A Pattern Language for Automated Testing of Indirect Inputs and Outputs using XUnit

Gerard Meszaros
ClearStream Consulting
plop2004@gerardmeszaros.com
http://testautomationpatterns.com/TestingIndirectIO.html

Abstract

Automated unit tests (A.K.A. "developer tests") and functional test (A.K.A. "customer tests") are a cornerstone of many agile development methods (such as eXtreme Programming). The availability of automated, self-checking tests allows developers to be much bolder in how they modify existing software. It is usually fairly straightforward to test the direct inputs and outputs of a class or component via it's API. But many classes and components depend on some other component that provides "indirect inputs" to the software under test. This pattern language describes key techniques used to verify that the software is interacting correctly with these "behind the scenes" components.

Pattern Template

Because these patterns are destined for publishing in a book, I have chosen to use a more "reader friendly" patterns template inspired by the one used by Martin Fowler in his more recent book: "Patterns of Enterprise Application Architecture". It is less structured than more classic pattern forms but still contains all the basic sections of a pattern.

The template starts off with the summary statement and a sketch. The summary statement captures the essence of the pattern in one or two sentences and the sketch provides a visual representation of the pattern. The next (untitled) section summarizes why you might want to use the pattern in just a few sentences. It includes the *Problem* section from the traditional pattern template. You should be able to get a sense of whether you want to read any further by reading this section.

The next two sections provide the meat of the pattern. The "How it Works" section describes the nuts and bolts of how the pattern is structured and what it is about. It also includes information about the "resulting context" when there are several ways to implement some important aspect of the pattern. This section corresponds to the "Solution" or "Therefore" sections of more traditional pattern forms.

The "When to Use It" sections describes the circumstances in which you should consider using the pattern. This section corresponds to the *Problem*, *Forces*, *Context*, and *Related Ratterns* sections of traditional pattern templates. It also includes information about the *Resulting Context* where this might affect whether you would want to use this pattern. I also include any "test smells" that would act as an indication that you should use this pattern.

Most of the concrete patterns include three additional sections. The *Motivating Example* section provides examples of what the test code might have looked like before this pattern was applied. The *Solved Example* shows what the test would look like after the pattern was applied. And the *Refactoring Notes* section provides step-by-step instructions on how to get from the "Motivating Example" to the "Solved Example".

I don't include "Known Uses" unless there is something particularly interesting about them. Most of these patterns have been seen in many, many systems and picking three uses to substantiate them would be arbitrary and meaningless.

Notation:

Underlined phrases in normal font beginning with capital letters are patterns; lowercase underlined phrases are definitions while hyperlinks in italics refer to "test smells". Hyperlinks beginning and ending with ?-marks are links to items that have not yet been written.

Scope:

This text is part of a book on patterns of XUnit test automation. Because there are a large number of patterns, I have excerpted a small subset of them for review at PLoP. The Introductory Narrative has also been highly abridged for reasons of space. For those who are interested in reading further, the most current version of this material is also available on our website at http://testautomationpatterns.com.

Testing Indirect Inputs and Outputs

Table of Contents

Patterns marked (#) have been omitted due to space considerations.

Introductory Narrative

As described in TestAutomationOverview, the SUT interacts with components through both the "front door" and the "back door". That is, software components have both an API (their front door) and make calls to the APIs of other components (their back door). Testing the interactions through the front door is the easier of the two; the test simply acts as though it were the client of the SUT and interacts through the "front door" interface.

Verifying SUT behaviour through the front door appears (at least on the surface) to be straight forward, but how do we verify that the interactions through the back door are correct? The first question we must answer is "Why do we care?" Assuming that we do care, the next question is "How do we verify it?"

Why do we care about Indirect Inputs?

Calls to depended on components often return objects, values, or even throw exceptions. Many of the execution paths within the SUT are there to deal with these different return values and to handle the various possible exceptions. Leaving these paths untested is an example of *Untested Code*. These paths can be the hardest to test effectively but they are also among the most likely to lead to failures. (… remainder omitted)

Why do we care about Indirect Outputs?

The concept of encapsulation often directs us to not care about how something is implemented. After all, that is the whole purpose of encapsulation--to alleviate the need for clients of our interface to care about our implementation. When testing, we are trying to verify the implementation precisely so our clients don't have to care about it.

```
(… remainder omitted)
```

Using Test Doubles

By now you are probably wondering about how to replace those inflexible and uncooperative real components with something that makes it easier to control indirect inputs and to verify indirect outputs.

As we've seen, to test the indirect inputs, we must be able to control the depended-on component well enough to cause it to return every possible kind of return value (valid, invalid, and exception). To test indirect outputs, we need to be able to track the calls the SUT makes to other components.

A Test Double is a type of object that is much more co-operative and let's us write tests the way we want to.

We came up with the name Test Double as the generic name for Dummies, Stubs, and Mock Objects. Feedback requested!

Types of Test Doubles

A Test Double is any object or component that we install in place of the real component specifically so that we can run a test. Depending on the reason for why we are using it, it can behave in one of three basic ways:

A ?Test Stub? is a Test Double implemented in a procedural programming language. Traditionally, they were introduced to allow debugging to proceed while waiting for other code to be ready. It was rare for them to be "swapped in" at runtime because this is hard to do in most procedural languages without introducing *?Test Logic in Production?* code.

A Mock Object is an object that is used by a test to replace a real component on which the SUT depends so that the test can either control the indirect inputs of the SUT or observe its indirect outputs. Typically, the Mock Object fakes the implementation by either returning hard-coded results or results that were pre-loaded by the test.

