomByteStream {

    public void generateBytes(byte[] stream);
}
```

Listing 20 shows the implementation the `StreamCipher` class. Notice that there are two `PseudoRandomByteStream` collaborators: one for encryption and one for decryption.

Listing 20

```
package com.hammingweight.hammockexamples.example3;

public class StreamCipher {

    private PseudoRandomByteStream encStream, decStream;

    public StreamCipher(PseudoRandomByteStream encStream,
            PseudoRandomByteStream decStream) {
        this.encStream = encStream;
        this.decStream = decStream;
    }

    public byte[] encrypt(byte[] plaintext) {
        byte[] rnd = new byte[plaintext.length];
        this.encStream.generateBytes(rnd);
        for (int i = 0; i < rnd.length; i++) {
            rnd[i] ^= plaintext[i];
        }
        return rnd;
    }

    public byte[] decrypt(byte[] ciphertext) {
        byte[] rnd = new byte[ciphertext.length];
        this.decStream.generateBytes(rnd);
        for (int i = 0; i < rnd.length; i++) {
            rnd[i] ^= ciphertext[i];
        }
        return rnd;
    }
}
```

Listing 21 shows the `TestCase` that exercises the `StreamCipher`'s `encrypt()` and `decrypt()` methods.

Listing 21

```
package com.hammingweight.hammockexamples.example3;

import java.io.UnsupportedEncodingException;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hammock;
import com.hammingweight.hammock.IArgumentMatcher;
import com.hammingweight.hammock.PopulateArrayMatcher;
import com.hammingweight.hammockexamples.mocks.MockPseudoRandomByteStream;

public class StreamCipherTest extends TestCase {

    // The OUT.
    private StreamCipher streamCipher;

    // The handler and mock(s).
    private Hammock hammock;
    private MockPseudoRandomByteStream mockStream1, mockStream2;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockStream1 = new MockPseudoRandomByteStream(this.hammock);
        this.mockStream2 = new MockPseudoRandomByteStream(this.hammock);
        this.streamCipher = new StreamCipher(this.mockStream1, this.mockStream2);
    }

    public void testEncrypt() throws UnsupportedEncodingException {
        // Expectations.
        byte[] b = { 1, 2, 3, 4, 5 };
        this.hammock.setExpectation(
                MockPseudoRandomByteStream.MTHD_GENERATE_BYTES_$_ARRAY_BYTE,
                this.mockStream1, new Object[] { b }).setArgumentMatcher(0,
                new IArgumentMatcher() {

                    public boolean areArgumentsEqual(Object expected,
                            Object actual) {
                        byte[] e = (byte[]) expected;
                        byte[] a = (byte[]) actual;
                        System.arraycopy(e, 0, a, 0, e.length);
                        return true;
                    }

                });

        // Exercise.
        byte[] msg = "hello".getBytes("UTF8");
        byte[] ciphertext = this.streamCipher.encrypt(msg);
        assertEquals(5, ciphertext.length);
        for (int i = 0; i < 5; i++) {
            assertEquals(msg[i] ^ (i + 1), ciphertext[i]);
        }

        // Verify.
        this.hammock.verify();
    }

    public void testDecrypt() {
        // Expectations.
        byte[] b = { 0x55, 0x55, 0x55, 0x55 };
        this.hammock.setExpectation(
                MockPseudoRandomByteStream.MTHD_GENERATE_BYTES_$_ARRAY_BYTE,
                new Object[] { b }).setArgumentMatcher(0,
                new PopulateArrayMatcher(0, 4, 0));

        // Exercise.
        byte[] msg = { 1, 2, 4, 8 };
        byte[] plaintext = this.streamCipher.decrypt(msg);
        assertEquals(4, plaintext.length);
        for (int i = 0; i < 4; i++) {
            assertEquals(msg[i] ^ 0x55, plaintext[i]);
        }

        // Verify.
        this.hammock.verify();
    }

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

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

        case 1:
            testDecrypt();
            break;

