Unit Testing with Stubs and Mocks

I was on site with some clients the other day, and they asked me about unit testing and mock objects. I decided to write up some of the discussion we had as a tutorial on creating dependencies (collaborators) for unit testing. We discuss two options, stubbing and mock objects and give some simple examples that illustrate the usage, and the advantages and disadvantages of both approaches.

It is common in unit tests to mock or stub collaborators of the class under test so that the test is independent of the implementation of the collaborators. It is also a useful thing to be able to do to control precisely the test data that are used by the test, and verify that the unit is behaving as expected.

Stubbing

The stubbing approach is easy to use and involves no extra dependencies for the unit test. The basic technique is to implement the collaborators as concrete classes which only exhibit the small part of the overall behaviour of the collaborator which is needed by the class under test. As an example consider the case where a service implementation is under test. The implementation has a collaborator:

```
public class SimpleService implements Service {


    private Collaborator collaborator;

    public void setCollaborator(Collaborator collaborator) {

        this.collaborator = collaborator;

    }


    // part of Service interface

    public boolean isActive() {

        return collaborator.isActive();

    }

}
```

To test the implementation of isActive we might have a unit test like this:

```
public void testActiveWhenCollaboratorIsActive() throws Exception {
```

```java
    Service service = new SimpleService();
    service.setCollaborator(new StubCollaborator());
    assertTrue(service.isActive());

}

...

class StubCollaborator implements Collaborator {
    public boolean isActive() {
        return true;
    }
}
```

The stub collaborator does nothing more than return the value that we need for the test.

It is common to see such stubs implemented inline as anonymous inner classes, e.g.

```java
public void testActiveWhenCollaboratorIsActive() throws Exception {

    Service service = new SimpleService();
    service.setCollaborator(new Collaborator() {
        public boolean isActive() {
           return true;
        }
    });
    assertTrue(service.isActive());
}
```

This saves us a lot of time maintaining stub classes as separate declarations, and also helps to avoid the common pitfalls of stub implementations: re-using stubs across unit tests, and explosion of the number of concrete stubs in a project.

What’s wrong with this picture? Well, often the collaborator interfaces in a service like this one are not as simple as this trivial example, and to implement the stub inline requires dozens of lines of empty declarations of methods that are not used in the service. Also, if the collaborator interface changes (e.g. adds a method), we have to manually change all the inline stub implementations in all the test cases, which can be a lot of work.

To solve those two problems we start with a base class, and instead of implementing the interface afresh for each test case, we extend a base class. If the interface changes, we only have to change the base class. Usually the base class would be stored in the unit test directory in our project, not in the production or main source directory.

For example, here is a suitable base class for the interface as defined:

```
public class StubCollaboratorAdapter implements Collaborator {

   public boolean isActive() {

       return false;

   }

}
```

and here is the new test case:

```
public void testActiveWhenCollaboratorIsActive() throws Exception {



    Service service = new SimpleService();

    service.setCollaborator(new StubCollaboratorAdapter() {

        public boolean isActive() {

           return true;

        }

    });
    assertTrue(service.isActive());



}
```

The test case is now insulated from changes to the collaborator interface that do not affect the isActive method. In fact, using an IDE, it will also be insulated from some changes in the interface that do affect the isActive method - for instance a name or signature change can be made automatically by the IDE in all test cases.

The inline stub approach is very useful and quick to implement, but to have more control over the test case, and ensure that if the implementation of the service object changes, the test case also changes accordingly, a mock object approach is better.

Mock Objects

Using mock objects (e.g. from EasyMock or JMock) we get a high level of control over testing the internals of the implementation of the unit under test.

To see this in practice consider the example above, rewritten to use EasyMock. First we look at EasyMock 1 (i.e. not taking advantage of Java 5 extensions in EasyMock 2). The test case would look like this:

```
MockControl control = MockControl.createControl(Collaborator.class);

Collaborator collaborator = (Collaborator) control.getMock();

control.expectAndReturn(collaborator.isActive(), true);

control.replay();



service.setCollaborator(collaborator);

assertTrue(service.isActive());



control.verify();
```

If the implementation changes to use a collaborator differently, then the unit test fails immediately, signalling to the developer that it needs to be re-written. Suppose the internals of the service changed to not use the collaborator at all:

```
public class SimpleService implements Service {
    ...



    public boolean isActive() {

        return calculateActive();

    }



}
```

The test above using EasyMock would fail with an obvious message saying that the expected method call on collaborator was not executed. In the stub implementation, the test might or might not fail: if it did fail the error meesage would be cryptic; if it did not, then it would only be accidental.