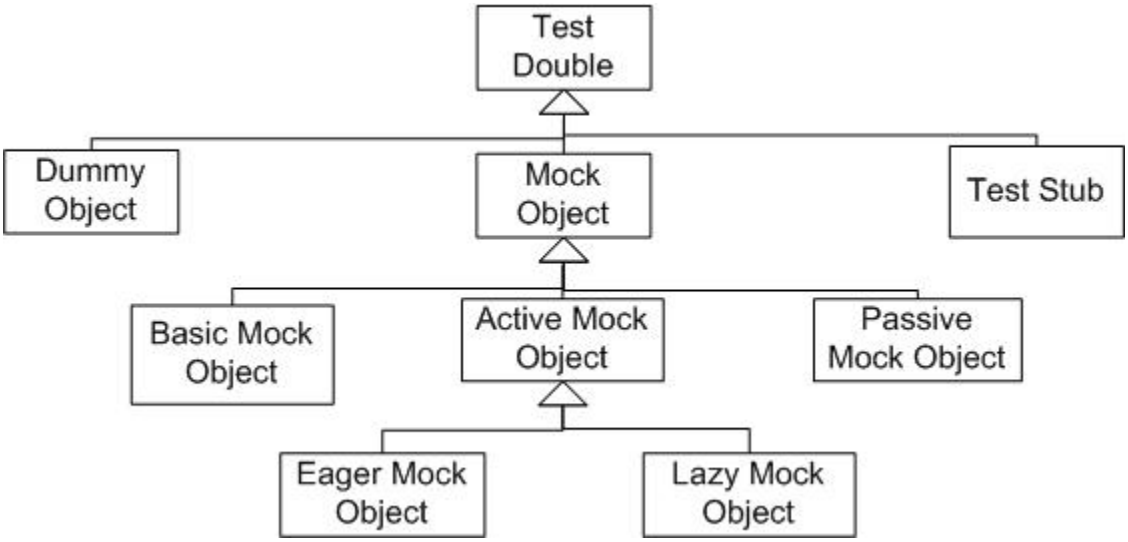
A Dummy Object (or "Test Dummy") is an object that replaces the functionality of the real depended-on component in a test for reasons other than verification of indirect inputs and outputs. Typically, it will implement the same or a subset of the functionality of the real depended-on component but in a much simpler way. While a Dummy Object is typically built specifically for testing, it is neither directly controlled nor observed by the test. The most common reason for using one is that the real depended-on component is not available yet, is too slow or is not available in the test environment. Memory Improves Speed describes how we encapsulated all database access behind a persistence layer interface and them dummied them out with hash tables and made our tests run 50 times faster. DesignForTestability describes various techniques for making your SUT easier to test.

Mock Objects come in several flavours including:

- A Basic Mock Object is an object that can be programmed with the indirect inputs (values to return and exceptions to throw when its methods are called.) This allows *Untested Code* paths in the SUT to be exercised that might otherwise be impossible to hit.
- A Passive Mock Object adds the capability to quietly record all the calls made to its methods. In the verification part of the test, the test verifies the calls it received thus allowing the test to perform Procedural Behaviour Verification on those calls via a series of assertions. One way to access the actuals calls received is by defining a Retrieval Interface on the Passive Mock Object.
- An Active Mock Object is an object that is programmed with all the Expected Behaviors that it should expect to see from the SUT. There are two basic variations of Active Mock Object: An Eager Mock Object fails the test immediately upon receiving an unexpected or incorrectly formed call while a Lazy Mock Object waits until the Final Verification Method is called to compare the actual calls it received with the pre-programmed expected calls. They make it possible to reuse the logic used to verify the indirect outputs of the SUT.

Passive Mock Objects and Active Mock Objects often include the capability of Basic Mock Objects to be programmed with any indirect inputs required to allow the SUT to advance to the point where it would generate the indirect outputs they are verifying.

This is all summarized in the following diagram:



Programming the Test Double

Most Test Doubles (with the exception of Dummy Objects) need to be told what values to return and/or what values to expect. A Hard-coded Test Double is told at design time by the Test Double developer while a Programmable Test Double is told at run-time by the test. In both cases, it is the test automater who ultimately decides so the primary considerations are code reuse of the Test Double and understandability of the test.

(Because of the nature of Dummy Objects, they do not need to be "programmed" at run-time. Instead, they are just used by the SUT and later outputs depend on the earlier calls by the SUT.)

?Test Stubs? are typically built as Hard-coded Test Doubles. That is, they are hard-coded to return a particular value when a certain function is called. This is the simplest form of test double.

Some kinds of test doubles are more likely to be built as Programmable Test Doubles and programmed as part of setting up the test while others are not. In particular, Basic Mock Objects and Active Mock Objects frequently need programming. By their very nature, Dummy Objects do not need programming because they are not used to verify behaviour and Hard-Coded Test Doubles do not because their behaviour is fixed at the time they are coded. A Basic Mock Object or a Passive Mock Object only needs to be programmed with the values to be returned by the methods we expect the SUT to invoke. An Active Mock Object needs to be programmed with the expected names and arguments of all the methods we expect the SUT to invoke on it.

Programmable Test Doubles can provide either a Programming Interface, or a Programming Mode that the tests use to program the Test Double with the values to return or verify. This makes these Programmable Test Double reusable across many tests. It also makes the test more understandable by making the values used by the Test Double visible within the test thus preventing visits by a *?Mystery Guest?*.

So where should all this programming be done? In FixtureSetup we discuss several alternatives including Inline Setup, Implicit Setup and Delegated Setup. The installation of the test double should be treated just like any other part of fixture setup. We encourage you to choose the fixture setup pattern that leads to the most understandable tests!

Installing The Test Double

Before we exercise the SUT, we need to install any Test Doubles on which our test depends. The normal sequence is to instantiate the double, program it if necessary and then install it into the SUT. We can install the test-specific replacement for the real DOC in any one of the following ways: (… omitted)

(ommited from this review)

Other Uses of Test Doubles

Endoscopic Testing

?[Mackinnon]? et al introduced the concept of ?Endoscopic Testing? in their initial Mock Objects paper. Endo-testing involves testing the SUT from the inside by passing in an Active Mock Object as an argument to the method under test. This allows verification of certain internal behaviours of the SUT that may not be at all visible from the outside.

The classic example they describe is the use of a mock collection class pre-loaded with all the expected members of the collection. When the SUT tries to add an unexpected member, the mock collection's assertion fails. This allows the full stack trace of the internal call stack to be visible in the JUnit failure report. If your IDE supports breaking on specified exceptions, you can also inspect the local variables at the point of failure.

Speeding Up Fixture Setup

Another use of Test Doubles is to reduce the runtime cost of Clean Slate test fixture setup. When the SUT needs to interact with other objects that are difficult to create because they have many dependencies, a single Test Doubles can be created instead of the complex network of objects. When applied to networks of entity objects, this technique is called Entity Chain Snipping.

Speeding Up Test Execution

Another use of Test Doubles is to improve the speed of tests by replacing slow components with faster ones ?Stub Out Slow Component?. Replacing a relational database with an in-memory Dummy Object can reduce test execution times by an order of magnitude! The extra effort of coding the dummy database is more than offset by the reduced waiting time and the quality improvement due to the more timely feedback that comes from running the tests more frequently.

Cite or include material from XP2001 paper (and XP Perspectives chapter) on "Improving the Efficiency of Automated Tests".

Other Considerations

When the SUT delegates to several depended-on components, we may want to test that the methods on the depended-on components are called in the right order, not just within a single depended-on component, but also that they are interleaved properly.

To do this, the test should create an instance of a ?Test Double Helper Class? and pass it to each Active Mock Object as it is instantiated. Their methods should be implemented by delegating to the ?Shared Test Double Helper?. Since the assertions are done by the ?Shared Test Double Helper? which sees all the calls made to all the mocked depended-on components, this ensures that the calls are interleaved properly.(Note: If there is any chance that several depended-on components will have methods with the same signature, the mock helper should also be passed the name of the mock component that is delegating to it.)

