Chapter 23

Test Double Patterns

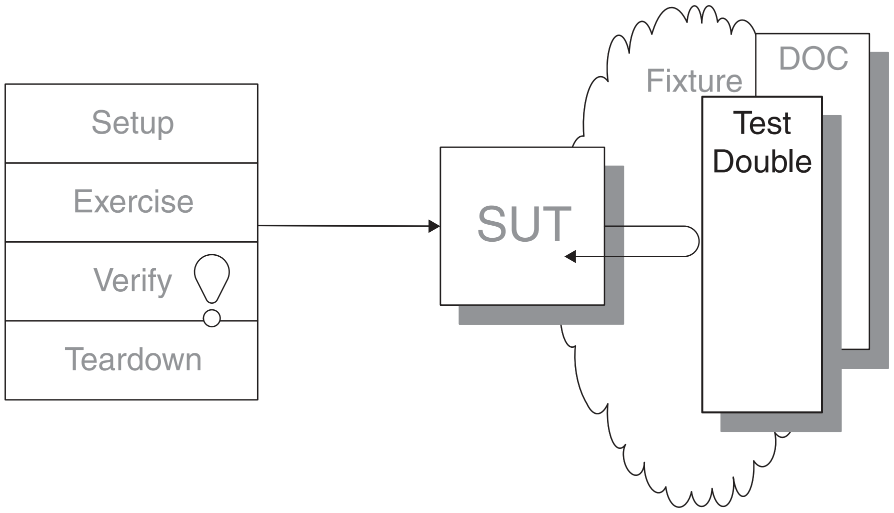
Patterns in This Chapter

Test Double

How can we verify logic independently when code it depends on is unusable?
How can we avoid Slow Tests?

Also known as: *Imposter*

We replace a component on which the SUT depends 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. Such a situation may arise because those components aren't available, because they will not return the results needed for the test, or because executing them would have undesirable side effects. In other cases, our test strategy requires us to have more control over or visibility of the internal behavior of the SUT.

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

How It Works

When the producers of a movie want 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. The stunt double may not be able to act, but he or she knows how to fall from great heights, crash a car, or do 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 the actor’s place.

For testing purposes, we can replace the real DOC (not the SUT!) with *our* equivalent of the “stunt double”: the *Test Double*. During the fixture setup phase of our *Four-Phase Test* (page 358), we replace the real DOC with our *Test Double*. Depending on the kind of test we are executing, we may hard-code the behavior of the *Test Double* or we may configure it 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 variation of *Test Double* we choose to use, we must keep in mind that we don’t need to implement the entire interface of the DOC. Instead, we provide only the functionality needed for our particular test. We can even build different *Test Doubles* for different tests that involve the same DOC.

When to Use It

We might want to use some sort of *Test Double* during our tests in the following circumstances:

- If we have an *Untested Requirement* (see *Production Bugs* on page 268) because neither the SUT nor its DOCs provide an observation point for the SUT’s indirect output that we need to verify using *Behavior Verification* (page 468)
- If we have *Untested Code* (see *Production Bugs*) and a DOC does not provide the control point to allow us to exercise the SUT with the necessary indirect inputs
- If we have *Slow Tests* (page 253) and we want to be able to run our tests more quickly and hence more often

Each of these scenarios can be addressed in some way by using a *Test Double*. Of course, we have to be careful when using *Test Doubles* because we are testing the SUT in a different configuration from the one that will be used in production. For this reason, we really should have at least one test that verifies the SUT works without a *Test Double*. We need to be careful that we don't replace the parts of the SUT that we are trying to verify because that practice can result in tests that test the wrong software! Also, excessive use of *Test Doubles* can result in *Fragile Tests* (page 239) as a result of *Overspecified Software*.

Test Doubles come in several major flavors, as summarized in Figure 23.1. The implementation variations of these patterns are described in more detail in the corresponding pattern write-ups.

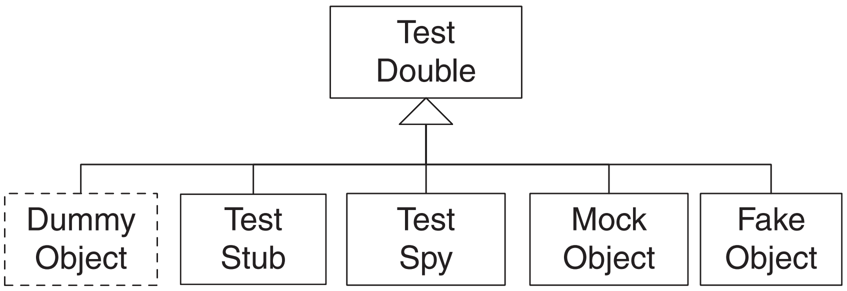


Figure 23.1 *Types of Test Doubles. Dummy Objects are really an alternative to the value patterns. Test Stubs are used to verify indirect inputs; Test Spies and Mock Objects are used to verify indirect outputs. Fake objects provide an alternative implementation.*

These variations are classified based on how/why we *use* the *Test Double*. We will deal with variations around how we *build* the *Test Doubles* in the "Implementation" section.

Variation: Test Stub

We use a *Test Stub* (page 529) to replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT. Its inclusion allows the test to force the SUT down paths it might not otherwise execute. We can further classify *Test Stubs* by the kind of indirect inputs they are used to inject into the SUT. A *Responder* (see *Test Stub*) injects valid values, while a *Saboteur* (see *Test Stub*) injects errors or exceptions.

Some people use the term "test stub" to mean a temporary implementation that is used only until the real object or procedure becomes available. I prefer to call this usage a *Temporary Test Stub* (see *Test Stub*) to avoid confusion.

Variation: Test Spy

We can use a more capable version of a *Test Stub*, the *Test Spy* (page 538), as an observation point for the indirect outputs of the SUT. Like a *Test Stub*, a *Test Spy* may need to provide values to the SUT in response to method calls. The *Test Spy*, however, also captures the indirect outputs of the SUT as it is exercised and saves them for *later* verification by the test. Thus, in many ways, the *Test Spy* is "just a" *Test Stub* with some recording capability. While a *Test Spy* is used for the same fundamental purpose as a *Mock Object* (page 544), the style of test we write using a *Test Spy* looks much more like a test written with a *Test Stub*.

Variation: Mock Object

We can use a *Mock Object* as an observation point to verify the indirect outputs of the SUT *as* it is exercised. Typically, the *Mock Object* also includes the functionality of a *Test Stub* in that it must return values to the SUT if it hasn't already failed the tests but the em*pha*sis[1] is on the verification of the indirect outputs. Therefore, a *Mock Object* is a lot more than just a *Test Stub* plus assertions: It is used in a fundamentally different way.

Variation: Fake Object

We use a *Fake Object* (page 551) to replace the functionality of a real DOC in a test for reasons other than verification of indirect inputs and outputs of the SUT. Typically, a *Fake Object* implements the same functionality as the real DOC but in a much simpler way. While a *Fake Object* is typically built specifically for testing, the test does not use it as either a control point or an observation point.

The most common reason for using a *Fake Object* is that the real DOC is not available yet, is too slow, or cannot be used in the test environment because of deleterious side effects. The sidebar "Faster Tests Without Shared Fixtures" (page 319) describes how we encapsulated all database access behind a persistence layer interface and then replaced the database with in-memory hash tables and made our tests run 50 times faster. Chapter 6, *Test Automation Strategy,* and Chapter 11, *Using Test Doubles,* provide an overview of the various techniques available for making our SUT easier to test.

[1] My mother grew up in Hungary and has retained a part of her Hungarian accent—think Zsa Zsa Gabor—all her life. She says, "It is important to put the em*pha*sis on the right syl*la*ble."

Variation: Dummy Object

Some method signatures of the SUT may require objects as parameters. If neither the test nor the SUT cares about these objects, we may choose to pass in a *Dummy Object* (page 728), which may be as simple as a null object reference, an instance of the `Object` class, or an instance of a *Pseudo-Object* (see *Hard-Coded Test Double* on page 568). In this sense, a *Dummy Object* isn't really a *Test Double* per se but rather an alternative to the value patterns *Literal Value* (page 714), *Derived Value* (page 718), and *Generated Value* (page 723).

Variation: Procedural Test Stub

A *Test Double* implemented in a procedural programming language is often called a "test stub," but I prefer to call it a *Procedural Test Stub* (see *Test Stub*) to distinguish this usage from the modern *Test Stub* variation of *Test Doubles*. Typically, we use a *Procedural Test Stub* to allow testing/debugging to proceed while waiting for other code to become available. It is rare for these objects to be "swapped in" at runtime but sometimes we make the code conditional on a "Debugging" flag—a form of *Test Logic in Production* (page 217).

Implementation Notes

Several considerations must be taken into account when we are building the *Test Double* (Figure 23.2):

- Whether the *Test Double* should be specific to a single test or reusable across many tests
- Whether the *Test Double* should exist in code or be generated on-the-fly
- How we tell the SUT to use the *Test Double* (installation)

The first and last points are addressed here. The discussion of *Test Double* generation is left to the section on *Configurable Test Doubles*.

Because the techniques for building *Test Doubles* are pretty much independent of their behavior (e.g., they apply to both *Test Stubs* and *Mock Objects*), I've chosen to split out the descriptions of the various ways we can build *Hard-Coded Test Doubles* and *Configurable Test Doubles* (page 558) into separate patterns.

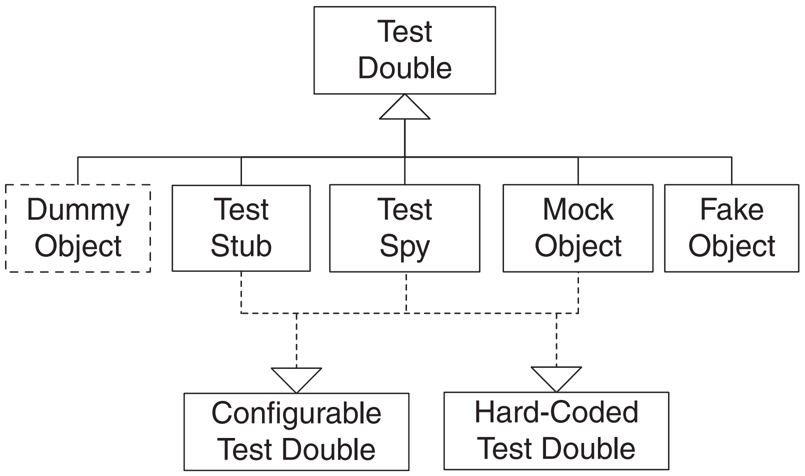


Figure 23.2 *Types of Test Doubles with implementation choices. Only Test Stubs, Test Spies, and Mock Objects need to be hard-coded or configured by the test. Dummy Objects have no implementation; Fake Objects are installed but not controlled by the test.*

Variation: Unconfigurable Test Doubles

Neither *Dummy Objects* nor *Fake Objects* need to be configured, each for their own reason. Dummies should never be used by the receiver so they need no "real" implementation. *Fake Objects*, by contrast, need a "real" implementation but one that is much simpler or "lighter" than the object that they replace. Therefore, neither the test nor the test automater will need to configure "canned" responses or expectations; we just install the *Test Double* and let the SUT use it as if it were real.

Variation: Hard-Coded Test Double

When we plan to use a specific *Test Double* in only a single test, it is often simplest to just hard-code the *Test Double* to return specific values (for *Test Stubs*) or expect specific method calls (*Mock Objects*). *Hard-Coded Test Doubles* are typically hand-built by the test automater. They come in several forms, including the *Self Shunt* (see *Hard-Coded Test Double*), where the *Testcase Class* (page 373) acts as the *Test Double*; the *Anonymous Inner Test Double* (see *Hard-Coded Test Double*), where language features are used to create the *Test Double* inside the *Test Method* (page 348); and the *Test Double* implemented as separate *Test Double Class* (see *Hard-Coded Test Double*). Each of these options is discussed in more detail in *Hard-Coded Test Double*.

Variation: Configurable Test Double

When we want to use the same *Test Double* implementation in many tests, we will typically prefer to use a *Configurable Test Double*. Although the test automater can manually build these objects, many members of the xUnit family have reusable toolkits available for generating *Configurable Test Doubles*.

Installing the Test Double

Before we can exercise the SUT, we must tell it to use the *Test Double* instead of the object that the *Test Double* replaces. We can use any of the **substitutable dependency** patterns to install the *Test Double* during the fixture setup phase of our *Four-Phase Test*. *Configurable Test Doubles* need to be configured before we exercise the SUT, and we typically perform this configuration before we install them.

