Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 21 — February 28, 2010

Patrick Lam *version 1*

Unit Testing

We've seen implementation details of how to use JUnit to perform unit testing, and you should be able to generalize to other unit testing frameworks on your own.

Today's lecture. We will discuss principles behind unit testing and how and when to best deploy unit testing. As you may have noticed with the SweetHome3D "unit tests", the use of JUnit alone doesn't ensure that you are actually writing unit tests, and we'll provide definitions of what people usually mean when they say unit tests.

1. Test small parts—units—of a software system (method, class, set of classes) independently.
2. Specify the desired behaviour of the unit using tests.

The following concepts are related to unit testing: test-driven development[1] (TDD), which is currently the most-advocated way to write unit tests; mock objects and dependency injection; and automation.

Some basic properties of unit tests:

- focus on the unit being tested; for instance, don't depend on a database, network, or other external resources;
- easy to run by anyone (e.g. by setting them up as JUnit tests): they must incur no setup costs to run nor require user input; they can therefore serve as regression tests;
- easy to write (a few minutes per test) and focus on one aspect of behaviour; they should tell you something about the unit.

Unit tests come with a lot of baggage and people telling you how to do things, e.g.:

- specify and document the requirements of the unit;
- test behaviour, not state of return values (more on this later), and use mock objects to verify behaviour;
- some say that unit tests ought to be created during development (TDD) and written first, even before the code to be tested exists.

We'll work through an example, which includes mock objects and dependency injection. After that, we'll talk a bit about mock objects and dependency injection in the abstract.

[1] `http://radio-weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html`

Case Study

We will examine some unit tests for Apache CXF[2], an "open-source services framework." CXF has 725 files named "*Test.java" which account for 125 kLOC, and 3778 Java files in all, with 500 kLOC. In particular, we will investigate the `OsgiDestination` class, which belongs to the package `org.apache.cxf.transport.http_osgi`, along with its associated test `OsgiDestinationTest`, also in the same package.

This test uses the EasyMock library to provide its mock objects.

Structure of test case.

Examining the test class, we find 6 private fields: a constant, a mock object `control` field, and four fields for test objects, three of which are mock objects (`bus`, `registry` and `observer`) and one of which is a real domain object (`endpoint`). Pure mock objects rely on interface/implementation separation and implement the declared interface of the real object, e.g. `BusMessageObserver`, using reflection.

We also find a setup method, `setUp()`, a tear-down method, `tearDown()`, and three test methods, `testCtor()`, `testShutdown()`, and `testMessage()`. The `testMessage()` method uses a private helper method, `setUpMessage()`.

Test setup. Recall that JUnit runs methods annotated `@Before` before running any test cases. In this case, `setUp()` is so annotated. It creates the mock objects to be used by the `OsgiDestination` under test, and one real object. (Note that many mock object libraries are available; this test uses EasyMock[3]. We'll talk about mocks more later.)

```
control = EasyMock.createNiceControl();
bus = control.createMock(Bus.class);
...
endpoint = new EndpointInfo(); endpoint.setAddress(ADDRESS);
// why isn't endpoint a mock object?
```

Here is a quick HOWTO for testing using mocks:

1. Create the mock object;
2. Record expected interactions for the mock object;
3. Switch EasyMock to replay mode;
4. Test the code, ideally causing the specified interactions;
5. Ask EasyMock to verify that the specified interactions took place.

[2]`http://cxf.apache.org`
[3]`http://easymock.org`

Test teardown. The teardown method nulls the mock fields.

Constructor test. Constructors are expected to set the state of the object being constructed; they usually don't have much behaviour. Hence:

```
@Test
public void testCtor() throws Exception {
    OsgiDestination destination = new OsgiDestination(bus, endpoint, registry, ''snafu'');

    assertNull(destination.getMessageObserver());
    assertNotNull(destination.getAddress());
    assertNotNull(destination.getAddress().getAddress());
    assertEquals(ADDRESS, destination.getAddress().getAddress().getValue());
}
```

Why is there no call to `control.verify()` or `replay()`?

doMessage test. The next test demonstrates the use of mock objects. The method under test, `doMessage()`, contains three lines:

```
        setHeaders(inMessage);
        inMessage.setDestination(this);
        incomingObserver.onMessage(inMessage);
```

(along with some logging code, which I ignore).