To fix the failed test we have to modify it to reflect the internal implementation of the service. The constant re-working of test cases to reflect the internals of the implementation is seen by some as a burden, but actually it is in the very nature of unit testing to have to do this. We are testing the implementation of the unit, not its contract with the rest of the system. To test the contract we would use an integration test, and treat the service as a black box, defined by its interface, not its implementation.

EasyMock 2

Note that the above test case implementation can be streamlined if we are using Java 5 and EasyMock 2:

```
Collaborator collaborator = EasyMock.createMock(Collaborator.class);

EasyMock.expect(collaborator.isActive()).andReturn(true);

EasyMock.replay(collaborator);



service.setCollaborator(collaborator);

assertTrue(service.isActive());



EasyMock.verify(collaborator);
```

There is no need for the MockControl in the new test case. Not a big deal if there is only one collaborator, like here, but if there are many, then the test case becomes significantly easier to write and read.

When to Use Stubs and Mocks?

If mock objects are superior, why would we ever use stubs? The question is likely to draw us into the realm of the religious debate, which we will take care to avoid for now. So the simple answer is, “do what suits your test case, and creates the simplest code to read and maintain”. If the test using a stub is quick to write and read, and you are not too concerned about changes to the collaborator, or uses of the collaborator internally to the unit under test, then that is fine. If the collabrator is not under your control (e.g. from a third-party library), it may often be the case that a stub is more difficult to write.

A common case where stubbing is easier to implement (and read) than mocks is where the unit under test needs to use a nested method call on the collaborator. For example, consider what happens if we change our service, so it no longer uses the collaborator’s isActive directly, but instead nests a call to another collaborator (of a different class, say Task):

```
public class SimpleService implements Service {


    public boolean isActive() {

        return !collaborator.getTask().isActive();

    }



}
```

To test this with mock objects in EasyMock 2:

```
Collaborator collaborator = EasyMock.createMock(Collaborator.class);

Task task = EasyMock.createMock(Task.class);



EasyMock.expect(collaborator.getTask()).andReturn(task);

EasyMock.expect(task.isActive()).andReturn(true);

EasyMock.replay(collaborator, task);
```

```
service.setCollaborator(collaborator);

assertTrue(service.isActive());



EasyMock.verify(collaborator, task);
```

The stub implementation of the same test would be

```
Service service = new SimpleService();

service.setCollaborator(new StubCollaboratorAdapter() {

    public Task getTask() {

        return (new StubTaskAdapter() {

            public boolean isActive() {

                return true;

            }

        }

    }

});

assertTrue(service.isActive());
```

there isn't much to distinguish between the two in terms of length (ignoring the code in the adapter base classes, which we can re-use in other tests). The mock version is more robust (for reasons discussed above), so we prefer it. But if we had to use EasyMock 1 because we were unable to use Java 5, things might be different: it would be quite a lot more ugly to implement the mock version. Here it is:

```
MockControl controlCollaborator = MockControl.createControl(Collaborator.class);

Collaborator collaborator = (Collaborator) controlCollaborator.getMock();



MockControl controlTask = MockControl.createControl(Task.class);

Task task = (Task) controlTask.getMock();
```

```
controlCollaborator.expectAndReturn(collaborator.getTask(), task);

controlTask.expectAndReturn(task.isActive(), true);



controlTask.replay();

controlCollaborator.replay();



service.setCollaborator(collaborator);

assertTrue(service.isActive());



controlCollaborator.verify();

controlTask.verify();
```

The test is half as long again, and correspondingly harder to read and maintain. Things can easily get much worse in reality. In this case we might consider the stub implementation in the interests of an easy life. Of course the true believers in mock objects will point out that this is a false economy, and the unit test will be more robust and better for the long term than the test using stubs.