Example: Test Double

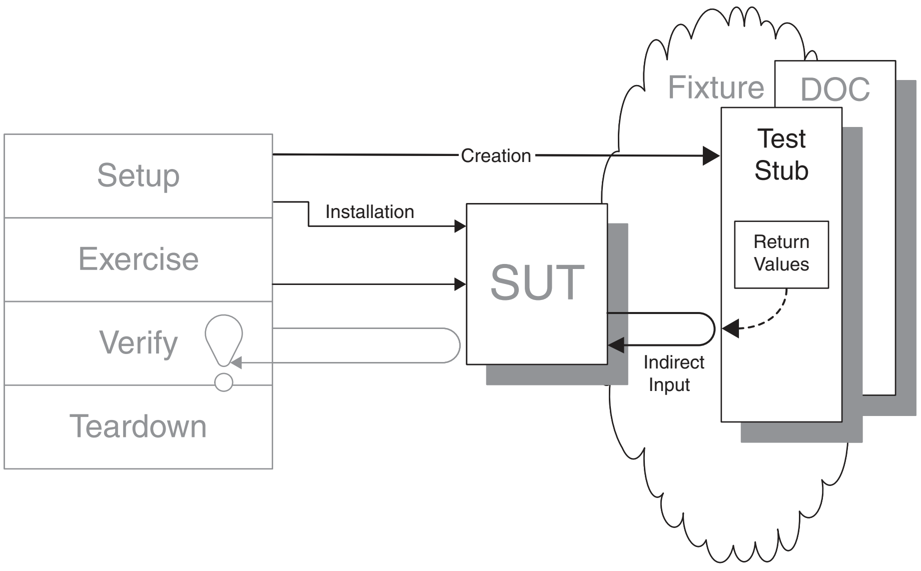
Because there are a wide variety of reasons for using the variations of *Test Doubles*, it is difficult to provide a single example that characterizes the motivation behind each style. Please refer to the examples in each of the more detailed patterns referenced earlier.

Test Stub

How can we verify logic independently when it depends on indirect inputs from other software components?

We replace a real object with a test-specific object that feeds the desired indirect inputs into the system under test.

Also known as: *Stub*



In many circumstances, the environment or context in which the SUT operates very much influences the behavior of the SUT. To get adequate control over the indirect inputs of the SUT, we may have to replace some of the context with something we can control—namely, a *Test Stub*.

How It Works

First, we define a test-specific implementation of an interface on which the SUT depends. This implementation is configured to respond to calls from the SUT with the values (or exceptions) that will exercise the *Untested Code* (see *Production Bugs* on page 268) within the SUT. Before exercising the SUT, we install the *Test Stub* so that the SUT uses it *instead of* the real implementation. When called by the SUT during test execution, the *Test Stub* returns the previously defined values. The test can then verify the expected outcome in the normal way.

When to Use It

A key indication for using a *Test Stub* is having *Untested Code* caused by our inability to control the indirect inputs of the SUT. We can use a *Test Stub* as a control point that allows us to control the behavior of the SUT with various indirect inputs and we have no need to verify the indirect outputs. We can also use a *Test Stub* to inject values that allow us to get past a particular point in the software where the SUT calls software that is unavailable in our test environment.

If we do need an observation point that allows us to verify the indirect outputs of the SUT, we should consider using a *Mock Object* (page 544) or a *Test Spy* (page 538). Of course, we must have a way of installing a *Test Double* (page 522) into the SUT to be able to use any form of *Test Double*.

Variation: Responder

A *Test Stub* that is used to inject valid indirect inputs into the SUT so that it can go about its business is called a *Responder. Responders* are commonly used in "happy path" testing when the real component is uncontrollable, is not yet available, or is unusable in the development environment. The tests will invariably be *Simple Success Tests* (see *Test Method* on page 348).

Variation: Saboteur

A *Test Stub* that is used to inject invalid indirect inputs into the SUT is often called a *Saboteur* because its purpose is to derail whatever the SUT is trying to do so that we can see how the SUT copes under these circumstances. The "derailment" might be caused by returning unexpected values or objects, or it might result from raising an exception or causing a runtime error. Each test may be either a *Simple Success Test* or an *Expected Exception Test* (see *Test Method*), depending on how the SUT is expected to behave in response to the indirect input.

Variation: Temporary Test Stub

A *Temporary Test Stub* stands in for a DOC that is not yet available. This kind of *Test Stub* typically consists of an empty shell of a real class with hard-coded return statements. As soon as the real DOC is available, it replaces the *Temporary Test Stub*. Test-driven development often requires us to create *Temporary*

Test Stubs as we write code from the outside in; these shells evolve into the real classes as we add code to them. In need-driven development, we tend to use *Mock Objects* because we want to verify that the SUT calls the right methods on the *Temporary Test Stub*; in addition, we typically continue using the *Mock Object* even after the real DOC becomes available.

Variation: Procedural Test Stub

A *Procedural Test Stub* is a *Test Stub* written in a procedural programming language. It is particularly challenging to create in procedural programming languages that do not support procedure variables (also known as **function pointers**). In most cases, we must put `if testing then` hooks into the production code (a form of *Test Logic in Production;* see page 217).

Variation: Entity Chain Snipping

Entity Chain Snipping (see *Test Stub* on page 529) is a special case of a *Responder* that is used to replace a complex network of objects with a single *Test Stub* that pretends to be the network of objects. Its inclusion can make fixture setup go much more quickly (especially when the objects would normally have to be persisted into a database) and can make the tests much easier to understand.

Implementation Notes

We must be careful when using *Test Stubs* because we are testing the SUT in a different configuration from the one that will be used in production. We really should have at least one test that verifies the SUT works without a *Test Stub*. A common mistake made by test automaters who are new to stubs is to replace a part of the SUT that they are trying to test. For this reason, it is important to be really clear about what is playing the role of SUT and what is playing the role of test fixture. Also, note that excessive use of *Test Stubs* can result in *Overspecified Software* (see *Fragile Test* on page 239).

Test Stubs may be built in several different ways depending on our specific needs and the tools we have on hand.

Variation: Hard-Coded Test Stub

A *Hard-Coded Test Stub* has its responses hard-coded within its program logic. These *Test Stubs* tend to be purpose-built for a single test or a very small number of tests. See *Hard-Coded Test Double* (page 568) for more information.

Variation: Configurable Test Stub

When we want to avoid building a different *Hard-Coded Test Stub* for each test, we can use a *Configurable Test Stub* (see *Configurable Test Double* on page 558). A test configures the *Configurable Test Stub* as part of its fixture setup phase. Many members of the xUnit family offer tools with which to generate *Configurable Test Doubles* (page 558), including *Configurable Test Stubs*.

Motivating Example

The following test verifies the basic functionality of a component that formats an HTML string containing the current time. Unfortunately, it depends on the real system clock so it rarely ever passes!

```
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);
}
```

We could try to address this problem by making the test calculate the expected results based on the current system time as follows:

```
public void testDisplayCurrentTime_whenever() {
   // fixture setup
   TimeDisplay sut = new TimeDisplay();
   // exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // verify outcome
   Calendar time = new DefaultTimeProvider().getTime();
   StringBuffer expectedTime = new StringBuffer();
   expectedTime.append("<span class=\"tinyBoldText\">");

   if ((time.get(Calendar.HOUR_OF_DAY) == 0)
        && (time.get(Calendar.MINUTE) <= 1)) {
      expectedTime.append( "Midnight");
   } else if ((time.get(Calendar.HOUR_OF_DAY) == 12)
               && (time.get(Calendar.MINUTE) == 0)) { // noon
      expectedTime.append("N3oon");
   } else  {
      SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
      expectedTime.append(fr.format(time.getTime()));
   }
```

```
      expectedTime.append("</span>");

      assertEquals( expectedTime, result);
   }
```

This *Flexible Test* (see *Conditional Test Logic* on page 200) introduces two problems. First, some test conditions are never exercised. (Do *you* want to come in to work to run the tests at midnight to prove the software works at midnight?) Second, the test needs to duplicate much of the logic in the SUT to calculate the expected results. How do we prove the logic is actually correct?

Refactoring Notes

We can achieve proper verification of the indirect inputs by getting control of the time. To do so, we use the Replace Dependency with Test Double (page 522) refactoring to replace the real system clock (represented here by `TimeProvider`) with a Virtual Clock [VCTP]. We then implement it as a *Test Stub* that is configured by the test with the time we want to use as the indirect input to the SUT.

Example: Responder (as Hand-Coded Test Stub)

The following test verifies one of the happy path test conditions using a *Responder* to get control over the indirect inputs of the SUT. Based on the time injected into the SUT, the expected result can be hard-coded safely.

```
   public void testDisplayCurrentTime_AtMidnight()
               throws Exception {
      // Fixture setup
      //      Test Double configuration
      TimeProviderTestStub tpStub = new TimeProviderTestStub();
      tpStub.setHours(0);
      tpStub.setMinutes(0);
      //   Instantiate SUT
      TimeDisplay sut = new TimeDisplay();
      //      Test Double installation
      sut.setTimeProvider(tpStub);
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // Verify outcome
      String expectedTimeString =
              "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }
```

This test makes use of the following hand-coded configurable *Test Stub* implementation:

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

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

// Configuration interface
public void setHours(int hours) {
   // 0 is midnight; 12 is noon
   myTime.set(Calendar.HOUR_OF_DAY, hours);
}

public void setMinutes(int minutes) {
   myTime.set(Calendar.MINUTE, minutes);
}
// Interface used by SUT
public Calendar getTime() {
   // @return the last time that was set
   return myTime;
}
```

Example: Responder (Dynamically Generated)

Here's the same test coded using the JMock *Configurable Test Double* framework:

```
public void testDisplayCurrentTime_AtMidnight_JM()
      throws Exception {
   // Fixture setup
   TimeDisplay sut = new TimeDisplay();
   //  Test Double configuration
   Mock tpStub = mock(TimeProvider.class);
   Calendar midnight = makeTime(0,0);
   tpStub.stubs().method("getTime").
                  withNoArguments().
                  will(returnValue(midnight));
   //  Test Double installation
   sut.setTimeProvider((TimeProvider) tpStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify outcome
   String expectedTimeString =
           "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```

There is no *Test Stub* implementation to examine for this test because the JMock framework implements the *Test Stub* using reflection. Thus we had to write a *Test Utility Method* (page 599) called `makeTime` that contains the logic to construct the `Calendar` object to be returned. In the hand-coded *Test Stub*, this logic appeared inside the `getTime` method.

Example: Saboteur (as Anonymous Inner Class)

The following test uses a *Saboteur* to inject invalid indirect inputs into the SUT so we can see how the SUT copes under these circumstances.

```
   public void testDisplayCurrentTime_exception()
         throws Exception {
      // Fixture setup
      //   Define and instantiate Test Stub
      TimeProvider testStub = new TimeProvider()
         { // Anonymous inner Test Stub
            public Calendar getTime() throws TimeProviderEx {
               throw new TimeProviderEx("Sample");
         }
      };
      //   Instantiate SUT
      TimeDisplay sut = new TimeDisplay();
      sut.setTimeProvider(testStub);
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // Verify direct output
      String expectedTimeString =
            "<span class=\"error\">Invalid Time</span>";
      assertEquals("Exception", expectedTimeString, result);
   }
```

In this case, we used an *Inner Test Double* (see *Hard-Coded Test Double*) to throw an exception that we expect the SUT to handle gracefully. One interesting thing about this test is that it uses the *Simple Success Test* method template rather than the *Expected Exception Test* template, even though we are injecting an exception as the indirect input. The rationale behind this choice is that we are expecting the SUT to catch the exception and change the string formatting; we are not expecting the SUT to throw an exception.

Example: Entity Chain Snipping

In this example, we are testing the `Invoice` but require a `Customer` to instantiate the `Invoice`. The `Customer` requires an `Address`, which in turn requires a `City`. Thus we find ourselves creating numerous additional objects just to set up the fixture. Suppose the behavior of the invoice depends on some attribute of the `Customer` that is calculated from the `Address` by calling the method `get_zone` on the `Customer`.

```
public void testInvoice_addLineItem_noECS() {
   final int QUANTITY = 1;
   Product product = new Product(getUniqueNumberAsString(),
                                 getUniqueNumber());
   State state = new State("West Dakota", "WD");
   City city = new City("Centreville", state);
   Address address = new Address("123 Blake St.", city, "12345");
   Customer customer= new Customer(getUniqueNumberAsString(),
                                   getUniqueNumberAsString(),
                                   address);
   Invoice inv = new Invoice(customer);
   // Exercise
   inv.addItemQuantity(product, QUANTITY);
   // Verify
   List lineItems = inv.getLineItems();
   assertEquals("number of items", lineItems.size(), 1);
   LineItem actual = (LineItem)lineItems.get(0);
   LineItem expItem = new LineItem(inv, product, QUANTITY);
   assertLineItemsEqual("",expItem, actual);
}
```

In this test, we want to verify only the behavior of the invoice logic that depends on this `zone` attribute—not the way this attribute is calculated from the `Customer`'s address. (There are separate `Customer` unit tests to verify the `zone` is calculated correctly.) All of the setup of the address, city, and other information merely distracts the reader.

Here's the same test using a *Test Stub* instead of the `Customer`. Note how much simpler the fixture setup has become as a result of *Entity Chain Snipping*!

```
public void testInvoice_addLineItem_ECS() {
   final int QUANTITY = 1;
   Product product = new Product(getUniqueNumberAsString(),
                                 getUniqueNumber());
   Mock customerStub = mock(ICustomer.class);
   customerStub.stubs().method("getZone").will(returnValue(ZONE_3));
   Invoice inv = new Invoice((ICustomer)customerStub.proxy());
   // Exercise
   inv.addItemQuantity(product, QUANTITY);
   // Verify
```

```
      List lineItems = inv.getLineItems();
      assertEquals("number of items", lineItems.size(), 1);
      LineItem actual = (LineItem)lineItems.get(0);
      LineItem expItem = new LineItem(inv, product, QUANTITY);
      assertLineItemsEqual("", expItem, actual);
   }
```

We have used JMock to stub out the `Customer` with a `customerStub` that returns `ZONE_3` when `getZone` is called. This is all we need to verify the `Invoice` behavior, and we have managed to get rid of all that distracting extra object construction. It is also much clearer from reading this test that invoicing behavior depends only on the value returned by `get_zone` and not any other attributes of the `Customer` or `Address`.