Let's look at the test (which is significantly longer):

```
private MessageImpl setUpMessage() {
    MessageImpl message = control.createMock(MessageImpl.class);
    HttpServletRequest request = control.createMock(HttpServletRequest.class);
    message.get(''HTTP.REQUEST'');
    EasyMock.expectLastCall().andReturn(request);
    request.getHeaderNames();
    EasyMock.expectLastCall().andReturn(new StringTokenizer(''content-type content-length''));
    request.getHeaders(''content-type'');
    EasyMock.expectLastCall().andReturn(new StringTokenizer(''text/xml''));
    request.getHeaders(''content-length'');
    EasyMock.expectLastCall().andReturn(new StringTokenizer(''1234''));
    observer.onMessage(message);
    EasyMock.expectLastCall();
    return message;
}

@Test
public void testMessage() throws Exception {
    MessageImpl message = setUpMessage();
    control.replay();
    // ... done setting up, let's see what tested method does

    // create object
    OsgiDestination destination = new OsgiDestination(bus, endpoint, registry, ''snafu'');
    // set up to observe if message is sent to the receiver:
    destination.setMessageObserver(observer);
    // ``send'' the message
    destination.doMessage(message);

    // check that the calls expected by the mock actually happened
    control.verify();
}
```

shutdown test. Let's also investigate the test of shutdown and its implementation:

```
@Test
public void testShutdown() throws Exception {
    registry.removeDestination("snafu");
    EasyMock.expectLastCall();
    control.replay();

    OsgiDestination destination = new OsgiDestination(bus, endpoint, registry, "snafu");
    destination.shutdown();
    control.verify();
}

@Override
public void shutdown() {
    factory.removeDestination(path);
    super.shutdown();
}
```

Note that the test requirement `registry.removeDestination(‘‘snafu’’)` implies the presence of the `removeDestination` call in the implementation.

Test-Driven Development Approach on shutdown()

- We would write `testShutdown()` first. It would cause a red bar in the JUnit plugin in the development environment, since there would be no implementation of `shutdown()`.
- We would next perhaps put in the call to `super.shutdown()`; or this call might be automatically generated.
- The idea is to "make the bar green", by causing `testShutdown()` to pass, and the obvious way to do that is by adding the call to `factory.removeDestination()`.

More on Mock Objects

Martin Fowler writes that mocks aren't stubs[4]. There are four kinds of test doubles:

- Dummy objects: passed around, but never interrogated;
- Fake objects: have an implementation, but not necessarily complete (e.g. an in-memory database);
- Stub objects: provide canned answers, and may record (perhaps aggregate) information about calls to the stub, e.g. the number of messages sent;
- Mock objects: may contain canned answers, but also carry around expectations about the calls they will receive.

For instance, you could ask a stub how many messages got sent to it (verifying the state), while the mock would expect a certain number of calls to its `send` message, each with a certain list of parameters.

Both stubs and mocks are useful for testing, but they test different aspects. They are vulnerable to different types of program changes.

[4]`http://martinfowler.com/articles/mocksArentStubs.html`

Dependency Injection

Let's revisit the constructor (which I didn't show before):

```
public OsgiDestination(Bus b, EndpointInfo ei, OsgiDestinationRegistryIntf fact, String p)
    throws IOException {
    // would add the default port to the address
    super(b, null, ei, false);
    factory = fact;
    path = p;
}
```

This constructor actually exhibits a form of dependency injection[5]. The `OsgiDestination` class depends on a `Bus` and several other collaborating classes. One could imagine a class which instantiates its dependencies. But that's problematic when you want to use different kinds of objects, perhaps not known at compile-time. So the `OsgiDestination` constructor actually accepts the objects it depends on as constructor arguments.

(This is a very brief overview of DI. You can read much more about it. But the basic idea is that someone else tells you about the classes that you depend on; you only rely on the public interfaces of these classes.)

[5]`http://martinfowler.com/articles/injection.html`