Using EasyMock to Create Stub Objects

There are two competing philosophies with regard to unit testing strategies; state based testing and behavior based testing. In state based testing we configure a starting state, execute a test method, and then examine the resultant state/returned result. In behavior driven testing we ensure that our test object collaborates with its dependencies in an expected way.

When applying a strictly state driven unit test strategy we can experience an explosion in the number of stub objects in our test code. Our System Under Test (SUT) requires dependencies that operate in a predetermined way (so that we can successfully predict the outcome/resultant state). This is more complex to set up, but gives the advantage that we are less fragile in the face of changes to our SUT implementation.

Our behavior driven brethren are laughing fanatically at our test strategy ideology, whilst enjoying the benefits of EasyMock. Mock objects are different to stub objects in that they record behavior rather than returning canned values. EasyMock is a nice tool that allows us to record operations and expectations on classes, replay those expectations against the actual operations performed and then verify that what really occurred matches that which was expected. Easy mock makes building a test very easy, although if the implementation changes the order in which methods are called, or the parameters used, or indeed which methods are executed at all, our test may break. This can lead to tests which are potentially fragile, and difficult to maintain.

The ideal of testing purely for state rather than behavior is somewhat of a pipe dream however in that we still have to supply stub objects/dependencies so that our SUT can execute. We still need to make*some* assumptions about our implementation to know that it will fetch data from a particular stub, or that it will use a particular getter. It is more general however and a stub could be made general enough to support multiple test methods or multiple tests. Getting this right is what makes a unit test, a good unit test.

Wouldn’t it be nice if we could use a tool like EasyMock to generate our stubs, rather than program them over and over again, or maintaining a large set of shared stubs? EasyMock is, thankfully, powerful enough to allow us to create stubs.

In the example we will create a stub for the following interface:

```
public interface IMyService {
    public Integer getData();
    public String convert(Object object);
}
```

The code that could be added to a unit test class to provide a stub for this interface with canned responses is as follows:

```
/**
 * Sets up all mock objects.
 */
@Before
protected void prepareStubs() {
    this.myStubObject = EasyMock.createMock(IMyService.class);
    prepareStubResponses();
}

/**
 * Generates the canned responses.
 */
private void prepareStubResponses() {
    EasyMock.expect(this.myStubObject.getData()).andReturn(new Integer(5)).anyTimes();
    EasyMock.expect(this.myStubObject.convert(EasyMock.anyObject())).andReturn("ABC").anyTimes();
    EasyMock.checkOrder(this.myStubObject, false);
    EasyMock.replay(this.myStubObject);
}
```

First we create the stub object, then we configure an expected call to “getData” and we return the canned response “new Integer(5)”. The important thing that we have done here is to relax the expected number of calls to “.anyTimes()”. This means that we are not interested in the number of executions. We also switch off the expected ordering of calls. Setting “false” in check order means that EasyMock doesn’t care anymore if “getData” is called first, or if “convert” comes first. Finally we replay the stub in order to “start” it.

There is a shorter way to implement this however using “andStubReturn(..)”. The code would then look as follows:

```
private void prepareStubResponses() {
    EasyMock.expect(this.myStubObject.getData()).andStubReturn(new Integer(5));
    EasyMock.expect(this.myStubObject.convert(EasyMock.anyObject())).andStubReturn("ABC");
    EasyMock.replay(this.myStubObject);
}
```

Using “andStubReturn(..)” instead of “andReturn(..)” means that the call can be made any number of times. Order is no longer checked, so we can eliminate the “.anyTimes()” suffix and the order disabling command.

To further improve our stub code, we can use “EasyMock.createNiceMock(..)” instead of “EasyMock.createMock(..)”. A “nice” mock always returns a value regardless of whether behavior has been specified or not, rather than causing an error.

We will never verify the stub, so we are not checking that the methods are called, we don’t care about the order or the number of executions. We simply state that if this method is called, this response is to be returned. The stub can still be made general enough to be shared between tests and it carries no *more* assumptions than creating an explicit stub does, but eliminates the burden of having lots of extra classes.