Further Reading

Almost every book on automated testing using xUnit has something to say about *Test Stubs*, so I won't list those resources here. As you are reading other books, however, keep in mind that the term *Test Stub* is often used to refer to a *Mock Object*. *Mocks, Fakes, Stubs, and Dummies* (in Appendix B) contains a more thorough comparison of the terminology used in various books and articles.

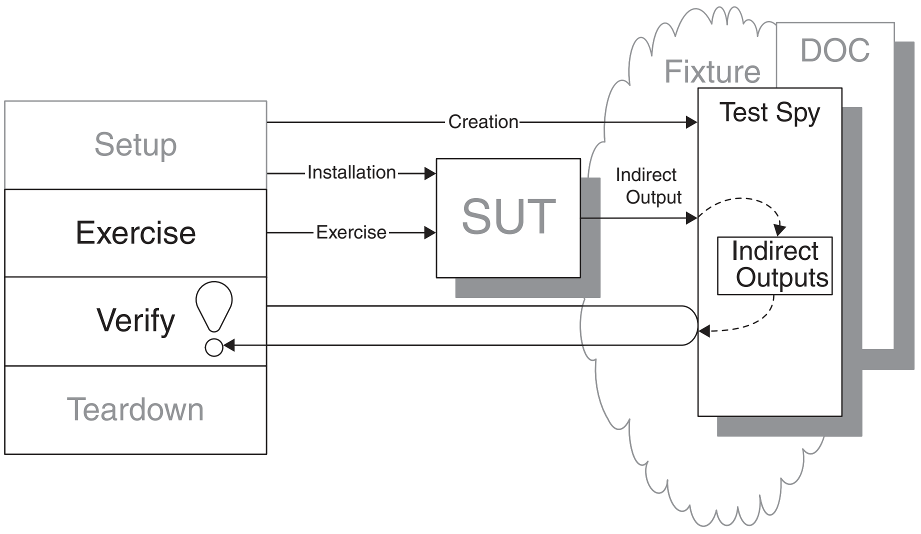
Sven Gorts describes a number of different ways we can use a *Test Stub* [UTwHCM]. I have adopted many of his names and adapted a few to better fit into this pattern language. Paolo Perrotta wrote a pattern describing a common example of a *Responder* called Virtual Clock. He uses a *Test Stub* as a Decorator [GOF] for the real system clock that allows the time to be "frozen" or resumed. Of course, we could use a *Hard-Coded Test Stub* or a *Configurable Test Stub* just as easily for most tests.

Test Spy

How do we implement Behavior Verification?
How can we verify logic independently when it has indirect outputs to other software components?

Also known as: *Spy, Recording Test Stub*

We 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 behavior of the SUT. To get adequate 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 *Test Spy* is a simple and intuitive way to implement *Behavior Verification* (page 468) via an observation point that exposes the indirect outputs of the SUT so they can be verified.

How It Works

Before we exercise the SUT, we install a *Test Spy* as a stand-in for a DOC used by the SUT. The *Test Spy* is designed to act as an observation point by recording the method calls made to it by the SUT as it is exercised. During the result verification phase, the test compares the actual values passed to the *Test Spy* by the SUT with the values expected by the test.

When to Use It

A key indication for using a *Test Spy* is having an *Untested Requirement* (see *Production Bugs* on page 268) caused by an inability to observe the side effects of invoking methods on the SUT. *Test Spies* are a natural and intuitive way to extend the existing tests to cover these indirect outputs because the calls to the *Assertion Methods* (page 362) are invoked by the test after the SUT has been exercised just like in "normal" tests. The *Test Spy* merely acts as the observation point that gives the *Test Method* (page 348) access to the values recorded during the SUT execution.

We should use a *Test Spy* in the following circumstances:

- We are verifying the indirect outputs of the SUT and we *cannot* predict the values of all attributes of the interactions with the SUT ahead of time.
- We want the assertions to be visible in the test and we don't think the way in which the *Mock Object* (page 544) expectations are established is sufficiently intent-revealing.
- Our test requires test-specific equality (so we cannot use the standard definition of equality as implemented in the SUT) *and* we are using tools that generate the *Mock Object* but do not give us control over the *Assertion Methods* being called.
- A failed assertion cannot be reported effectively back to the *Test Runner* (page 377). This might occur if the SUT is running inside a container that catches all exceptions and makes it difficult to report the results or if the logic of the SUT runs in a different thread or process from the test that invokes it. (Both of these cases really beg refactoring to allow us to test the SUT logic directly, but that is the subject of another chapter.)
- We would like to have access to all the outgoing calls of the SUT before making any assertions on them.

If none of these criteria apply, we may want to consider using a *Mock Object*. If we are trying to address *Untested Code* (see *Production Bugs*) by controlling the indirect inputs of the SUT, a simple *Test Stub* (page 529) may be all we need.

Unlike a *Mock Object*, a *Test Spy* does not fail the test at the first deviation from the expected behavior. Thus our tests will be able to include more detailed diagnostic information in the *Assertion Message* (page 370) based on information gathered after a *Mock Object* would have failed the test. At the point of test failure, however, only the information within the *Test Method* itself is available to be used in the calls to the *Assertion Methods*. If we need to include information that is accessible only while the SUT is being exercised, either we must explicitly capture it within our *Test Spy* or we must use a *Mock Object*.

Of course, we won't be able to use any *Test Doubles* (page 522) unless the SUT implements some form of substitutable dependency.

Implementation Notes

The *Test Spy* itself can be built as a *Hard-Coded Test Double* (page 568) or as a *Configurable Test Double* (page 558). Because detailed examples appear in the discussion of those patterns, only a quick summary is provided here. Likewise, we can use any of the substitutable dependency patterns to install the *Test Spy* *before* we exercise the SUT.

The key characteristic in how a test uses a *Test Spy* relates to the fact that assertions are made from within the *Test Method*. Therefore, the test must recover the indirect outputs captured by the *Test Spy* before it can make its assertions, which can be done in several ways.

Variation: Retrieval Interface

We can define the *Test Spy* as a separate class with a *Retrieval Interface* that exposes the recorded information. The *Test Method* installs the *Test Spy* instead of the normal DOC as part of the fixture setup phase of the test. After the test has exercised the SUT, it uses the *Retrieval Interface* to retrieve the actual indirect outputs of the SUT from the *Test Spy* and then calls *Assertion Methods* with those outputs as arguments.

Also known as: *Loopback*

Variation: Self Shunt

We can collapse the *Test Spy* and the *Testcase Class* (page 373) into a single object called a *Self Shunt*. The *Test Method* installs itself, the *Testcase Object* (page 382), as the DOC into the SUT. Whenever the SUT delegates to the DOC, it is actually calling methods on the *Testcase Object,* which implements the methods by saving the actual values into instance variables that can be accessed by the *Test Method*. The methods could also make assertions in the *Test Spy* methods, in which case the *Self Shunt* is a variation on a *Mock Object* rather than a *Test Spy*. In statically typed languages, the *Testcase Class* must implement the outgoing interface (the observation point) on which the SUT depends so that the *Testcase Class is* type-compatible with the variables that are used to hold the DOC.

Variation: Inner Test Double

A popular way to implement the *Test Spy* as a *Hard-Coded Test Double* is to code it as an **anonymous inner class** or **block closure** within the *Test Method* and to have this class or **block** save the actual values into instance or local variables that are accessible by the *Test Method*. This variation is really another way to implement a *Self Shunt* (see *Hard-Coded Test Double*).

Variation: Indirect Output Registry

Yet another possibility is to have the *Test Spy* store the actual parameters in a well-known place where the *Test Method* can access them. For example, the *Test Spy* could save those values in a file or in a Registry [PEAA] object.

Motivating Example

The following test verifies the basic functionality of removing a flight but does not verify the indirect outputs of the SUT—namely, the fact that the SUT is expected to log each time a flight is removed along with the date/time and username of the requester.

```
public void testRemoveFlight() throws Exception {
   // setup
   FlightDto expectedFlightDto = createARegisteredFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight should not exist after being removed",
               facade.flightExists( expectedFlightDto.
                                           getFlightNumber()));
}
```

Refactoring Notes

We can add verification of indirect outputs to existing tests using a Replace Dependency with Test Double (page 522) refactoring. It involves adding code to the fixture setup logic of the tests to create the *Test Spy*, configuring the *Test Spy* with any values it needs to return, and installing it. At the end of the test, we add assertions comparing the expected method names and arguments of the indirect outputs with the actual values retrieved from the *Test Spy* using the *Retrieval Interface*.

Example: Test Spy

In this improved version of the test, `logSpy` is our *Test Spy*. The statement `facade.setAuditLog(logSpy)` installs the *Test Spy* using the *Setter Injection* pattern (see *Dependency Injection* on page 678). The methods `getDate`, `getActionCode`, and so on are the *Retrieval Interface* used to access the actual arguments of the call to the logger.

```
public void testRemoveFlightLogging_recordingTestStub()
         throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnUnregFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   //    Test Double setup
   AuditLogSpy logSpy = new AuditLogSpy();
   facade.setAuditLog(logSpy);
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight still exists after being removed",
               facade.flightExists( expectedFlightDto.
                                          getFlightNumber()));
   assertEquals("number of calls", 1,
                logSpy.getNumberOfCalls());
   assertEquals("action code",
                Helper.REMOVE_FLIGHT_ACTION_CODE,
                logSpy.getActionCode());
   assertEquals("date", helper.getTodaysDateWithoutTime(),
                logSpy.getDate());
   assertEquals("user", Helper.TEST_USER_NAME,
                logSpy.getUser());
   assertEquals("detail",
                expectedFlightDto.getFlightNumber(),
                logSpy.getDetail());
}
```

This test depends on the following definition of the *Test Spy*:

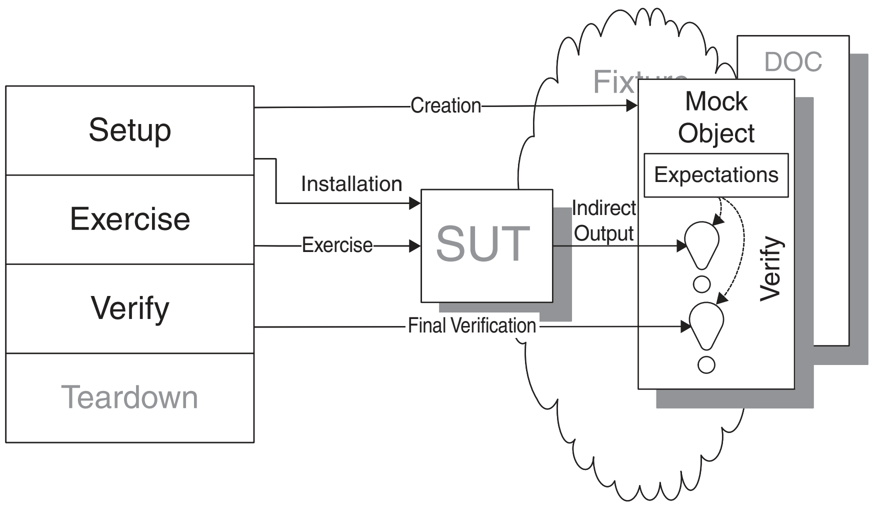
```
public class AuditLogSpy implements AuditLog {
   // Fields into which we record actual usage information
   private Date date;
   private String user;
   private String actionCode;
   private Object detail;
   private int numberOfCalls = 0;
   // Recording implementation of real AuditLog interface
   public void logMessage(Date date,
                          String user,
                          String actionCode,
                          Object detail) {
      this.date = date;
      this.user = user;
      this.actionCode = actionCode;
      this.detail = detail;

      numberOfCalls++;
   }

   // Retrieval Interface
   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;
   }
}
```

Of course, we could have implemented the *Retrieval Interface* by making the various fields of our spy public and thereby avoided the need for accessor methods. Please refer to the examples in *Hard-Coded Test Double* for other implementation options.

Mock Object

How do we implement Behavior Verification for indirect outputs of the SUT?
How can we verify logic independently when it depends on indirect inputs from other software components?

We replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.



In many circumstances, the environment or context in which the SUT operates very much influences the behavior of the SUT. In other cases, we must peer "inside"[2] the SUT to determine whether the expected behavior has occurred.

A *Mock Object* is a powerful way to implement *Behavior Verification* (page 468) while avoiding *Test Code Duplication* (page 213) between similar tests. It works by delegating the job of verifying the indirect outputs of the SUT entirely to a *Test Double* (page 522).

[2] Technically, the SUT is whatever software we are testing and doesn't include anything it depends on; thus "inside" is somewhat of a misnomer. It is better to think of the DOC that is the destination of the indirect outputs as being "behind" the SUT and part of the fixture.