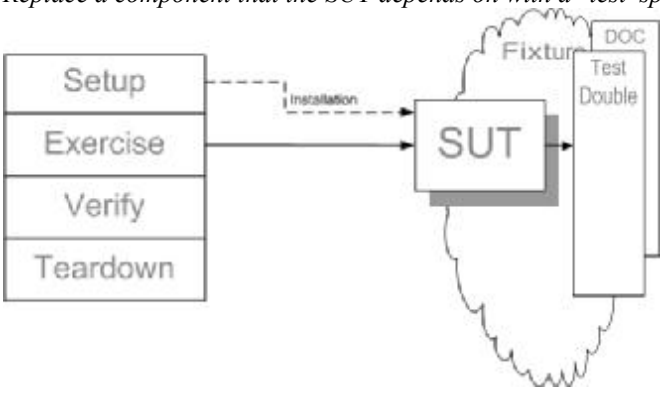
Since most of our tests will involve replacing the real depended-on component with a mock object, how do we know that it works properly when the real depended-on component is used? Of course, we would expect our functional tests to verify behaviour with the real depended-on components in place (except, of course, when the real depended-on components are interfaces to other systems that need to be stubbed out during single-system testing.) We should have a unit test, a ?Stubbable Initialization Test? to verify that the real depended-on component is installed properly. The trigger for writing this test is the first test that replaces the depended-on component with a mock since that is often when the mockable depended-on component mechanism is first built.

Finally, we want to be careful that we don't fall into the "new hammer trap". ("When you have a new hammer, everything looks like a nail"). Overuse of mock objects or stubs can lead to *?Overspecified Software?* by encoding implementation-specific information about the design in your tests. This can make it harder to change the design

because many tests are impacted by the change only because they use a Test Double that has be affected by the design change.

Test Double

Replace a component that the SUT depends on with a "test-specific equivalent".



Sometimes it is just plain hard to test the SUT because it depends on other components that cannot be used in the test environment. This could be because they aren't available, they will not return the results needed for the test or because executing them would have undesirable side effects.

When we are writing a test in which we cannot use a real depended-on component, we can replace it with a *Test Double*. The *Test Double* doesn't have to behave exactly like the real depended-on component; it merely has to provide the same API as the real one so that the SUT *thinks* it is the real one!

How It Works

When the movie industry wants to film something that is potentially risky or dangerous for the leading actor to carry out, they hire a "stunt double" to take the place of the actor in the scene. The stunt double is a highly trained individual who is capable of meeting the specific requirements of the scene. They may not be able to act, but they know how to fall from great heights, crash a car, or whatever the scene calls for. How closely the stunt double needs to resemble the actor depends on the nature of the scene. Usually, things can be arranged such that someone who vaguely resembles the actor in stature can take their place.

For testing purposes, we can replace the real depended-on component (not the SUT!) with *our* equivalent of the "stunt double": the *Test Double*. During the ?fixture setup? phase of our Four Phase Test, we replace the real depended-on component with our *Test Double*. Depending on the kind of test we are executing, we may hard-code the behaviour of the SUT or we may "program" it (like a VCR) during the setup phase. When the SUT interacts with the *Test Double*, it won't be aware that it isn't talking to the real McCoy, but we will have achieved our goal of making impossible tests possible.

Regardless of which of the variations of *Test Double* you choose to use, keep in mind that you don't need to implement the whole interface of the depended-on component! Just provide whatever functionality is needed for your particular test. You can even build different *Test Doubles* for different tests that involve the same depended-on component.

When To Use It

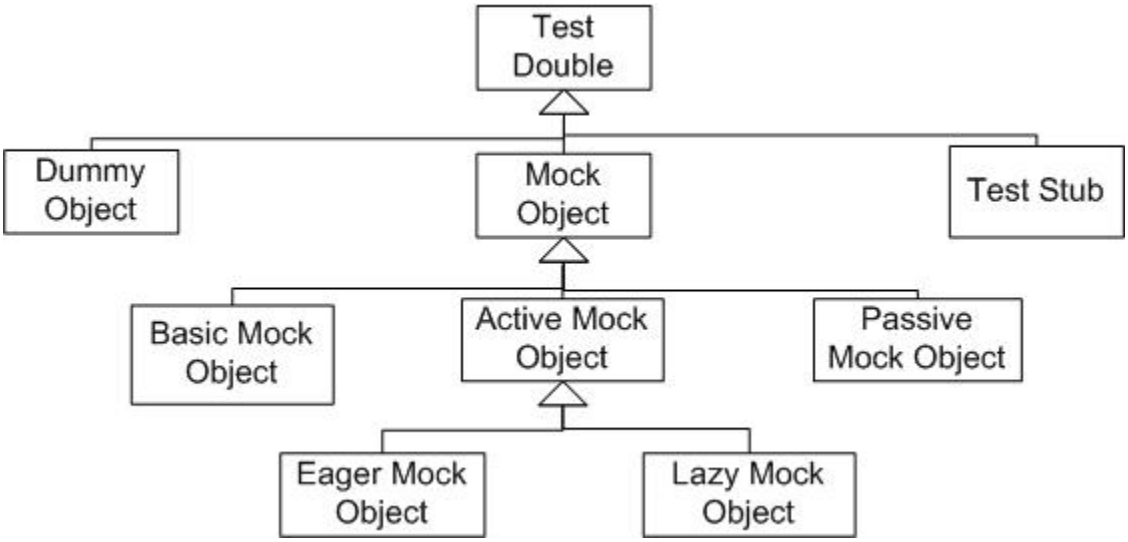
There are several different circumstances in which you might want to use some sort of *Test Double* during your tests.

- If you have an *Untested Requirement* and you cannnot verify the requirements using the context in which the SUT normally operates because it does not provide an ?observation point? for the indirect output you need to verify.
- If you have *Untested Code* or *?Untestable Behaviour?* and a depended-on component (DOC) does not provide the ability to control the indirect input of the SUT.
- If you have *?Slow Tests?* and you want to be able to run your tests more quickly and hence more often.

Each of these can be addressed in some way by using a *Test Double*. But you have to be careful when using *Test Doubles* because you are testing the SUT in a different configuration from that which will be used in production. So you really should have at least one test that verifies it works without a *Test Double*. Beware that you don't replace the parts of the SUT that you are trying to verify as this can result in tests that test the wrong software! Also, excessive use of *Test Doubles* can result in *?Overspecification of Software?*.

Variations

Test Doubles come in several main flavours as summarized in the following diagram. The subflavours of the top-level variations are described in more detail in the pattern write-ups of the variations.



Dummy Object

A Dummy Object typically implements the "same" functionality as the object it replaces but in a much simpler way. Typically, it is "swapped in" at runtime for use in a particular test. While a Dummy Object is typically built specifically for testing, it is neither directly controlled nor observed by the test.