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

}
```

In the `testEncrypt()` method we use an argument matcher to change the contents of the actual argument passed using the expected argument. An interesting point illustrated in this test is that we have two mocked instances of `PseudoRandomByteStream`; notice that the `setExpectation()` method has an overloaded form in which we specify that the method must be invoked on `mockStream1`.

While we can implement our own `IArgumentMatcher` to change any mutable object, populating arrays is such a common requirement that Hammock provides a convenience matcher, `PopulateArrayMatcher`, specifically for this task. The `testDecrypt()` method illustrates the use of the convenience class. The constructor for `PopulateArrayMatcher`, takes three `int` arguments: the first argument is the offset from which to start copying from the array passed in the expectation, the second argument is the number of array elements to copy and the third argument is the offset in the destination (actual argument) array to start copying to.

3.4 Example 4: Refactoring for Testability

In the first example, we saw that it was impractical to test using a `MockDataOutputStream` since the methods that would be invoked on the mock object were declared final. This is a well known problem when testing with mock objects. In section 7.4

of "JUnit in Action" (Manning, 2004) Vincent Massol presents a similar example of testing where a class collaborates with an instance of the final Java SE `URL` class. We illustrate Massol's solution to the problem using the refactoring technique known as *Class factory refactoring.*

It can be seen as a disadvantage of mock objects that we have to refactor our code to make it testable. However we might also conclude that the refactoring improves the quality of our code.

3.4.1 Refactoring the GetName Example

Listing 22 introduces a factory class for opening input and output streams. Important features of the interface are:

- The factory returns instances of `DataInput` and `DataOutput` rather than `DataInputStream` and `DataOutputStream` allowing us to program to an interface rather than an implementation.
- There are no references to `HttpConnection` meaning that our code that needs to read and write to the input and output streams does not need to include any code for managing the `HttpConnection`. This change should improve the code's readability and simplify switching from HTTP to HTTPS, Bluetooth or even SMS by replacing the factory class implementation.
- A bug in the code of listing 10 was that we did not close the input and output streams; adding this boilerplate code would have made the code less readable. Our interface exposes a method `closeConnections()` that takes care of closing any streams returned by the factory.

Listing 22

```
package com.hammingweight.hammockexamples.example4;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public interface ConnectionFactory {

    public void openConnections(String relativePath) throws IOException;

    public DataInput getInputConnection() throws IOException;

    public DataOutput getOutputConnection() throws IOException;

    public void closeConnections();
}
```

Listing 23 shows our `TestCase` for testing our class that collaborates with a `ConnectionFactory` to get the name corresponding to an ID.

Listing 23

```
package com.hammingweight.hammockexamples.example4;

import java.io.IOException;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hammock;
import com.hammingweight.hammock.mocks.io.MockDataInput;
import com.hammingweight.hammock.mocks.io.MockDataOutput;
import com.hammingweight.hammockexamples.mocks.MockConnectionFactory;

public class GetName2Test extends TestCase {
    // The OUT.
    private GetName2 getName;

    // The handler and mock(s)
    private Hammock hammock;
    private MockConnectionFactory mockConnFactory;
    private MockDataOutput mockDataOutput;
    private MockDataInput mockDataInput;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockConnFactory = new MockConnectionFactory(this.hammock);
        this.mockDataOutput = new MockDataOutput(this.hammock);
        this.mockDataInput = new MockDataInput(this.hammock);
        this.getName = new GetName2(this.mockConnFactory);
    }

    public void testGetNameForId() throws IOException {
        // Expectations.
        this.hammock.setExpectation(
                MockConnectionFactory.MTHD_OPEN_CONNECTIONS_$_STRING,
                new Object[] { "getname.inm" });
        this.hammock.setExpectation(
                MockConnectionFactory.MTHD_GET_OUTPUT_CONNECTION)
                .setReturnValue(this.mockDataOutput);
        this.hammock.setExpectation(MockDataOutput.MTHD_WRITE_INT_$_INT,
                new Object[] { new Integer(1234) });
        this.hammock.setExpectation(
                MockConnectionFactory.MTHD_GET_INPUT_CONNECTION)
                .setReturnValue(this.mockDataInput);
        this.hammock.setExpectation(MockDataInput.MTHD_READ_UTF)
                .setReturnValue("Carl␣Meijer");
        this.hammock.setExpectation(MockConnectionFactory.MTHD_CLOSE_CONNECTIONS);