How It Works

First, we define a *Mock Object* that implements the same interface as an object on which the SUT depends. Then, during the test, we configure the *Mock Object* with the values with which it should respond to the SUT *and* the method calls (complete with expected arguments) it should expect from the SUT. Before exercising the SUT, we install the *Mock Object* so that the SUT uses it *instead of* the real implementation. When called during SUT execution, the *Mock Object* compares the actual arguments received with the expected arguments using *Equality Assertions* (see *Assertion Method* on page 362) and fails the test if they don't match. The test need not make any assertions at all!

When to Use It

We can use a *Mock Object* as an observation point when we need to do *Behavior Verification* to avoid having an *Untested Requirement* (see *Production Bugs* on page 268) caused by our inability to observe the side effects of invoking methods on the SUT. This pattern is commonly used during endoscopic testing [ET] or need-driven development [MRNO]. Although we don't need to use a *Mock Object* when we are doing *State Verification* (page 462), we might use a *Test Stub* (page 529) or *Fake Object* (page 551). Note that test drivers have found other uses for the *Mock Object toolkits,* but many of these are actually examples of using a *Test Stub* rather than a *Mock Object.*

To use a *Mock Object*, we must be able to predict the values of most or all arguments of the method calls *before* we exercise the SUT. We should not use a *Mock Object* if a failed assertion cannot be reported back to the *Test Runner* (page 377) effectively. This may be the case if the SUT runs inside a container that catches and eats all exceptions. In these circumstances, we may be better off using a *Test Spy* (page 538) instead.

Mock Objects (especially those created using dynamic mocking tools) often use the `equals` methods of the various objects being compared. If our test-specific equality differs from how the SUT would interpret `equals`, we may not be able to use a *Mock Object* or we may be forced to add an `equals` method where we didn't need one. This smell is called *Equality Pollution* (see *Test Logic in Production* on page 217). Some implementations of *Mock Objects* avoid this problem by allowing us to specify the "comparator" to be used in the *Equality Assertions*.

Mock Objects can be either "strict" or "lenient" (sometimes called "nice"). A "strict" *Mock Object* fails the test if the calls are received in a different order than was specified when the *Mock Object* was programmed. A "lenient" *Mock Object* tolerates out-of-order calls.

Implementation Notes

Tests written using *Mock Objects* look different from more traditional tests because all the expected behavior must be specified *before* the SUT is exercised. This makes the tests harder to write and to understand for test automation neophytes. This factor may be enough to cause us to prefer writing our tests using *Test Spies*.

The standard *Four-Phase Test* (page 358) is altered somewhat when we use *Mock Objects*. In particular, the fixture setup phase of the test is broken down into three specific activities and the result verification phase more or less disappears, except for the possible presence of a call to the "final verification" method at the end of the test.

Fixture setup:

- Test constructs *Mock Object*.
- Test configures *Mock Object*. This step is omitted for *Hard-Coded Test Doubles* (page 568).
- Test installs *Mock Object* into SUT.

Exercise SUT:

- SUT calls *Mock Object; Mock Object* does assertions.

Result verification:

- Test calls "final verification" method.

Fixture teardown:

- No impact.

Let's examine these differences a bit more closely:

Construction

As part of the fixture setup phase of our *Four-Phase Test*, we must construct the *Mock Object* that we will use to replace the substitutable dependency. Depending on which tools are available in our programming language, we can either build the *Mock Object* class manually, use a code generator to create a *Mock Object* class, or use a dynamically generated *Mock Object*.

Configuration with Expected Values

Because the *Mock Object* toolkits available in many members of the xUnit family typically create *Configurable Mock Objects* (page 544), we need to configure the *Mock Object* with the expected method calls (and their parameters) as well as the values to be returned by any functions. (Some *Mock Object* frameworks allow us to disable verification of the method calls or just their parameters.) We typically perform this configuration before we install the *Test Double*.

This step is not needed when we are using a *Hard-Coded Test Double* such as an *Inner Test Double* (see *Hard-Coded Test Double*).

Installation

Of course, we must have a way of installing a *Test Double* into the SUT to be able to use a *Mock Object*. We can use whichever substitutable dependency pattern the SUT supports. A common approach in the test-driven development community is *Dependency Injection* (page 678); more traditional developers may favor *Dependency Lookup* (page 686).

Usage

When the SUT calls the methods of the *Mock Object*, these methods compare the method call (method name plus arguments) with the expectations. If the method call is unexpected or the arguments are incorrect, the assertion fails the test immediately. If the call is expected but came out of sequence, a strict *Mock Object* fails the test immediately; by contrast, a lenient *Mock Object* notes that the call was received and carries on. Missed calls are detected when the final verification method is called.

If the method call has any outgoing parameters or return values, the *Mock Object* needs to return or update something to allow the SUT to continue executing the test scenario. This behavior may be either hard-coded or configured at the same time as the expectations. This behavior is the same as for *Test Stubs,* except that we typically return happy path values.

Final Verification

Most of the result verification occurs inside the *Mock Object* as it is called by the SUT. The *Mock Object* will fail the test if the methods are called with the wrong arguments or if methods are called unexpectedly. But what happens if the expected method calls are never received by the *Mock Object*? The *Mock Object* may have trouble detecting that the test is over and it is time to check for unfulfilled expectations. Therefore, we need to ensure that the final verification method is called. Some *Mock Object* toolkits have found a way to invoke this method automatically by including the call in the `tearDown` method.[3] Many other toolkits require us to remember to call the final verification method ourselves.

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 the date/time and username of the requester.

```
public void testRemoveFlight() throws Exception {
   // setup
   FlightDto expectedFlightDto = createARegisteredFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight should not exist after being removed",
               facade.flightExists( expectedFlightDto.
                                          getFlightNumber()));
}
```

Refactoring Notes

Verification of indirect outputs can be added to existing tests by using a Replace Dependency with Test Double (page 522) refactoring. This involves adding code to the fixture setup logic of our test to create the *Mock Object*; configuring the *Mock Object* with the expected method calls, arguments, and values to be returned; and installing it using whatever substitutable dependency mechanism is provided by the SUT. At the end of the test, we add a call to the final verification method if our *Mock Object* framework requires one.

Example: Mock Object (Hand-Coded)

In this improved version of the test, `mockLog` is our *Mock Object*. The method `setExpectedLogMessage` is used to program it with the expected log message. The statement `facade.setAuditLog(mockLog)` installs the *Mock Object* using the *Setter Injection* (see *Dependency Injection*) test double-installation pattern. Finally, the `verify()` method ensures that the call to `logMessage()` was actually made.

[3] This usually requires that we subclass our testcase from a special MockObjectTestCase class.

```
public void testRemoveFlight_Mock() throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnonRegFlight();
   // mock configuration
   ConfigurableMockAuditLog mockLog =
      new ConfigurableMockAuditLog();
   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();
}
```

This approach was made possible by use of the following *Mock Object*. Here we have chosen to use a hand-built *Mock Object*. In the interest of space, just the `logMessage` method is shown:

```
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);
}
```

The *Assertion Methods* are called as static methods. In JUnit, this approach is required because the *Mock Object* is not a subclass of `TestCase`; thus it does not inherit the assertion methods from `Assert`. Other members of the xUnit family may provide different mechanisms to access the *Assertion Methods*. For example, NUnit provides them *only* as static methods on the `Assert` class, so even *Test Methods* (page 348) need to access the *Assertion Methods* this way. Test::Unit, the xUnit family member for the Ruby programming language, provides them as **mixins;** as a consequence, they can be called in the normal fashion.

Example: Mock Object (Dynamically Generated)

The last example used a hand-coded *Mock Object*. Most members of the xUnit family, however, have dynamic *Mock Object* frameworks available. Here's the same test rewritten using JMock:

```
public void testRemoveFlight_JMock() throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnonRegFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   // mock configuration
   Mock mockLog = mock(AuditLog.class);
   mockLog.expects(once()).method("logMessage")
            .with(eq(helper.getTodaysDateWithoutTime()),
                  eq(Helper.TEST_USER_NAME),
                  eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
                  eq(expectedFlightDto.getFlightNumber()));
   // mock installation
   facade.setAuditLog((AuditLog) mockLog.proxy());
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight still exists after being removed",
               facade.flightExists( expectedFlightDto.
                                       getFlightNumber()));
   // verify() method called automatically by JMock
}
```

Note how JMock provides a "fluent" *Configuration Interface* (see *Configurable Test Double*) that allows us to specify the expected method calls in a fairly readable fashion. JMock also allows us to specify the comparator to be used by the assertions; in this case, the calls to `eq` cause the default `equals` method to be called.

Further Reading

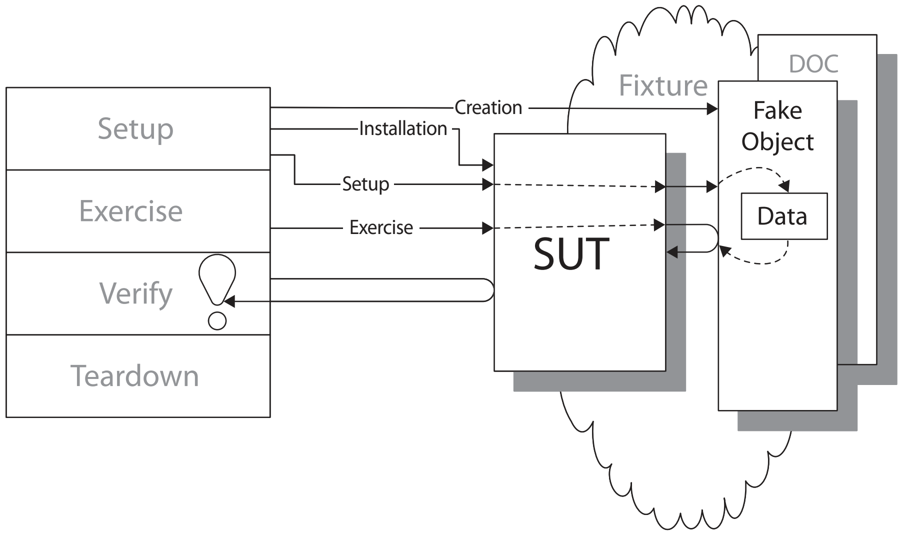
Almost every book on automated testing using xUnit has something to say about *Mock Objects*, so I won't list those resources here. As you are reading other books, keep in mind that the term *Mock Object* is often used to refer to a *Test Stub* and sometimes even to *Fake Objects*. *Mocks, Fakes, Stubs, and Dummies* (in Appendix B) contains a more thorough comparison of the terminology used in various books and articles.

Fake Object

How can we verify logic independently when depended-on objects cannot be used?
How can we avoid Slow Tests?

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

Also known as:
Dummy



The SUT often depends on other components or systems. Although the interactions with these other components may be necessary, the side effects of these interactions *as implemented by the real DOC* may be unnecessary or even detrimental.

A *Fake 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

We acquire or build a very lightweight implementation of the same functionality as provided by a component on which the SUT depends 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 provide only the equivalent services to the SUT so that the SUT remains unaware it isn't using the real DOC.

A *Fake Object* is a kind of *Test Double* (page 522) that is similar to a *Test Stub* (page 529) in many ways, including the need to install into the SUT a substitutable dependency. Whereas a *Test Stub* acts as a control point to inject indirect inputs into the SUT, however, the *Fake Object* does not: It merely provides a way for the interactions to occur in a self-consistent manner. These interactions (i.e., between the SUT and the *Fake 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 behavior with that of *Test Stubs* and *Mock Objects* (page 544), where the responses are either hard-coded or configured by the test.

While the test does not normally configure a *Fake Object*, complex fixture setup that would typically involve initializing the state of the DOC may also be done with the *Fake Object* directly using *Back Door Manipulation* (page 327). Techniques such as *Data Loader* (see *Back Door Manipulation*) and *Back Door Setup* (see *Back Door Manipulation*) can be used quite successfully with less fear of *Overspecified Software* (see *Fragile Test* on page 239) because they simply bind us to the interface between the SUT and the *Fake Object*; the interface used to configure the *Fake Object* is a test-only concern.

When to Use It

We should use a *Fake Object* whenever the SUT depends on other components that are unavailable or that make testing difficult or slow (e.g., *Slow Tests;* see page 253) and the tests need more complex sequences of behavior than are worth implementing in a *Test Stub* or *Mock Object*. It must also be easier to create a lightweight implementation than to build and program suitable *Mock Objects*, at least in the long run, if building a *Fake Object* is to be worthwhile.

Using a *Fake Object* helps us avoid *Overspecified Software* because we do not encode the exact calling sequences expected of the DOC within the test. The SUT can vary how many times the methods of the DOC are called without causing tests to fail.

If we need to control the indirect inputs or verify the indirect outputs of the SUT, we should probably use a *Mock Object* or *Test Stub* instead.

Some specific situations where we replace the real component with a *Fake Object* are described next.

Variation: Fake Database

With the *Fake Database* pattern, the real database or persistence layer is replaced by a *Fake Object* that is functionally equivalent but that has much better performance characteristics. An approach we have often used involves replacing the database with a set of in-memory `HashTables` that act as a very lightweight way of retrieving objects that have been "persisted" earlier in the test.

Variation: In-Memory Database

Another example of a *Fake Object* is the use of a small-footprint, diskless database instead of a full-featured disk-based database. This kind of *In-Memory Database* will improve the speed of tests by at least an order of magnitude while giving up less functionality than a *Fake Database*.