Mock Object

A Mock Object fakes the implementation by either returning hard-coded results or results that were pre-loaded by the test. Typically, it is "swapped in" at runtime for use in a particular test. A key characteristic is that its main purpose for being is to give the test better control/visibility of the SUT. Mock Objects come in several flavours that are described in more detail in their own patterns.

Test Stub

A *Test Double* implemented in a procedural programming language is often called a ?Test Stub?. Traditionally, they were introduced to allow debugging to proceed while waiting for other code to be ready. It was rare for them to be "swapped in" at runtime.

Implementation Notes

We build the *Test Double* to be the “the simplest thing that could possibly work” (which might be a completely hard-coded implementation!) Depending on the behaviour we are trying to test, we may use a Programmable Test Double (whose behaviour is directly controlled by the test) or a Hard-Coded Test Double which always does exactly the same thing every time it is used.

Test Doubles can be built a number of different ways including ?Hand-Built Test Double?, ?Static Test Double Generation? or ?Dynamic Test Double Generation?. Before the SUT can be exercised, the *Test Double* must be installed as part of your fixture setup logic. Use one of the ?Test Double Installation? patterns to install it. Some *Test Doubles* need to be programmed before you exercise the SUT. (Refer to the specific pattern for details.)

Motivating Example

Because there are a wide variety of reasons for using the variations of *Test Doubles* it is hard to provide a single example that characterizes the motivation behind each style. Please refer to the examples in the associated patterns.

Solution Example

Because there are a wide variety of *Test Doubles*, please refer to the examples in the associated patterns.

Dummy Object

Replace a component that the SUT depends on with a much lighter-weight implementation.

Sketch goes here.

The SUT often depend on other components or systems. The interactions with these other components may be necessary but the side-effects of these interactions may be unnecessary or even detrimental.

A *Dummy Object* is a much simpler and lighter weight implementation of the functionality provided by the DOC without the side effects we choose to do without.

How It Works

Build a very lightweight implementation of the same functionality as provided by a component that the SUT depends on and instruct the SUT to use it instead of the real DOC. This implementation need not have any of the "-ilities" that the real DOC needs to have (such as scalability); it need only provide the equivalent services to the SUT so that the SUT isn't aware it isn't using the real DOC.

A *Dummy Object* is similar to a Mock Object in many ways including the need to install into the SUT. But while a Mock Object specifies the interactions between the SUT and the DOC, the *Dummy Object* does not. Merely provides a way for the interactions to occur in a self-consistent manner. These interactions (between the SUT and the *Dummy Object*) will typically be many and the values passed in as arguments of earlier method calls will often be returned as results of later method calls. (Contrast this with Mock Objects where the responses are either hard-coded or programmed by the test.)

While the test does not normally "program" a *Dummy Object*, complex fixture setup that would normally involve initializing the state of the DOC may also be done with the *Dummy Object*. Techniques like Data Loader and Back Door Fixture Setup can be used quite successfully without fear of *?Overspecification of Software?* (because the implementation of the *Dummy Object* is our primary concern.)

When To Use It

Use a *Dummy Object* whenever your SUT depends on other components that make testing difficult or slow (e.g. *?Slow Tests?*) and the tests need more complex behaviour or more complex sequences of behaviour than is worth implementing in a ?Test Stub? or Mock Object and it is easier to create a lightweight implementation than it would be to build and program suitable Mock Objects.

Using a *Dummy Object* helps avoid *?Overspecification of Software?* by not encoding within the test the exact calling sequences expected of the DOC. The SUT can how many times the methods of the DOC are called without causing tests to fail.

If you need to control either the indirect inputs or indirect outputs of the SUT, you should probably be using a Mock Object instead.

Implementation Notes

Most *Dummy Objects* are hand-built. Often, the *Dummy Object* is used to replace a real implementation that suffers from latency issues due to real messaging or disk i/o with a much lighter *in memory* implementation. With the rich

class libraries available in most object-oriented programming languages, it is usually possible to build a dummy implementation with relatively little effort that is sufficient to satisfy the need of the SUT, at least for the purposes of specific tests.

A commonly used strategy is to start by building a *Dummy Object* to support a specific set of tests where the SUT requires only a subset of the services of the DOC. If this proves successful, consider expanding the *Dummy Object* to handle additional tests. Over time, you may find that you can run all your tests using the *Dummy Object*. (See Memory Improves Speed for a description of how we dummied out the entire database with hash tables and made our tests run 50 times faster.)

Motivating Example

In this example, we have a SUT that needs to read and write records from a database. The test must set up the fixture in the database (several writes), the SUT interacts (reads and writes) with the database several more times and and these test then removes the records from the database (several deletes). All this takes time. Several seconds per test. This very quickly adds up to minutes and we find our developers aren't running the tests very often any more.

Refactoring Notes

The steps for introducing a *Dummy Object* are very similar to those for adding a Mock Object. If one doesn't already exist, introduce a way to substitute the *Dummy Object* for the DOC, usually a field (attribute) to hold the reference to it. You may have to do an ?Extract Interface? refactoring before you can introduce the dummy implementation. (In statically typed languages, use this interface as the type of the variable that holds the reference to the DOC.)

Solution Example

In this example, we've created a *Dummy Object* that replaces the database; a "dummy database" implemented entirely in memory using hash tables. The test doesn't change a lot, but the test execution occurs much, much faster.

And here's the implementation of the *Dummy Object*:

Now our developers are more than happy to run all the tests after every code change.

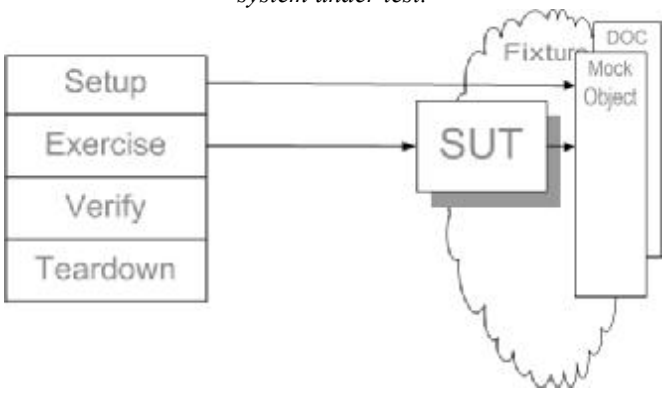
Further Reading

Memory Improves Speed describes how we dummied out the entire database with hash tables and made our tests run 50 times faster.

[UTMJ] describes the use of "dummy objects" in various circumstances.

Mock Object

Replace a real object with a test-specific object that can be controlled and observed by the test as it is used by the system under test.



In many circumstances, the environment or context in which the SUT operates very much influences the behaviour of the SUT. In other cases, we must peer inside the SUT to be able to determine whether the expected behaviour has occurred. Both of these circumstances are made possible by the use of *Mock Objects*.