        // Exercise.
        assertEquals("Carl␣Meijer", this.getName.getNameForId(1234));

        // Verify.
        this.hammock.verify();
    }

    public void testClose() {
        // Expectations.
        this.hammock.setExpectation(
                MockConnectionFactory.MTHD_OPEN_CONNECTIONS_$_STRING)
                .setThrowable(new IOException());
        this.hammock.setExpectation(MockConnectionFactory.MTHD_CLOSE_CONNECTIONS);

        // Exercise.
        try {
            this.getName.getNameForId(1234);
            fail("AN␣IOException␣should␣have␣been␣thrown.");
        } catch (IOException e) {
            // Expected.
        }

        // Verify.
        this.hammock.verify();
    }

    public GetName2Test() {
        super(2, "GetName2Test");
    }

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

        case 1:
            testClose();
            break;

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

Listing 24 shows our implementation of the code to pass the tests of listing 23.

Listing 24

```
package com.hammingweight.hammockexamples.example4;

import java.io.IOException;

public class GetName2 {

    private ConnectionFactory connFactory;

    public GetName2(ConnectionFactory connFactory) {
        this.connFactory = connFactory;
    }

    public String getNameForId(int id) throws IOException {
        try {
            this.connFactory.openConnections("getname.inm");
            this.connFactory.getOutputConnection().writeInt(id);
            return this.connFactory.getInputConnection().readUTF();
        } finally {
            this.connFactory.closeConnections();
        }
    }
}
```

3.4.2 An HttpConnectionFactory

Listing 25 shows an implementation of the `ConnectionFactory` interface that uses HTTP.

Listing 25

```
package com.hammingweight.hammockexamples.example4;

import java.io.DataInput;
import java.io.DataInputStream;
import java.io.DataOutput;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;

public class HttpConnectionFactory implements ConnectionFactory {

    private String host;

    private HttpConnection httpConn;

    private DataOutputStream dos;

    private DataInputStream dis;

    public HttpConnectionFactory(String host) {
        this.host = host;
    }

    public void closeConnections() {
        if (this.dos != null) {
            try {
                this.dos.close();
            } catch (IOException e) {
            }
        }

        if (this.dis != null) {
            try {
                this.dis.close();
            } catch (IOException e) {
            }
        }

        if (this.httpConn != null) {
            try {
                this.httpConn.close();
            } catch (IOException e) {
            }
        }

        this.dos = null;
        this.dis = null;
        this.httpConn = null;
    }

    public DataInput getInputConnection() throws IOException {
        this.dis = this.httpConn.openDataInputStream();
        return this.dis;
    }

    public DataOutput getOutputConnection() throws IOException {
        this.dos = this.httpConn.openDataOutputStream();
        return this.dos;
    }

    protected void openConnections(HttpConnection httpConn) throws IOException {
        this.httpConn = httpConn;
        this.httpConn.setRequestMethod(HttpConnection.GET);
    }

    public void openConnections(String relativePath) throws IOException {
        openConnections((HttpConnection) Connector.open("http://" + this.host
                + "/" + relativePath));
    }
}
```

Naturally, the code of listing 25 was tested. Listing 26 shows the test that we wrote (note that we used a `MockHttpConnection` for testing our HTTP factory class).