Variation: Fake Web Service

When testing software that depends on other components that are accessed as Web services, we can build a small hard-coded or data-driven implementation that can be used instead of the real Web service to make our tests more robust and to avoid having to create a test instance of the real Web service in our development environment.

Variation: Fake Service Layer

When testing user interfaces, we can avoid *Data Sensitivity* (see *Fragile Test*) and *Behavior Sensitivity* (see *Fragile Test*) of the tests by replacing the component that implements the Service Layer [PEAA] (including the domain layer) of our application with a Fake Object that returns remembered or data-driven results. This approach allows us to focus on testing the user interface without having to worry about the data being returned changing over time.

Implementation Notes

Introducing a *Fake Object* involves two basic concerns:

- Building the *Fake Object* implementation
- Installing the *Fake Object*

Building the Fake Object

Most *Fake Objects* are hand-built. Often, the *Fake Object* is used to replace a real implementation that suffers from latency issues owing 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 fake implementation that is sufficient to satisfy the needs of the SUT, at least for the purposes of specific tests, with relatively little effort.

A popular strategy is to start by building a *Fake Object* to support a specific set of tests where the SUT requires only a subset of the DOC's services. If this proves successful, we may consider expanding the *Fake Object* to handle additional tests. Over time, we may find that we can run all of our tests using the *Fake Object*. (See the sidebar "Faster Tests Without Shared Fixtures" on page 319 for a description of how we faked out the entire database with hash tables and made our tests run 50 times faster.)

Installing the Fake Object

Of course, we must have a way of installing the *Fake Object* into the SUT to be able to take advantage of it. We can use whichever substitutable dependency pattern the SUT supports. A common approach in the test-driven development community is *Dependency Injection* (page 678); more traditional developers may favor *Dependency Lookup* (page 686). The latter technique is also more appropriate when we introduce a *Fake Database* (see *Fake Object* on page 551) in an effort to speed up execution of the customer tests; *Dependency Injection* doesn't work so well with these kinds of tests.

Motivating Example

In this example, the SUT 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 then the test removes the records from the database (several deletes). All of this work takes time—several seconds per test. This very quickly adds up to minutes, and soon we find that our developers aren't running the tests quite so frequently. Here is an example of one of these tests:

```
public void testReadWrite() throws Exception{
   // Setup
   FlightMngtFacade facade = new FlightMgmtFacadeImpl();
   BigDecimal yyc = facade.createAirport("YYC", "Calgary", "Calgary");
   BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
   facade.createFlight(yyc, lax);
   // Exercise
   List flights = facade.getFlightsByOriginAirport(yyc);
```

```
      // Verify
      assertEquals( "# of flights", 1, flights.size());
      Flight flight = (Flight) flights.get(0);
      assertEquals( "origin",
                    yyc, flight.getOrigin().getCode());
   }
```

The test calls createAirport on our *Service Facade* [CJ2EEP], which calls, among other things, our data access layer. Here is the actual implementation of several of the methods we are calling:

```
   public BigDecimal createAirport( String airportCode,
                                    String name,
                                    String nearbyCity)
   throws FlightBookingException{
      TransactionManager.beginTransaction();
      Airport airport = dataAccess.
            createAirport(airportCode, name, nearbyCity);
      logMessage("Wrong Action Code", airport.getCode());//bug
      TransactionManager.commitTransaction();
      return airport.getId();
   }

   public List getFlightsByOriginAirport(
                    BigDecimal originAirportId)
         throws FlightBookingException {

      if (originAirportId == null)
         throw new InvalidArgumentException(
                 "Origin Airport Id has not been provided",
                 "originAirportId", null);
      Airport origin = dataAccess.getAirportByPrimaryKey(originAirportId);
      List flights = dataAccess.getFlightsByOriginAirport(origin);

      return flights;
   }
```

The calls to dataAccess.createAirport, dataAccess.createFlight, and TransactionManager.commitTransaction cause our test to slow down the most. The calls to dataAccess.getAirportByPrimaryKey and dataAccess.getFlightsByOriginAirport are a lesser factor but still contribute to the slow test.

Refactoring Notes

The steps for introducing a *Fake Object* are very similar to those for adding a *Mock Object*. If one doesn't already exist, we use a Replace Dependency with Test Double (page 522) refactoring to introduce a way to substitute the *Fake Object* for the DOC—usually a field (attribute) to hold the reference to it. In statically typed languages, we may have to do an Extract Interface [Fowler] refactoring before we can introduce the fake implementation. Then, we use this interface as the type of variable that holds the reference to the substitutable dependency.

One notable difference is that we *do not* need to configure the *Fake Object* with expectations or return values; we merely set up the fixture in the normal way.

Example: Fake Database

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

```
   public void testReadWrite_inMemory() throws Exception{
      // Setup
      FlightMgmtFacadeImpl facade = new FlightMgmtFacadeImpl();
      facade.setDao(new InMemoryDatabase());
      BigDecimal yyc = facade.createAirport("YYC", "Calgary", "Calgary");
      BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
      facade.createFlight(yyc, lax);
      // Exercise
      List flights = facade.getFlightsByOriginAirport(yyc);
      // Verify
      assertEquals( "# of flights", 1, flights.size());
      Flight flight = (Flight) flights.get(0);
      assertEquals( "origin",
                    yyc, flight.getOrigin().getCode());
   }
```

Here's the implementation of the *Fake Database:*

```
public class InMemoryDatabase implements FlightDao{
   private List airports = new Vector();
   public Airport createAirport(String airportCode,
                            String name, String nearbyCity)
         throws DataException, InvalidArgumentException {
      assertParamtersAreValid(  airportCode, name, nearbyCity);
      assertAirportDoesntExist( airportCode);
      Airport result = new Airport(getNextAirportId(),
            airportCode, name, createCity(nearbyCity));
      airports.add(result);
      return result;
   }
   public Airport getAirportByPrimaryKey(BigDecimal airportId)
         throws DataException, InvalidArgumentException {
      assertAirportNotNull(airportId);

      Airport result = null;
      Iterator i = airports.iterator();
      while (i.hasNext()) {
         Airport airport = (Airport) i.next();
         if (airport.getId().equals(airportId)) {
            return airport;
         }
      }
      throw new DataException("Airport not found:"+airportId);
   }
```

Now all we need is the implementation of the method that installs the *Fake Database* into the facade to make our developers more than happy to run all the tests after every code change.

```
   public void setDao(FlightDao) {
      dataAccess = dao;
   }
```

Further Reading

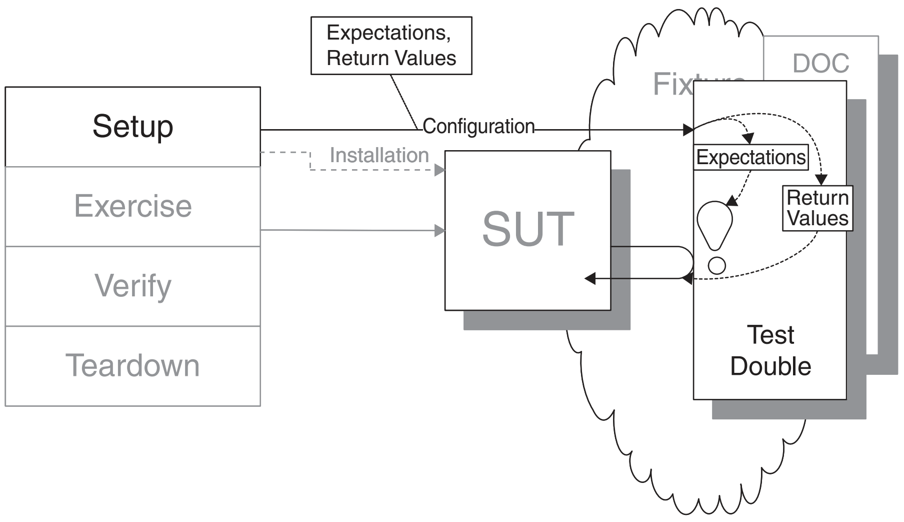
The sidebar "Faster Tests Without Shared Fixtures" on page 319 provides a more in-depth description of how we faked out the entire database with hash tables and made our tests run 50 times faster. *Mocks, Fakes, Stubs, and Dummies* (in Appendix B) contains a more thorough comparison of the terminology used in various books and articles.

Configurable Test Double

How do we tell a Test Double what to return or expect?

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

Also known as: *Configurable Mock Object, Configurable Test Spy, Configurable Test Stub*



Some tests require unique values to be fed into the SUT as indirect inputs or to be verified as indirect outputs of the SUT. This approach typically requires the use of *Test Doubles* (page 522) as the conduit between the test and the SUT; at the same time, the *Test Double* somehow needs to be told which values to return or verify.

A *Configurable Test Double* is a way to reduce *Test Code Duplication* (page 213) by reusing a *Test Double* in many tests. The key to its use is to configure the *Test Double's* values to be returned or expected at runtime.

How It Works

The *Test Double* is built with instance variables that hold the values to be returned to the SUT or to serve as the expected values of arguments to method calls. The test initializes these variables during the setup phase of the test by calling the appropriate methods on the *Test Double's* interface. When the SUT calls the methods on the *Test Double*, the *Test Double* uses the contents of the appropriate variable as the value to return or as the expected value in assertions.

When to Use It

We can use a *Configurable Test Double* whenever we need similar but slightly different behavior in several tests that depend on *Test Doubles* and we want to avoid *Test Code Duplication* or *Obscure Tests* (page 186)—in the latter case, we need to see what values the *Test Double* is using as we read the test. If we expect only a single usage of a *Test Double*, we can consider using a *Hard-Coded Test Double* (page 568) if the extra effort and complexity of building a *Configurable Test Double* are not warranted.

Implementation Notes

A *Test Double* is a *Configurable Test Double* because it needs to provide a way for the tests to configure it with values to return and/or method arguments to expect. Configurable *Test Stubs* (page 529) and *Test Spies* (page 538) simply require a way to configure the responses to calls on their methods; configurable *Mock Objects* (page 544) also require a way to configure their expectations (which methods *should* be called and with which arguments).

Configurable Test Doubles may be built in many ways. Deciding on a particular implementation involves making two relatively independent decisions: (1) how the *Configurable Test Double* will be configured and (2) how the *Configurable Test Double* will be coded.

There are two common ways to configure a *Configurable Test Double*. The most popular approach is to provide a *Configuration Interface* that is used only by the test to configure the values to be returned as indirect inputs and the expected values of the indirect outputs. Alternatively, we may build the *Configurable Test Double* with two modes. The *Configuration Mode* is used during fixture setup to install the indirect inputs and expected indirect outputs by calling the methods of the *Configurable Test Double* with the expected arguments. Before the *Configurable Test Double* is installed, it is put into the normal ("usage" or "playback") mode.

The obvious way to build a *Configurable Test Double* is to create a *Hand-Built Test Double*. If we are lucky, however, someone will have already built a tool to generate a *Configurable Test Double* for us. *Test Double* generators come in two flavors: code generators and tools that fabricate the object at runtime. Developers have built several generations of "mocking" tools, and several of these have been ported to other programming languages; check out http://xprogramming.com to see what is available in your programming language of choice. If the answer is "nothing," you can hand-code the *Test Double* yourself, although this does take somewhat more effort.

Variation: Configuration Interface

A *Configuration Interface* comprises a separate set of methods that the *Configurable Test Double* provides specifically for use by the test to set each value that the *Configurable Test Double* returns or expects to receive. The test simply calls these methods during the fixture setup phase of the *Four-Phase Test* (page 358). The SUT uses the "other" methods on the *Configurable Test Double* (the "normal" interface). It isn't aware that the *Configuration Interface* exists on the object to which it is delegating.

Configuration Interfaces come in two flavors. Early toolkits, such as MockMaker, generated a distinct method for each value we needed to configure. The collection of these setter methods made up the *Configuration Interface*. More recently introduced toolkits, such as JMock, provide a generic interface that is used to build an *Expected Behavior Specification* (see *Behavior Verification* on page 468) that the *Configurable Test Double* interprets at runtime. A well-designed **fluent interface** can make the test much easier to read and understand.

Variation: Configuration Mode

We can avoid defining a separate set of methods to configure the *Test Double* by providing a *Configuration Mode* that the test uses to "teach" the *Configurable Test Double* what to expect. At first glance, this means of configuring the *Test Double* can be confusing: Why does the *Test Method* (page 348) call the methods of this other object before it calls the methods it is exercising on the SUT? When we come to grips with the fact that we are doing a form of "record and playback," this technique makes a bit more sense.

The main advantage of using a *Configuration Mode* is that it avoids creating a separate set of methods for configuring the *Configurable Test Double* because we reuse the same methods that the SUT will be calling. (We do have to provide a way to set the values to be returned by the methods, so we have at least one additional method to add.) On the flip side, each method that the SUT is expected to call now has two code paths through it: one for the *Configuration Mode* and another for the "usage mode."

Variation: Hand-Built Test Double

A *Hand-Built Test Double* is one that was defined by the test automater for one or more specific tests. A *Hard-Coded Test Double* is inherently a *Hand-Built Test Double*, while a *Configurable Test Double* can be either hand-built or generated. This book uses *Hand-Built Test Doubles* in a lot of the examples because it is easier to see what is going on when we have actual, simple, concrete code to look at. This is the main advantage of using a *Hand-Built Test Double;* indeed, some people consider this benefit to be so important that they use *Hand-Built Test Doubles* exclusively. We may also use a *Hand-Built Test Double* when no third-party toolkits are available or if we are prevented from using those tools by project or corporate policy.