Mock Objects are powerful tools we can use to gain control of the context to make our tests more effective, robust and repeatable.

How It Works

Mock Object works by replacing one or more real objects in the SUT with a test-specific stand-in that can be programmed by the test to behave it specific ways. This gives the test much more control over the context of the SUT and makes it possible to test scenarios that would otherwise be very difficult or impossible to test.

The term *Mock Object* is used to refer to a group of Test Double techniques. In some contexts, the term is used to describe an Active Mock Object or, more specifically, an Eager Mock Object. Unfortunately, the term has also become synonymous with any variation of Test Double that is implemented as an object (as opposed to a ?Test Stub? which some people interpret to mean an early, hard-coded version of a *function or procedure*.) This set of patterns is an attempt at establishing a single common vocabulary for the various techniques.

When To Use It

A key indication for using a Mock Object is having either *Untested Code* caused by the inability to control the context in which the SUT operates or an *Untested Requirement* caused by an inability to observe side-effects of invoking methods on the SUT. Use of a *Mock Object* allows the test to control the indirect inputs of the SUT thus improving code coverage. It also provides a means to verify the indirect outputs of the SUT.

If the test doesn't want to control the indirect inputs or verify the indirect outputs of the SUT but still needs to replace the real depended-on component with a lightweight implementation for other reasons, consider using a ?Test Dummy?.

Use of a *Mock Object* will improve the test coverage of your code but the tests will be more complicated than they were before you tried to improve the test coverage. And introducing a mechanism for replacing the real DOC with a *Mock Object* can be difficult if the SUT was not designed for testability. One common bug (which is quickly found) is that the SUT works well when the DOC is mocked out but doesn't work at all when no *Mock Object* is installed. A special form of constructor test, the ?Stubbable Initialization Test?, can ensure this does not occur.

Beware that you don't replace the parts of the SUT that you are trying to verify as this can result in tests that test the wrong software! Note that excessive use of *Mock Objects* can result in *?Overspecification of Software?* which can make refactoring excessively difficult or expensive and thereby discourage continuous improvement of the design.

Variations

Mock Objects comes in several flavours:

Basic Mock Object

When writing tests to address *Untested Code* rooted in an inability to cause the SUT to go down certain "defensive" code paths, we can use a simple Basic Mock Object to feed indirect inputs into the SUT by replacing a component that the SUT depends on.

Passive Mock Object

In addition to the capabilities of a Basic Mock Object, a Passive Mock Object also records all the calls made to its methods so that the test can inspect them during the verification phase of the Four Phase Test to find out how the *Mock Object* was used by the SUT.

Active Mock Object

Most of the Mock Objects literature implicitly assumes an Active Mock Object and more specifically, a variation we call Eager Mock Object. In addition to the capabilities of a Basic Mock Object, an Active Mock Object is also pre-programmed with the method calls it should expect and it verifies each of the method calls made to it against the expected calls. Use an Active Mock Object when verifying indirect outputs of the SUT or when doing ?Endoscopic Testing?.

Implementation Notes

There are several different ways to implement your *Mock Object* including ?Common Interface Test Doubles?, ?Anonymous Test Double Class? and ?Self Shunts?.

If available for your member of the XUnit family, use ?Static Test Double Generation? or ?Dynamic Test Double Generation? tools as an alternative to building your own ?Hand-Built Test Doubles?.

You'll need to use one of the ?Test Double Installation? patterns to tell the SUT to use the *Mock Object*.

Motivating Example

The following test attempts to verify the basic functionality of a component that formats some information for display to the user. The formatting logic includes several different scenarios depending on the values returned by the business logic component. Because the latter is time dependent, theses tests are non-deterministic. In fact, you can never have all of them pass in the same test run because they all require a different time to be returned by the TimeProvider !

```
    public void testDisplayCurrentTime_AtMidnight() {
       // fixture setup
       TimeDisplay sut = new TimeDisplay();
       // exercise sut
       String result = sut.getCurrentTimeAsHtmlFragment();
       // verify direct output
       String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
       assertEquals( expectedTimeString, result);
    }

   public void testDisplayCurrentTime_AtOneMinuteAfterMidnight() {
      String expectedTimeString = "<span class=\"tinyBoldText\">12:01 AM</span>";
      TimeDisplay actualTimeDisplay = new TimeDisplay();
      assertEquals("12:01 AM", expectedTimeString,
                           actualTimeDisplay.getCurrentTimeAsHtmlFragment());
   }

    public void testDisplayCurrentTime_AtNoon() {
       TimeDisplay sut = new TimeDisplay();
       String expectedTimeString = "<span class=\"tinyBoldText\">Noon</span>";
       assertEquals( expectedTimeString, sut.getCurrentTimeAsHtmlFragment());
    }
```

We've shown the first test in the fleshed out format with 3 of the Four Test Phases broken out. The other tests are in a more compact format to save space. Most of the time, all these tests fail.

Refactoring Notes

Verification of indirect outputs can be added to existing tests merely by adding code to the fixture setup logic of your test to create the mock object, programming it with values to return to the SUT when itis called, and installing it into the SUT.

You may want to do an ?Extract Interface? refactoring on the depended-on component to make it easier to use mock object generation tools.

Solution Example

In this improved version of the test, mockProvider is our *Mock Object* . The method setHours and setMinutes are used to program it with the time it should return. In the first test, the statement sut.setTimeProvider(mockProvider) installs the *Mock Object* using the ?Overridable Attribute? test double installation pattern. In the subsequent tests, we pass the mockProvider to the TimeDisplay component as an optional ?Test Double Constructor Argument?;

```
   public void testDisplayCurrentTime_AtMidnight() {
      // fixture setup
      TimeDisplay sut = new TimeDisplay();
      // mock setup
      MockTimeProvider mockProvider = new MockTimeProvider();
      mockProvider.setHours(0);
```



```
      mockProvider.setMinutes(0);
      sut.setTimeProvider(mockProvider);
      // exercise sut
      String result = sut.getCurrentTimeAsHtmlFragment();
      // verify direct output
      String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }

   public void testDisplayCurrentTime_AtOneMinuteAfterMidnight() {
      String expectedTimeString = "<span class=\"tinyBoldText\">12:01 AM</span>";
      TimeDisplay actualTimeDisplay = new TimeDisplay(new MockTimeProvider(0, 1));
      assertEquals("12:01 AM", expectedTimeString, actualTimeDisplay
            .getCurrentTimeAsHtmlFragment());
   }
   public void testDisplayCurrentTime_AtNoon() {
      String expectedTimeString = "<span class=\"tinyBoldText\">Noon</span>";
      TimeDisplay actualTimeDisplay = new TimeDisplay(
            new MockTimeProvider(12, 0));
      assertEquals("Noon", expectedTimeString, actualTimeDisplay
            .getCurrentTimeAsHtmlFragment());
   }
```