Listing 26

```
package com.hammingweight.hammockexamples.example4;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

import javax.microedition.io.HttpConnection;

import jmunit.framework.cldc10.TestCase;

import com.hammingweight.hammock.Hammock;
import com.hammingweight.hammock.mocks.io.MockDataInputStream;
import com.hammingweight.hammock.mocks.io.MockDataOutputStream;
import com.hammingweight.hammock.mocks.microedition.io.MockHttpConnection;

public class HttpConnectionFactoryTest extends TestCase {

    // The OUT.
    private HttpConnectionFactory httpConnFactory;

    // The handler and mock(s).
    private Hammock hammock;
    private MockHttpConnection mockHttpConn;
    private MockDataOutputStream mockDataOutputStream;
    private MockDataInputStream mockDataInputStream;

    public void setUp() {
        this.hammock = new Hammock();
        this.mockHttpConn = new MockHttpConnection(this.hammock);
        this.mockDataOutputStream = new MockDataOutputStream(
                new ByteArrayOutputStream(), this.hammock);
        this.mockDataInputStream = new MockDataInputStream(
                new ByteArrayInputStream("".getBytes()), this.hammock);
        this.httpConnFactory = new HttpConnectionFactory("example.com");
    }

    public void testOpenAndClose() throws IOException {
        // Expectations.
        this.hammock.setExpectation(
                MockHttpConnection.MTHD_SET_REQUEST_METHOD_$_STRING,
                new Object[] { HttpConnection.GET });
        this.hammock.setExpectation(
                MockHttpConnection.MTHD_OPEN_DATA_OUTPUT_STREAM)
                .setReturnValue(this.mockDataOutputStream);
        this.hammock.setExpectation(
                MockHttpConnection.MTHD_OPEN_DATA_INPUT_STREAM).setReturnValue(
                this.mockDataInputStream);
        this.hammock.setExpectation(MockDataOutputStream.MTHD_CLOSE);
        this.hammock.setExpectation(MockDataInputStream.MTHD_CLOSE);
        this.hammock.setExpectation(MockHttpConnection.MTHD_CLOSE);

        // Exercise.
        this.httpConnFactory.openConnections(this.mockHttpConn);
        this.httpConnFactory.getInputConnection();
        this.httpConnFactory.getOutputConnection();
        this.httpConnFactory.closeConnections();
        // Close again, to check that the HttpConnection, DataOutputStream and
        // DataInputStream are closed only once.
        this.httpConnFactory.closeConnections();

        // Verify.
        this.hammock.verify();
    }

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

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

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

}
```

4 Effective Testing with Mocks

Here are some tips for effective testing with mock objects:

- Not all tests need mock objects; if most of your tests don't use mock objects that's probably OK.
- If you need to to set lots of expectations, your test is going to be brittle and refactoring will be difficult. If a single test uses more than two mock objects and sets more than five expectations, the method that you're testing may be doing too much. Try to decompose the method into smaller submethods that do less and test those separately. The submethods could have `package` or `protected` access so that your unit tests can access the methods but collaborators can only invoke the methods indirectly through the original, `public` method.
- Most of your mocks should be of your own classes; it's unlikely that using a `MockVector` offers advantages over a real `Vector` except in certain, rare situations (e.g. a method should retrieve the 100-th element from a `Vector` and you don't want to populate a `Vector` with 99 irrelevant items when you could simply set an expectation that the 100-th element will be read).
- Using mocks of networking `Objects` (e.g. a `MockHttpConnection` or a `MockMessage`) can be advantageous since unit tests that establish real connections are fragile and slow. However, even when testing connectivity, consider adding another level of abstraction so that your classes don't need to be aware of the particular network protocol (e.g. provide a factory class that provides input and output streams so that your class under test doesn't need boilerplate code for setting HTTP connection properties and methods).
- Persistence is a common requirement of Java ME applications. Create DAOs that marshall and unmarshall your domain objects for storage in the `RecordStore`. Use mocked DAOs in your tests since this simplifies your tests' `setUp()` and `tearDown()` code. The use of DAOs results in more loosely coupled-code than if your domain objects are littered with `RecordStore` logic. Note that it is not possible to mock the `RecordStore` class but in practice you don't need to; your DAOs should be tested with real record stores in any case.

5 Conclusions

This user guide introduced the Hammock mock object library. Specifically it covered:

- The `Hammock` class for handling invocations of mock objects.

- The use of some of the pre-built MIDP mock objects (e.g. `MockHttpConnection`, `MockDataInput` and `MockRandom`).
- How to build mock objects using the `<hammockmaker>` task.
- How to set expectations that a method will be invoked (using the `setExpectation()` method).
- How to return values (via `setReturnValue()`) and throw exceptions (via `setThrowable()`).
- How to verify that a mock object was invoked as expected (using the `verify()` method).
- How to use argument matchers (by creating instances of `IArgumentMatcher`) and to ignore method arguments (using `ignoreArgument()`).
- How to populate arrays using the `PopulateArrayMatcher` class and to modify other mutable Object arguments using `IArgumentMatcher`s.
- How to control the behavior of a mock object using the `setDelay()`, `setStrictOrdering()` and `setNumberInvocations()` methods.