Variation: Statically Generated Test Double

The early third-party toolkits used code generators to create the code for *Statically Generated Test Doubles*. The code is then compiled and linked with our handwritten test code. Typically, we will store the code in a source code repository [SCM]. Whenever the interface of the target class changes, of course, we must regenerate the code for our *Statically Generated Test Doubles*. It may be advantageous to include this step as part of the automated build script to ensure that it really does happen whenever the interface changes.

Instantiating a *Statically Generated Test Double* is the same as instantiating a *Hand-Built Test Double*. That is, we use the name of the generated class to construct the *Configurable Test Double*.

An interesting problem arises during refactoring. Suppose we change the interface of the class we are replacing by adding an argument to one of the methods. Should we then refactor the generated code? Or should we regenerate the *Statically Generated Test Double* after the code it replaces has been refactored? With modern refactoring tools, it may seem easier to refactor the generated code and the tests that use it in a single step; this strategy, however, may leave the *Statically Generated Test Double* without argument verification logic or variables for the new parameter. Therefore, we should regenerate the *Statically Generated Test Double* after the refactoring is finished to ensure that the refactored *Statically Generated Test Double* works properly and can be recreated by the code generator.

Variation: Dynamically Generated Test Double

Newer third-party toolkits generate *Configurable Test Doubles* at runtime by using the reflection capabilities of the programming language to examine a class or interface and build an object that is capable of understanding all calls to its methods. These *Configurable Test Doubles* may interpret the behavior specification at runtime or they may generate executable code; nevertheless, there is no source code for us to generate and maintain or regenerate. The down side is simply that there is no code to look at—but that really isn't a disadvantage unless we are particularly suspicious or paranoid.

Most of today's tools generate *Mock Objects* because they are the most fashionable and widely used options. We can still use these objects as *Test Stubs*, however, because they do provide a way of setting the value to be returned when a particular method is called. If we aren't particularly interested in verifying the methods being called or the arguments passed to them, most toolkits provide a way to specify "don't care" arguments. Given that most toolkits generate *Mock Objects*, they typically don't provide a *Retrieval Interface* (see *Test Spy*).

Motivating Example

Here's a test that uses a *Hard-Coded Test Double* to give it control over the time:

```
   public void testDisplayCurrentTime_AtMidnight_HCM()
            throws Exception {
      // Fixture Setup
      //   Instantiate hard-code Test Stub:
      TimeProvider testStub = new MidnightTimeProvider();
      //   Instantiate SUT
      TimeDisplay sut = new TimeDisplay();
      //   Inject Stub into SUT
      sut.setTimeProvider(testStub);
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // Verify Direct Output
      String expectedTimeString =
         "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }
```

This test is hard to understand without seeing the definition of the *Hard-Coded Test Double*. It is easy to see how this lack of clarity can lead to a *Mystery Guest* (see *Obscure Test*) if the definition is not close at hand.

```
   class MidnightTimeProvider implements TimeProvider {
      public Calendar getTime() {
         Calendar myTime = new GregorianCalendar();
         myTime.set(Calendar.HOUR_OF_DAY, 0);
         myTime.set(Calendar.MINUTE, 0);
         return myTime;
      }
   }
```

We can solve the *Obscure Test* problem by using a *Self Shunt* (see *Hard-Coded Test Double*) to make the *Hard-Coded Test Double* visible within the test:

```
public class SelfShuntExample extends TestCase
implements TimeProvider {
   public void testDisplayCurrentTime_AtMidnight() throws Exception {
      // Fixture Setup
      TimeDisplay sut = new TimeDisplay();
      // Mock Setup
      sut.setTimeProvider(this); // self shunt installation
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // Verify Direct Output
      String expectedTimeString =
         "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }

   public Calendar getTime() {
      Calendar myTime = new GregorianCalendar();
      myTime.set(Calendar.MINUTE, 0);
      myTime.set(Calendar.HOUR_OF_DAY, 0);
      return myTime;
   }
}
```

Unfortunately, we will need to build the *Test Double* behavior into each *Testcase Class* (page 373) that requires it, which results in *Test Code Duplication*.

Refactoring Notes

Refactoring a test that uses a *Hard-Coded Test Double* to become a test that uses a third-party *Configurable Test Double* is relatively straightforward. We simply follow the directions provided with the toolkit to instantiate the *Configurable Test Double* and configure it with the same values as we used in the *Hard-Coded Test Double*. We may also have to move some of the logic that was originally hard-coded within the *Test Double* into the *Test Method* and pass it in to the *Test Double* as part of the configuration step.

Converting the actual *Hard-Coded Test Double* into a *Configurable Test Double* is a bit more complicated, but not overly so if we need to capture only simple behavior. (For more complex behavior, we're probably better off examining one of the existing toolkits and porting it to our environment if it is not yet available.) First we need to introduce a way to set the values to be returned or expected. The best choice is to start by modifying the test to see how we want to interact with the *Configurable Test Double*. After instantiating it during the fixture setup part of the test, we then pass the test-specific values to the *Configurable Test Double* using the emerging *Configuration Interface* or *Configuration Mode*. Once we've seen how we want to use the *Configurable Test Double*, we can use an Introduce Field [JetBrains] refactoring to create the instance variables of the *Configurable Test Double* to hold each of the previously hard-coded values.

Example: Configuration Interface Using Setters

The following example shows how a test would use a simple hand-built *Configuration Interface* using *Setter Injection*:

```
public void testDisplayCurrentTime_AtMidnight()
            throws Exception {
   // Fixture setup
   //      Test Double configuration
   TimeProviderTestStub tpStub = new TimeProviderTestStub();
   tpStub.setHours(0);
   tpStub.setMinutes(0);
   //   Instantiate SUT
   TimeDisplay sut = new TimeDisplay();
   //      Test Double installation
   sut.setTimeProvider(tpStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify Outcome
   String expectedTimeString =
           "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```

Configurable Test Double

The *Configurable Test Double* is implemented as follows:

```
class TimeProviderTestStub implements TimeProvider {
   // Configuration Interface
   public void setHours(int hours) {
      // 0 is midnight; 12 is noon
      myTime.set(Calendar.HOUR_OF_DAY, hours);
   }

   public void setMinutes(int minutes) {
      myTime.set(Calendar.MINUTE, minutes);
   }
   // Interface Used by SUT
   public Calendar getTime() {
      // @return the last time that was set
      return myTime;
   }
}
```

Example: Configuration Interface Using Expression Builder

Now let's contrast the *Configuration Interface* we defined in the previous example with the one provided by the JMock framework. JMock generates *Mock Objects* dynamically and provides a generic fluent interface for configuring the *Mock Object* in an intent-revealing style. Here's the same test converted to use JMock:

```
public void testDisplayCurrentTime_AtMidnight_JM()
      throws Exception {
   // Fixture setup
   TimeDisplay sut = new TimeDisplay();
   //  Test Double configuration
   Mock tpStub = mock(TimeProvider.class);
   Calendar midnight = makeTime(0,0);
   tpStub.stubs().method("getTime").
               withNoArguments().
               will(returnValue(midnight));
   //  Test Double installation
   sut.setTimeProvider((TimeProvider) tpStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify Outcome
   String expectedTimeString =
           "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```

Here we have moved some of the logic to construct the time to be returned into the *Testcase Class* because there is no way to do it in the generic mocking framework; we've used a *Test Utility Method* (page 599) to construct the time to be returned. This next example shows a configurable *Mock Object* complete with multiple expected parameters:

```
public void testRemoveFlight_JMock() throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnonRegFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   // mock configuration
   Mock mockLog = mock(AuditLog.class);
   mockLog.expects(once()).method("logMessage")
            .with(eq(helper.getTodaysDateWithoutTime()),
                  eq(Helper.TEST_USER_NAME),
                  eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
                  eq(expectedFlightDto.getFlightNumber()));
   // mock installation
   facade.setAuditLog((AuditLog) mockLog.proxy());
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight still exists after being removed",
               facade.flightExists( expectedFlightDto.
                                       getFlightNumber()));
   // verify() method called automatically by JMock
}
```

The *Expected Behavior Specification* is built by calling expression-building methods such as `expects`, `once`, and `method` to describe how the *Configurable*

Test Double should be used and what it should return. JMock supports the specification of much more sophisticated behavior (such as multiple calls to the same method with different arguments and return values) than does our hand-built *Configurable Test Double*.

Example: Configuration Mode

In the next example, the test has been converted to use a *Mock Object* with a *Configuration Mode*:

```
public void testRemoveFlight_ModalMock() throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnonRegFlight();
   // mock configuration (in Configuration Mode)
   ModalMockAuditLog mockLog = new ModalMockAuditLog();
   mockLog.logMessage(Helper.getTodaysDateWithoutTime(),
                      Helper.TEST_USER_NAME,
                      Helper.REMOVE_FLIGHT_ACTION_CODE,
                      expectedFlightDto.getFlightNumber());
   mockLog.enterPlaybackMode();
   // 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();
}
```

Here the test calls the methods on the *Configurable Test Double* during the fixture setup phase. If we weren't aware that this test uses a *Configurable Test Double* mock, we might find this structure confusing at first glance. The most obvious clue to its intent is the call to the method `enterPlaybackMode`, which tells the *Configurable Test Double* to stop saving expected values and to start asserting on them.

The *Configurable Test Double* used by this test is implemented like this:

```
private int mode = record;

public void enterPlaybackMode() {
   mode = playback;
}

public void logMessage( Date date,
                        String user,
                        String action,
                        Object detail) {
```

```
      if (mode == record) {
         Assert.assertEquals("Only supports 1 expected call",
                             0, expectedNumberCalls);
         expectedNumberCalls = 1;
         expectedDate = date;
         expectedUser = user;
         expectedCode = action;
         expectedDetail = detail;
      } else {
         Assert.assertEquals("Date", expectedDate, date);
         Assert.assertEquals("User", expectedUser, user);
         Assert.assertEquals("Action", expectedCode, action);
         Assert.assertEquals("Detail", expectedDetail, detail);
      }
   }
```

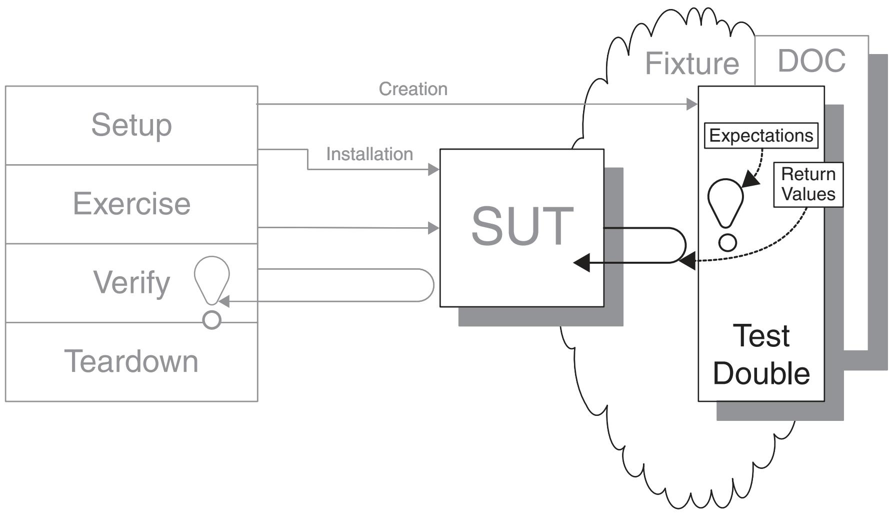
The `if` statement checks whether we are in record or playback mode. Because this simple hand-built *Configurable Test Double* allows only a single value to be stored, a *Guard Assertion* (page 490) fails the test if it tries to record more than one call to this method. The rest of the `then` clause saves the parameters into variables that it uses as the expected values of the *Equality Assertions* (see *Assertion Method* on page 362) in the `else` clause.

Hard-Coded Test Double

How do we tell a Test Double what to return or expect?

Also known as: *Hard-Coded Mock Object, Hard-Coded Test Stub, Hard-Coded Test Spy*

We build the Test Double by hard-coding the return values and/or expected calls.



Test Doubles (page 522) are used for many reasons during the development of *Fully Automated Tests* (see page 26). The behavior of the *Test Double* may vary from test to test, and there are many ways to define this behavior.

When the *Test Double* is very simple or very specific to a single test, the simplest solution is often to hard-code the behavior into the *Test Double*.

How It Works

The test automater hard-codes all of the *Test Double's* behavior into the *Test Double*. For example, if the *Test Double* needs to return a value for a method call, the value is hard-coded into the return statement. If it needs to verify that a certain parameter had a specific value, the assertion is hard-coded with the value that is expected.

When to Use It

We typically use a *Hard-Coded Test Double* when the behavior of the *Test Double* is very simple or is very specific to a single test or *Testcase Class* (page 373). The *Hard-Coded Test Double* can be either a *Test Stub* (page 529), a *Test Spy* (page 538), or a *Mock Object* (page 544), depending on what we encode in the method(s) called by the SUT.