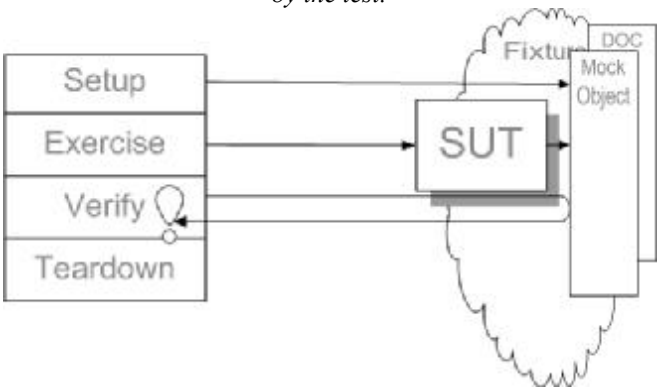
Note that the variable holding the mockProvider has been declared to be of type MockTimeProvider even though the ?Test Double Installation? mechanism only requires something that implements the TimeProvider interface. This is done to expose the Programming Interface methods to the test.

All this was made possible through the use of the following *Mock Object*. This one was a ?Hand-Coded Test Double? and in the interest of space, we'll just show you what the Programming Interface looks like:

```
   private Calendar myTime = new GregorianCalendar();
   /**
    * The complete constructor for the TimeProviderTestStub
    * @param hours specify the hours using a 24 hour clock
    *    (e.g. 10 = 10 AM, 12 = noon, 22 = 10 PM, 0 = midnight)
    * @param minutes specify the minutes after the hour
    *    (e.g. 0 = exactly on the hour, 1 = one minute after the hour, etc.)
    */
   public MockTimeProvider(int hours, int minutes) {
      setTime(hours, minutes);
   }

   public void setHours(int hours) {
      myTime.set(Calendar.HOUR_OF_DAY, hours);
   }

   public void setMinutes(int minutes) {
      myTime.set(Calendar.MINUTE, minutes);
   }

   public void setTime(int hours, int minutes) {
      setHours(hours);
      setMinutes(minutes);
   }
```


Passive Mock Object

Use a Test Double to capture the indirect output calls made to another component by the SUT for later verification by the test.



In many circumstances, the environment or context in which the SUT operates very much influences the behaviour of the SUT. To get good enough visibility of the indirect outputs of the SUT, we may have to replace some of the context with something we can use to capture these outputs of the SUT.

Use of a *Passive Mock Object* is a simple and intuitive way to expose the indirect outputs of the SUT so they can be verified.

How It Works

Before exercising the SUT, install a Test Double as a stand-in for an object or component used by the SUT. Design the Test Double such that it records the values that were passed to it in any operations delegated to it by the SUT as it is exercised. After the SUT has been exercised, the test compares the actual values passed to the Test Double by the SUT with the values expected by the test.

When To Use It

A key indication for using a Mock Object is having an *Untested Requirement* caused by an inability to observe side-effects of invoking methods on the SUT. The requirement *Passive Mock Objects* are a natural and intuitive way to extend the existing tests to also cover these indirect outputs because the calls to the Assertion Methods are invoked by the test after the SUT has been exercised just like in "normal" tests.

Use a *Passive Mock Object* if any of the following are true:

- You are verifying the indirect outputs of the SUT and you can **cannot** predict the value of all attributes of the interactions with the SUT ahead of time.
- You want the assertions to be visible in the test to avoid *?Mystery Guest?* result verification.
- Your test requires test-specific equality therefore you cannot use the standard definition of equality as implemented in the SUT **and** you are using tools that generate the Mock Object but which do not give you control over the Assertion Methods that are being called.
- A failed assertion cannot be reported effectively back to the Test Runner. This might occur if the SUT is running inside a container that catches all exceptions and makes it difficult to report the results

If none of these are true, you may want to consider using an Active Mock Object. If you are trying to address *Untested Code* by controlling the indirect inputs of the SUT, a Basic Mock Object may be all you need.

Because a *Passive Mock Object* does not fail the test at the first deviation from the expected behaviour like an Eager Mock Object, your tests will be able to include more detailed diagnostic information in the Assertion Message based on information gathered after an Eager Mock Object would have failed the test. But at the point of test failure, only the information within the test itself is available be used in the calls to the Assertion Methods. If you need to include information that is only accessible while the SUT is being exercised, you'll have to use an ?Eager Mock Object?.

Of course, you won't be able to use any Test Double unless you have a way to tell the SUT that you want it to use the Test Double.

Implementation Notes

There are several ways the test can access the actual parameters passed by the SUT to the *Passive Mock Object*. The first approach involves the definition and use of a Retrieval Interface on the Test Double. (This is the most common approach to *Passive Mock Object*.) The second alternative is to use ?callbacks? from the *Passive Mock Object* to the test to save the actual parameters. (The test needs to pass a reference to itself to the *Passive Mock Object* during fixture setup and implement an interface that the *Passive Mock Object* is aware of.) A variation on this is for the test to use itself as the *Passive Mock Object*. In this case, the Testcase Class must implement the same interface as the object it is standing in for and the implementation just saves the actual parameters in instance variables for later use in the test. This approach is called ?Self Shunt?. Yet another possibility is to have the *Passive Mock Object* store the actual parameters in a well-known place where the test can access them. E.g. It could save them in a file.

In each case, what makes the Test Double a *Passive Mock Object* is the fact that the Assertion Methods are called from within the test itself **after** the SUT has been exercised rather than being called by the Mock Object itself. This is what gives the test full control over how the indirect outputs are verified (e.g. using Custom Assertions instead of Equality Assertions that depend on the equals defined in the SUT.) But it also means that the test has to do all the work and each test that uses the same *Passive Mock Object* probably has to do a similar set of assertions so this may result in *?Test Code Duplication?*. Of course, the tests could use some other mechanism (such as Custom Assertions accessed via a ?Test Helper?) to enable reuse of the assertion logic.

Passive Mock Objects can be built as ?Hand-coded Test Doubles? or, if tools exist for your variant of XUnit, you can use ?Static Mock Generation? or ?Dynamic Mock Generation? tools to build your *Passive Mock Object*. If the SUT requires indirect inputs from the *Passive Mock Object*, you can use either a Programming Interface or Programming Mode to program it with any indirect inputs required to run the test. Use one of the ?Test Double Installation? patterns to install it *before* you exercise the SUT.