Because each *Hard-Coded Test Double* is purpose-built by hand, its construction may take more effort than using a third-party *Configurable Test Double* (page 558). It can also result in more test code to maintain and refactor as the SUT changes. If different tests require that the *Test Double* behave in different ways and the use of *Hard-Coded Test Doubles* results in too much *Test Code Duplication* (page 213), we should consider using a *Configurable Test Double* instead.

Implementation Notes

Hard-Coded Test Doubles are inherently *Hand-Built Test Doubles* (see *Configurable Test Double*) because there tends to be no point in generating *Hard-Coded Test Doubles* automatically. *Hard-Coded Test Doubles* can be implemented with dedicated classes, but they are most commonly used when the programming language supports blocks, **closures,** or **inner classes.** All of these language features help to avoid the file/class overhead associated with creating a *Hard-Coded Test Double*; they also keep the *Hard-Coded Test Double's* behavior visible within the test that uses it. In some languages, this can make the tests a bit more difficult to read. This is especially true when we use anonymous inner classes, which require a lot of syntactic overhead to define the class in-line. In languages that support blocks directly, and in which developers are very familiar with their usage idioms, using *Hard-Coded Test Doubles* can actually make the tests easier to read.

There are many different ways to implement a *Hard-Coded Test Double*, each of which has its own advantages and disadvantages.

Variation: Test Double Class

We can implement the *Hard-Coded Test Double* as a class distinct from either the *Testcase Class* or the SUT. This allows the *Hard-Coded Test Double* to be reused by several *Testcase Classes* but may result in an *Obscure Test* (page 186; caused by a *Mystery Guest*) because it moves important indirect inputs or indirect outputs of the SUT out of the test to somewhere else, possibly out of sight of the test reader. Depending on how we implement the *Test Double Class*, it may also result in code proliferation and additional *Test Double* classes to maintain.

One way to ensure that the *Test Double Class* is type-compatible with the component it will replace is to make the *Test Double Class* a subclass of that component. We then override any methods whose behavior we want to change.

Variation: Test Double Subclass

We can also implement the *Hard-Coded Test Double* by subclassing the real DOC and overriding the behavior of the methods we expect the SUT to call as we exercise it. Unfortunately, this approach can have unpredictable consequences if the SUT calls other DOC methods that we have not overridden. It also ties our test code very closely to the implementation of the DOC and can result in *Overspecified Software* (see *Fragile Test* on page 239). Using a *Test Double Subclass* may be a reasonable option in very specific circumstances (e.g., while doing a spike or when it is the only option available to us), but this strategy isn't recommended on a routine basis.

Variation: Self Shunt

Also known as: *Loopback, Testcase Class as Test Double*

We can implement the methods that we want the SUT to call on the *Testcase Class* and install the *Testcase Object* (page 382) into the SUT as the *Test Double* to be used. This approach is called a *Self Shunt*.

The *Self Shunt* can be either a *Test Stub*, a *Test Spy*, or a *Mock Object*, depending on what the method called by the SUT does. In each case, it will need to access instance variables of the *Testcase Class* to know what to do or expect. In statically typed languages, the *Testcase Class* must also implement the interface on which the SUT depends.

We typically use a *Self Shunt* when we need a *Hard-Coded Test Double* that is very specific to a single *Testcase Class*. If only a single *Test Method* (page 348) requires the *Hard-Coded Test Double*, using an *Inner Test Double* may result in greater clarity if our language supports it.

Variation: Inner Test Double

A popular way to implement a *Hard-Coded Test Double* is to code it as an anonymous inner class or block closure within the *Test Method*. This strategy gives the *Test Double* access to instance variables and constants of the *Testcase Class* and even the local variables of the *Test Method*, which can eliminate the need to configure the *Test Double*.

While the name of this variation is based on the name of the Java language construct of which it takes advantage, many programming languages have an equivalent mechanism for defining code to be run later using blocks or closures.

We typically use an *Inner Test Double* when we are building a *Hard-Coded Test Double* that is relatively simple and is used only within a single *Test Method*. Many people find the use of a *Hard-Coded Test Double* more intuitive than using a *Self Shunt* because they can see exactly what is going on within the *Test Method*. Readers who are unfamiliar with the syntax of anonymous inner classes or blocks may find the test difficult to understand, however.

Variation: Pseudo-Object

One challenge facing writers of *Hard-Coded Test Doubles* is that we must implement all the methods in the interface that the SUT *might* call. In statically typed languages such as Java and C#, we must at least implement all methods declared in the interface implied by the class or type associated with however we access the DOC. This often "forces" us to subclass from the real DOC to avoid providing dummy implementations for these methods.

One way of reducing the programming effort is to provide a default class that implements all the interface methods and throws a unique error. We can then implement a *Hard-Coded Test Double* by subclassing this concrete class and overriding just the one method we expect the SUT to call while we are exercising it. If the SUT calls any other methods, the *Pseudo-Object* throws an error, thereby failing the test.

Motivating Example

The following test verifies the basic functionality of the component that formats an HTML string containing the current time. Unfortunately, it depends on the real system clock, so it rarely passes!

```
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);
}
```

Refactoring Notes

The most common transition is from using the real component to using a *Hard-Coded Test Double*.[4] To make this transition, we need to build the *Test Double* itself and install it from within our *Test Method*. We may also need to introduce a way to install the *Test Double* using one of the *Dependency Injection* patterns (page 678) if the SUT does not already support this installation. The process for doing so is described in the Replace Dependency with Test Double (page 522) refactoring.

Example: Test Double Class

Here's the same test modified to use a *Hard-Coded Test Double* class to allow control over the time:

```
public void testDisplayCurrentTime_AtMidnight_HCM()
         throws Exception {
   // Fixture setup
   //   Instantiate hard-coded Test Stub
   TimeProvider testStub = new MidnightTimeProvider();
   //   Instantiate SUT
   TimeDisplay sut = new TimeDisplay();
   //   Inject Test Stub into SUT
   sut.setTimeProvider(testStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify direct output
   String expectedTimeString =
      "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```

This test is hard to understand without seeing the definition of the *Hard-Coded Test Double*. We can readily see how this approach might lead to an *Obscure Test* caused by a *Mystery Guest* if the *Hard-Coded Test Double* is not close at hand.

```
class MidnightTimeProvider implements TimeProvider {
   public Calendar getTime() {
      Calendar myTime = new GregorianCalendar();
      myTime.set(Calendar.HOUR_OF_DAY, 0);
      myTime.set(Calendar.MINUTE, 0);
      return myTime;
   }
}
```

[4] We rarely move from a *Configurable Test Double* to a *Hard-Coded Test Double* because we generally seek to make the *Test Double* more—not less—reusable.

Depending on the programming language, this *Test Double Class* can be defined in a number of different places, including within the body of the *Testcase Class* (an inner class) and as a separate free-standing class either in the same file as the test or in its own file. Of course, the farther away the *Test Double Class* resides from the *Test Method*, the more of a *Mystery Guest* it becomes.

Example: Self Shunt/Loopback

Here's a test that uses a *Self Shunt* to allow control over the time:

```
public class SelfShuntExample extends TestCase
implements TimeProvider {
   public void testDisplayCurrentTime_AtMidnight() throws Exception {
      // fixture setup
      TimeDisplay sut = new TimeDisplay();
      // mock setup
      sut.setTimeProvider(this); // self shunt installation
      // exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // verify direct output
      String expectedTimeString =
         "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }

   public Calendar getTime() {
      Calendar myTime = new GregorianCalendar();
      myTime.set(Calendar.MINUTE, 0);
      myTime.set(Calendar.HOUR_OF_DAY, 0);
      return myTime;
   }
}
```

Note how both the *Test Method* that installs the *Hard-Coded Test Double* and the implementation of the `getTime` method called by the SUT are members of the same class. We used the *Setter Injection* pattern (see *Dependency Injection*) to install the *Hard-Coded Test Double*. Because this example is written in a statically typed language, we had to add the clause `implements TimeProvider` to the *Testcase Class* declaration so that the `sut.setTimeProvider(this)` statement will compile. In a dynamically typed language, this step is unnecessary.

Example: Subclassed Inner Test Double

Here's a JUnit test that uses a *Subclassed Inner Test Double* using Java's "Anonymous Inner Class" syntax:

```
   public void testDisplayCurrentTime_AtMidnight_AIM() throws Exception {
      // Fixture setup
      //    Define and instantiate Test Stub
      TimeProvider testStub = new TimeProvider() {
      // Anonymous inner stub
         public Calendar getTime() {
            Calendar myTime = new GregorianCalendar();
            myTime.set(Calendar.MINUTE, 0);
            myTime.set(Calendar.HOUR_OF_DAY, 0);
            return myTime;
         }
      };
      //   Instantiate SUT
      TimeDisplay sut = new TimeDisplay();
      //   Inject Test Stub into SUT
      sut.setTimeProvider(testStub);
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
      // Verify direct output
      String expectedTimeString =
              "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals("Midnight", expectedTimeString, result);
   }
```

Here we used the name of the real depended-on class (`TimeProvider`) in the call to `new` for the definition of the *Hard-Coded Test Double*. By including a definition of the method `getTime` within curly braces after the classname, we are actually creating an anonymous *Subclassed Test Double* inside the *Test Method*.

Example: Inner Test Double Subclassed from Pseudo-Class

Suppose we have replaced one implementation of a method with another implementation that we need to leave around for backward-compatibility purposes, but we want to write tests to ensure that the old method is no longer called. This is easy to do if we already have the following *Pseudo-Object* definition:

```
/**
 * Base class for hand-coded Test Stubs and Mock Objects
 */
public class PseudoTimeProvider implements ComplexTimeProvider {

   public Calendar getTime() throws TimeProviderEx {
      throw new PseudoClassException();
   }

   public Calendar getTimeDifference(Calendar baseTime,
                                     Calendar otherTime)
            throws TimeProviderEx {
      throw new PseudoClassException();
   }

   public Calendar getTime( String timeZone ) throws TimeProviderEx {
      throw new PseudoClassException();
   }
}
```

We can now write a test that ensures the old version of the `getTime` method is *not* called by subclassing and overriding the newer version of the method (the one we *expect* to be called by the SUT):

```
public void testDisplayCurrentTime_AtMidnight_PS() throws Exception {
   // Fixture setup
   //    Define and instantiate Test Stub
   TimeProvider testStub = new PseudoTimeProvider()
   { // Anonymous inner stub
      public Calendar getTime(String timeZone) {
         Calendar myTime = new GregorianCalendar();
         myTime.set(Calendar.MINUTE, 0);
         myTime.set(Calendar.HOUR_OF_DAY, 0);
         return myTime;
      }
   };
   //   Instantiate SUT
   TimeDisplay sut = new TimeDisplay();
   //   Inject Test Stub into SUT:
   sut.setTimeProvider(testStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify direct output
   String expectedTimeString =
           "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```

If any of the other methods are called, the base class methods are invoked and throw an exception. Therefore, if we run this test and one of the methods we didn't override is called, we will see the following output as the first line of the JUnit stack trace for this test error:

```
com..PseudoClassEx: Unexpected call to unsupported method.
at com..PseudoTimeProvider.getTime(PseudoTimeProvider.java:22)
at com..TimeDisplay.getCurrentTimeAsHtmlFragment(TimeDisplay.java:64)
at com..TimeDisplayTestSolution.
   testDisplayCurrentTime_AtMidnight_PS(
      TimeDisplayTestSolution.java:247)
```

What's in a (Pattern) Name?

The Importance of Good Names

Names are important because they are a key part of how we communicate. Names are labels we attach to concepts. Good names help us communicate those concepts. This is true when we are communicating with people who already know the names, but especially when we are communicating with people who don't. Consider the following example.

Early in my pattern-writing days, I attended the very first Pattern Languages of Programs (PLoP) conference (http://www.hillside.net/conferences/plop). At the conference, the well-known author Jim Coplien ("Cope," to his friends) had a pattern language of organizational patterns being workshopped. One of the patterns was called "Buffalo Mountain"; another was called "Architect Also Implements." These two pattern names are at opposite ends of the spectrum as far as pattern names are concerned.

The gist of "Architect Also Implements" can be gleaned from the pattern name even if a person has not read the actual pattern. The name is both a placeholder for the pattern and meaningful in its own right.

The name "Buffalo Mountain," by contrast, does not readily communicate its underlying meaning. To this day I can still remember the story behind the name—but I cannot remember the actual focus of the pattern. The name was based on a graph that plotted some data related to the pattern. An early reviewer thought it resembled the profile of a nearby mountain called Buffalo Mountain. Thus, while the pattern name is memorable, it is not very evocative.

Closer to home, *Self Shunt* (see *Hard-Coded Test Double* on page 568) is an example of a name that is less than evocative because the term "shunt" is not widely used except in a few specialized fields. Michael Feathers does a good job explaining the background of the name in his description of the pattern. Unless you've read that description, however, the name is "just a name." A more evocative name might be something like "Testcase Class as Test Double" or "Loopback" but even the latter suffers from ambiguity because it isn't clear what is being looped back. So the name *Self Shunt* survives because it is in common use.

Other Naming Considerations

People might ask why I sometimes propose alternative names for some patterns. The preceding story highlights one of the reasons. Another reason is that in a larger collection of patterns (such as this book), it is important that there exists a "system of names."