Motivating Example

The following test verifies the basic functionality of creating a flight. But it does not verify the indirect outputs of the SUT, namely, it is expected to log each time a flight is created along with data/time and userid of the requester.

```
public void testCreateFlight() throws Exception {
   // setup
   FlightDto expectedFlightDto = helper.createAnonymousUnregisteredFlightDto();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();

   // exercise
   FlightDto actualFlightDto = facade.createFlight(
         expectedFlightDto.getAirline(), expectedFlightDto.getDepartureTime(),
         expectedFlightDto.getOriginAirportId(),
         expectedFlightDto.getArrivalTime(),
         expectedFlightDto.getDestinationAirportId());

   // validation
   assertFlightDtosEqual(expectedFlightDto, actualFlightDto);
}
```

Refactoring Notes

Verification of indirect outputs can be added to existing tests merely by adding code to the fixture setup logic of your test to create the mock object, programming it with values to return and installing it. At the tend of the test, add assertions comparing the expected names and arguments of the indirect outputs with the actual values retrieved from the *Passive Mock Object* using the Retrieval Interface.

Solution Example

In this improved version of the test, mockLog is our *Passive Mock Object*. The statement facade.setAuditLog(mockLog) installs the *Passive Mock Object* using the ?Mock as Overridable Attribute? pattern. The methods getDate, getActionCode, etc. are the Retrieval Interface used to retrieve the actual arguments of the call to the logger.

```
public void testCreateFlight_passiveMock_inline() throws Exception {
   //   fixture setup
   FlightDto expectedFlightDto = helper.createAnonymousUnregisteredFlightDto();
```



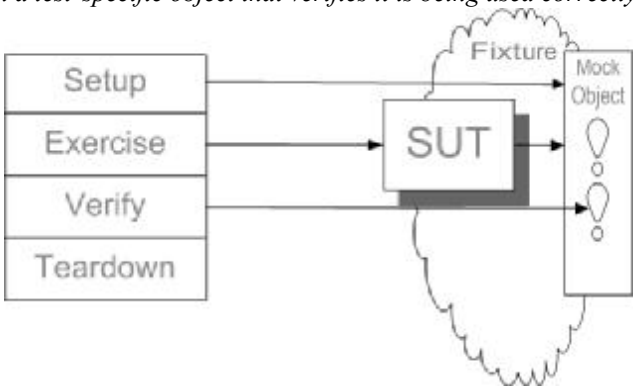
```
      FlightManagementFacade facade = new FlightManagementFacadeImpl();
      //   mock setup
      PassiveMockAuditLog mockLog = new PassiveMockAuditLog();
      facade.setAuditLog(mockLog);
      // exercise
      FlightDto actualFlightDto = facade.createFlight(
            expectedFlightDto.getAirline(),   expectedFlightDto.getDepartureTime(),
            expectedFlightDto.getOriginAirportId(),
            expectedFlightDto.getArrivalTime(),
            expectedFlightDto.getDestinationAirportId());
      // verify direct output of SUT
      assertFlightDtosEqual(expectedFlightDto, actualFlightDto);
      // verify indirect outputs of SUT
      assertEquals("number of calls", 1, mockLog.getNumberOfCalls());
      assertEquals("action code", helper.CREATE_FLIGHT_ACTION_CODE,
mockLog.getActionCode());
      assertEquals("date", helper.getTodaysDateWithoutTime(), mockLog.getDate());
      assertEquals("user", helper.TEST_USER_NAME, mockLog.getUser());
      assertEquals("detail", actualFlightDto.getFlightNumber(), mockLog.getDetail());
   }
```

This test depends on the following definition of the *Passive Mock Object*:

```
public class PassiveMockAuditLog implements AuditLog {
    private Date date;
    private String user;
    private String actionCode;
    private Object detail;
    private int numberOfCalls = 0;

     public void logMessage(Date date, String user, String actionCode, Object detail)
{
         this.date = date;
         this.user = user;
         this.actionCode = actionCode;
         this.detail = detail;

        numberOfCalls++;
    }
    public int getNumberOfCalls() {
        return numberOfCalls;
    }
    public Date getDate() {
        return date;
    }

   public String getUser() {
        return user;
    }

    public String getActionCode() {
        return actionCode;
    }

    public Object getDetail() {
        return detail;
    }
}
```


Active Mock Object

Replace a real object with a test-specific object that verifies it is being used correctly by the system under test.



In many circumstances, the environment or context in which the SUT operates very much influences the behaviour of the SUT. In other cases, we must peer inside the SUT to be able to determine whether the expected behaviour has occurred. To get good enough control or visibility, we must replace some of the context with something we can control, a Test Double. When the behaviour of the Test Double is controlled entirely by the test (or hard-coded), we call that a Mock Object.

We can avoid *?Test Code Duplication?* between similar tests by giving the Mock Object the information it needs to verify these outputs itself. This is called an *Active Mock Object*.

An alternative wording:

An *Active Mock Object* is a powerful way to verify the indirect outputs of the SUT by giving the Mock Object the information it needs to verify these outputs itself thus avoiding *?Test Code Duplication?* between similar tests.

How It Works

First, we define a mock implementation of an interface on which the SUT depends. Then, during the test, we program the mock with the values with which it should respond to the SUT **and** the method calls (complete with expected arguments) to expect from the SUT. Before exercising the SUT, we install the mock so that the SUT uses it **instead of** the real implementation. When called during SUT execution, it is the *Active Mock Object* itself which compares the actual arguments received with the expected arguments using Equality Assertions and fails the test if they don't match. The test need not do any assertions at all!

As most people mean Eager Mock Object when they say *Active Mock Object* most of the comments in this description apply to the eager variety. See the description of Lazy Mock Object, the less-commonly used "country relative" for information specific to that variety.

When To Use It

A key indication for using a Mock Object is having either *Untested Code* caused by the inability to control the context in which the SUT operates or an *Untested Requirement* caused by an inability to observe side-effects of invoking methods on the SUT.

Use an *Active Mock Object* if you are verifying the indirect outputs of the SUT and you can predict the value of all attributes of the interactions before the SUT is exercised **and** you can live with the standard definition of equality as implemented in the SUT. If you cannot predict all the values or you don't want to compare all of them, you may have to use a Passive Mock Object instead. (A Passive Mock Object merely records the method calls and parameter values and leaves it to the test to access them via the Retrieval Interface. This makes it possible to use test-specific equality by calling Custom Assertions for determining equality.)

Because an *Active Mock Object* will fail the test at the first deviation from previously programmed expected behaviour, you won't know what would have happened after the test was failed by the *Active Mock Object*. Using an Eager Mock Object, you **will** be able to include more detailed diagnostic information that is only available while the SUT is executing in the Assertion Message than you might have if you had used a Passive Mock Object .

Do not use a *Active Mock Object* if a failed assertion cannot be reported effectively back to the Test Runner. This may be the case if the SUT is running inside a container that catches and eats all exceptions. In these cases you may be better off using a Passive Mock Object instead.

Note that tests written using *Active Mock Objects* look different from more traditional tests because all the Expected Behaviour must be specified **before** the SUT is exercised. This makes them harder to write and to understand for test automation neophytes. This factor may be enough to cause you to prefer writing your tests using Passive Mock Objects.

Active Mock Objects (specifically Eager Mock Objects, and and especially those created using ?Dynamic Test Double Generation?) tend to use the equals method of the various objects being compared. If your test-specific equality is different from how the SUT would interpret equals, you may not be able to use *Active Mock Object* or you may be forced to add an equals method where you didn't need one. This smell is called *?Equality Pollution?*.

Implementation Notes

If available in your language, use ?Static Test Double Generation? or ?Dynamic Test Double Generation? tools to build your *Active Mock Object*. Otherwise, you may have to resort to a ?Hand-Built Test Double? such as an ?Inner Test Double?.

When verifying indirect outputs of the SUT or when doing ?Endoscopic Testing?, you'll want to use an Active Mock Object or a Passive Mock Object to intercept the SUT's indirect outputs.

Of course, you must have a way of installing a Test Double into the SUT to be able to use any form of Test Double.

Variations

Active Mock Objects comes in two flavours; most of the Mock Objects literature assumes an Eager Mock Object. If you are using ?Dynamic Test Double Generation? tools, the odds are they are implementing an Eager Mock Object. If you are using ?Static Test Double Generation? tools, you can look at the generated code to find out which variety is being generated. In either case, the stack trace of an assertion failure will also tell you. If the SUT appears on the stack as a caller of the *Active Mock Object*, you are using an Eager Mock Object

Eager Mock Object

An Eager Mock Object verifies each actual call against the expected call(s) as it is received and fails the test immediately. This allows debugging information from the size of the failure to be captured but results in a loss of "big picture" information This might include whether the calls are merely out of sequence; all you know is you received a call that was not immediately expected. Some mocking tools allow you to tell them how picky you want them to be about order of multiple calls to the same method.

Lazy Mock Object

It is possible to build a Lazy Mock Object which defers all the comparisons to the Final Verification Method. This addresses the issue of losing the "big picture" but at a cost: because the Final Verification Method is called from the test, not the SUT, you won't have access to the additional diagnostic information. In that sense, a Lazy Mock Object is really a cross between a Passive Mock Object that collects the indirect outputs and a ?Test Helper? that provides a set of useful Custom Assertions.

Motivating Example

The following test verifies the basic functionality of creating a flight. But it does not verify the indirect outputs of the SUT, namely, the SUT is expected to log each time a flight is created along with day/time and userid of the requester.

```
public void testCreateFlight() throws Exception {
    // setup
    FlightDto expectedFlightDto = helper.createAnonymousUnregisteredFlightDto();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();

    // exercise
    FlightDto actualFlightDto = facade.createFlight(
```



```
            expectedFlightDto.getAirline(), expectedFlightDto.getDepartureTime(),
            expectedFlightDto.getOriginAirportId(),
            expectedFlightDto.getArrivalTime(),
            expectedFlightDto.getDestinationAirportId());

      // validation
      assertFlightDtosEqual(expectedFlightDto, actualFlightDto);
   }
```

Refactoring Notes

Verification of indirect outputs can be added to existing tests merely by adding code to the fixture setup logic of your test to create the mock object, program it with values to return and the expected outputs, and install it. At the end of the test, add a call to the ?Final Verification Method.?

When using ?Hand-Coded Test Doubles? (or even ?Static Test Double Generation?), converting between eager and lazy *Active Mock Objects* is fairly straight-forward; just move the assertions from the dummy implementations of the methods to the final verification method or vice versa.

Solution Example

In this improved version of the test, mockLog is our *Active Mock Object*. The method setExpectedLogMessage is used to program it with the expected log message. The statement facade.setAuditLog(mockLog) installs the *Active Mock Object* using the ?Mock as Overridable Attribute? test double installation pattern. And finally, the verify() method ensures that the call to logMessage() was actually made at all.

```
   public void testRemoveFlight_activeMock() throws Exception {
      // fixture setup
      FlightDto expectedFlightDto = helper.createAnonymousRegisteredFlightDto();
      //      mock setup
      ActiveMockAuditLog mockLog = new ActiveMockAuditLog();
      mockLog.setExpectedLogMessage(
            helper.getTodaysDateWithoutTime(),
            helper.TEST_USER_NAME,
            helper.REMOVE_FLIGHT_ACTION_CODE,
            expectedFlightDto.getFlightNumber());
      mockLog.setExpectedNumberCalls(1);
      //      mock installation
      FlightManagementFacade facade = new FlightManagementFacadeImpl();
      facade.setAuditLog(mockLog);
      // exercise
      facade.removeFlight(expectedFlightDto.getFlightNumber());
      // verify
      assertFalse("flight still exists after being removed",
               facade.flightExists(expectedFlightDto.getFlightNumber()));
      mockLog.verify();
   }
```

All this was made possible through the use of the following *Active Mock Object*. This one was a ?Hand-Coded Test Double? and in the interest of space, we'll just show you what the logMessage method looks like:

```
   public void logMessage( Date actualDate, String actualUser,
                     String actualActionCode, Object actualDetail) {
      actualNumberCalls++;

      Assert.assertEquals("date", expectedDate, actualDate);
      Assert.assertEquals("user", expectedUser, actualUser);
      Assert.assertEquals("action code", expectedActionCode, actualActionCode);
      Assert.assertEquals("detail", expectedDetail, actualDetail);
   }
```

Note that the Assertion Methods are being called as static methods. In JUnit, this is required because the *Active Mock Object* is not a subclass of TestCase so it does not inherit the assertion methods from Assert. Other members

of the XUnit family may provide different mechanisms to access the Assertion Methods. Ruby, for example, provides them as ?mixins?.

Pattern Thumbnails

The following are the summary statements of the patterns I didn't have space to include here.

Final Verification Method

At the end of any test that uses an Active Mock Object, call a method that verifies that all the expected calls have been received.

Retrieval Interface

Provide an interface on a Passive Mock Object to retrieve information about which methods were called and what arguments were passed to them.

Hard-Coded Test Double

Build the Test Double by hard-coding the return values (and/or) expected calls.

Programmable Test Double

Configure a reusable Test Double with the values to be returned or verified during the ?fixture setup? phase of a test.

Programming Interface

`Provide a separate interface on the Programmable Test Double to tell it how to behave.`

Programming Mode

Have the methods of the Programmable Test Double do double-duty by learning what indirect outputs to expect when the methods are called during a special "programming mode".

Acknowledgements

I would like to thank all the clients and co-workers who shared the experiences which ultimately led to these patterns. Michael Stal provided useful comments while shepherding some related patterns for EuroPLoP 2004. I would especially like to thank my PLoP 2004 shepherd, Joe Yoder, for all the detailed comments he provided on these pattern.

References

[UTMJ] Johannes Link et al, "Unit Testing With Java : How Tests Drive the Code" Morgan Kaufmann April 2003

[McKinnon] Endo Testing