Let me illustrate this second reason with an example. Many people advocate the use of a `setUp` method to create the test fixture. This approach moves the fixture setup logic out of each individual *Test Method* (page 348) and into a single place where it can be reused. Many people might refer to this pattern as "Shared Setup Method." But in this **pattern language,** I've chosen to call it *Implicit Setup* (page 424). Why?

It comes down to the names of other patterns in the language. On the one hand, "Shared Setup Method" could easily be confused with the existing pattern *Shared Fixture* (page 317). (The former pattern deals with sharing code, whereas the latter pattern focuses on sharing the runtime objects in the fixture.) On the other hand, the two major alternatives to *Implicit Setup* are called *In-line Setup* (page 408) and *Delegated Setup* (page 411). Wouldn't you agree that "In-line Setup, Delegated Setup, Implicit Setup" forms a better "system of names" than "In-line Setup, Delegated Setup, Shared Setup Method"? The connection between the pattern names is much more obvious when we consider all the major alternative patterns when choosing the system of names.

Why Standardize Testing Patterns?

The last part of this soapbox highlights why I think it is important for us to standardize the names of the test automation patterns, especially those related to *Test Stubs* (page 529) and *Mock Objects* (page 544). The key issue here relates to succinctness of communication.

When someone tells you, "Put a mock in it" (pun intended!), what advice is that person giving you? Depending on what the person means by a "mock," he or she could be suggesting that you control the indirect inputs of your SUT using a *Test Stub* or that you replace your database with a *Fake Database* (see *Fake Object* on page 551) that will reduce test interactions and speed up your tests by a factor of 50. (Yes, 50! See the sidebar "Faster Tests Without Shared Fixtures" on page 319.) Or perhaps the person is suggesting that you verify that your SUT calls the correct methods by installing an *Eager Mock Object* (see *Mock Object*) preconfigured

Continued...

with the *Expected Behavior* (see *Behavior Verification* on page 468). If everyone used "mock" to mean a *Mock Object*—no more or less—then the advice would be pretty clear. As I write this, the advice is very murky because we have taken to calling just about any *Test Double* (page 522) a "mock object" (despite the objections of the authors of the original paper on *Mock Objects* [ET]).

Further Reading

If you want to find out what "Buffalo Mountain" is really about, go to http://www1.bell-labs.com/user/cope/Patterns/Process/section29.html.

You can find "Architect Also Implements" at http://www1.bell-labs.com/user/cope/Patterns/Process/section16.html.

Interestingly, Alistair Cockburn wrote a similar comparison of pattern names in an article on his Web site (http://alistair.cockburn.us) and chose exactly the same two pattern names in his comparison. Coincidence or pattern?

In addition to failing the test, this scheme makes it very easy to see exactly which method *was* called. The bonus is that it works for calls to all unexpected methods with no additional effort.

Further Reading

Many of the "how to" books on test-driven development provide examples of *Self Shunt,* including [TDD-APG], [TDD-BE], [UTwJ], [PUT], and [JuPG]. The original write-up was by Michael Feathers and is accessible at http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf

The original "Shunt" pattern is written up at http://http://c2.com/cgi/wiki?ShuntPattern, along with a list of alternative names including "Loopback." See the sidebar "What's in a (Pattern) Name?" on page 576 for a discussion of how to select meaningful and evocative pattern names.

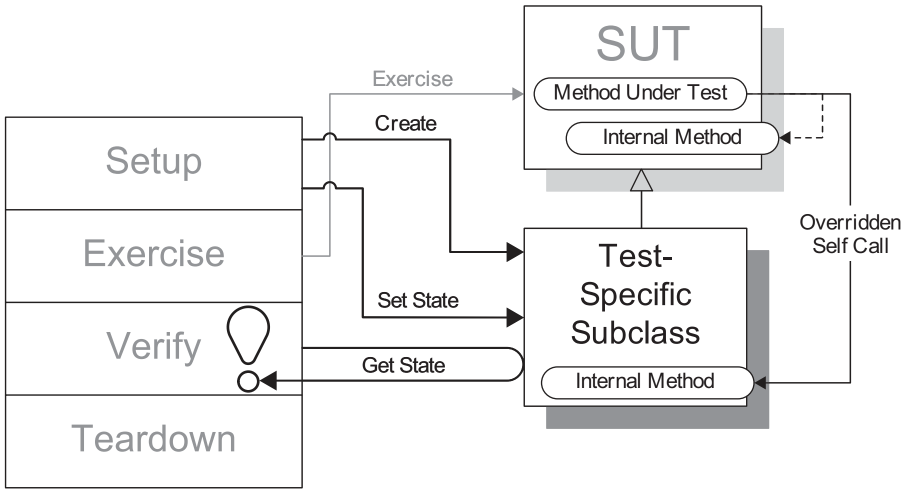
The *Pseudo-Object* pattern is described in the paper "Pseudo-Classes: Very Simple and Lightweight Mock Object-like Classes for Unit-Testing" available at http://www.devx.com/Java/Article/22599/1954?pf=true.

Test-Specific Subclass

How can we make code testable when we need to access private state of the SUT?

We add methods that expose the state or behavior needed by the test to a subclass of the SUT.

Also known as: *Test-Specific Extension*



If the SUT was not designed specifically to be testable, we may find that the test cannot gain access to a state that it must initialize or verify at some point in the test.

A *Test-Specific Subclass* is a simple yet very powerful way to open up the SUT for testing purposes without modifying the code of the SUT itself.

How It Works

We define a subclass of the SUT and add methods that modify the behavior of the SUT just enough to make it testable by implementing control points and observation points. This effort typically involves exposing instance variables using setters and getters or perhaps adding a method to put the SUT into a specific state without moving through its entire life cycle.

Because the *Test-Specific Subclass* would be packaged together with the tests that use it, the use of a *Test-Specific Subclass* does not change how the SUT is seen by the rest of the application.

When to Use It

We should use a *Test-Specific Subclass* whenever we need to modify the SUT to improve its testability but doing so directly would result in *Test Logic in Production* (page 217). Although we can use a *Test-Specific Subclass* for a number of purposes, all of those scenarios share a common goal: They improve testability by letting us get at the insides of the SUT more easily. A *Test-Specific Subclass* can be a double-edged sword, however. By breaking encapsulation, it allows us to tie our tests even more closely to the implementation, which can in turn result in *Fragile Tests* (page 239).

Variation: State-Exposing Subclass

If we are doing *State Verification* (page 462), we can subclass the SUT (or some component of it) so that we can see the internal state of the SUT for use in *Assertion Methods* (page 362). Usually, this effort involves adding **accessor** methods for private instance variables. We may also allow the test to set the state as a way to avoid *Obscure Tests* (page 186) caused by *Obscure Setup* (see *Obscure Test*) logic.

Variation: Behavior-Exposing Subclass

If we want to test the individual steps of a complex algorithm individually, we can subclass the SUT to expose the private methods that implement the Self-Calls [WWW]. Because most languages do not allow for relaxing the visibility of a method, we often have to use a different name in the *Test-Specific Subclass* and make a call to the superclass's method.

Variation: Behavior-Modifying Subclass

If the SUT contains some behavior that we do not want to occur when testing, we can override whatever method implements the behavior with an empty method body. This technique works best when the SUT uses Self-Calls (or a Template Method [GOF]) to delegate the steps of an algorithm to methods on itself or subclasses.

Also known as: *Subclassed Test Double*

Variation: Test Double Subclass

To ensure that a *Test Double* (page 522) is type-compatible with a DOC we wish to replace, we can make the *Test Double* a subclass of that component. This may be the only way we can build a *Test Double* that the compiler will accept when variables are statically typed using concrete classes.[5] (We should not have to take this step with dynamically typed languages such as Ruby, Python, Perl, and JavaScript.) We then override any methods whose behavior we want to change and add any methods we require to transform the *Test Double* into a *Configurable Test Double* (page 558) if we so desire.

Unlike the *Behavior-Modifying Subclass*, the *Test Double Subclass* does not just "tweak" the behavior of the SUT (or a part thereof) but replaces it entirely with canned behavior.

Variation: Substituted Singleton

Also known as: *Subclassed Singleton, Substitutable Singleton*

The Substituted Singleton is a special case of *Test Double Subclass*. We use it when we want to replace a DOC with a *Test Double* and the SUT does not support *Dependency Injection* (page 678) or *Dependency Lookup* (page 686).

Implementation Notes

The use of a *Test-Specific Subclass* brings some challenges:

- Feature granularity: ensuring that any behavior we want to override or expose is in its own single-purpose method. It is enabled through copious use of small methods and Self-Calls.
- Feature visibility: ensuring that subclasses can access attributes and behavior of the SUT class. It is primarily an issue in statically typed languages such as Java, C#, and C++; dynamically typed languages typically do not enforce visibility.

As with *Test Doubles*, we must be careful to ensure that we do not replace any of the behavior we are actually trying to test.

In languages that support class extensions without the need for subclassing (e.g., Smalltalk, Ruby, JavaScript, and other dynamic languages), a *Test-Specific Subclass* can be implemented as a class extension in the test package. We need to be aware, however, whether the extensions will make it into production; doing so would introduce *Test Logic in Production*.

[5] That is, by using a concrete class as the type of the variable rather than an abstract class or interface.

Visibility of Features

In languages that enforce scope (visibility) of variables and methods, we may need to change the visibility of the variables to allow subclasses to access them. While such a change affects the actual SUT code, it would typically be considered much less intrusive or misleading than changing the visibility to `public` (thereby allowing any code in the application to access the variables) or adding the test-specific methods directly to the SUT.

For example, in Java, we might change the visibility of instance variables from `private` to `protected` to allow the *Test-Specific Subclass* to access them. Similarly, we might change the visibility of methods to allow the *Test-Specific Subclass* to call them.

Granularity of Features

Long methods are difficult to test because they often bring too many dependencies into play. By comparison, short methods tend to be much simpler to test because they do only one thing. Self-Call offers an easy way to reduce the size of methods. We delegate parts of an algorithm to other methods implemented on the same class. This strategy allows us to test these methods independently. We can also confirm that the calling method calls these methods in the right sequence by overriding them in a *Test Double Subclass* (see *Test-Specific Subclass* on page 579).

Self-Call is a part of good object-oriented code design in that it keeps methods small and focused on implementing a single responsibility of the SUT. We can use this pattern whenever we are doing test-driven development and have control over the design of the SUT. We may find that we need to introduce Self-Call when we encounter long methods where some parts of the algorithm depend on things we do not want to exercise (e.g., database calls). This likelihood is especially high, for example, when the SUT is built using a *Transaction Script* [PEAA] architecture. Self-Call can be retrofitted easily using the Extract Method [Fowler] refactoring supported by most modern IDEs.

Motivating Example

The test in the following example is nondeterministic because it depends on the time. Our SUT is an object that formats the time for display as part of a Web page. It gets the time by asking a Singleton called `TimeProvider` to retrieve the time from a calendar object that it gets from the container.

```
public void testDisplayCurrentTime_AtMidnight() throws Exception {
   // Set up SUT
```

```
      TimeDisplay theTimeDisplay = new TimeDisplay();
      // Exercise SUT
      String actualTimeString =
            theTimeDisplay.getCurrentTimeAsHtmlFragment();
      // Verify outcome
      String expectedTimeString =
            "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals( "Midnight",
                    expectedTimeString,
                    actualTimeString);
   }

   public void testDisplayCurrentTime_AtOneMinuteAfterMidnight()
            throws Exception {
      // Set up SUT
      TimeDisplay actualTimeDisplay = new TimeDisplay();
      // Exercise SUT
      String actualTimeString =
            actualTimeDisplay.getCurrentTimeAsHtmlFragment();
      // Verify outcome
      String expectedTimeString =
            "<span class=\"tinyBoldText\">12:01 AM</span>";
      assertEquals( "12:01 AM",
                    expectedTimeString,
                    actualTimeString);
   }
```

These tests rarely pass, and they never pass in the same test run! The code within the SUT looks like this:

```
   public String getCurrentTimeAsHtmlFragment() {
      Calendar timeProvider;
      try {
         timeProvider = getTime();
      } catch (Exception e) {
         return e.getMessage();
      }
         // etc.
   }

   protected Calendar getTime() {
      return TimeProvider.getInstance().getTime();
   }
```

The code for the Singleton follows:

```
public class TimeProvider {
   protected static TimeProvider soleInstance = null;

   protected TimeProvider() {};

   public static TimeProvider getInstance() {
      if (soleInstance==null) soleInstance = new TimeProvider();
      return soleInstance;
   }

   public Calendar getTime() {
      return Calendar.getInstance();
   }
}
```

Refactoring Notes

The precise nature of the refactoring employed to introduce a *Test-Specific Subclass* depends on why we are using one. When we are using a *Test-Specific Subclass* to expose "private parts" of the SUT or override undesirable parts of its behavior, we merely define the *Test-Specific Subclass* as a subclass of the SUT and create an instance of the *Test-Specific Subclass* to exercise in the setup fixture phase of our *Four-Phase Test* (page 358).

When we are using the *Test-Specific Subclass* to replace a DOC of the SUT, however, we need to use a Replace Dependency with Test Double (page 522) refactoring to tell the SUT to use our *Test-Specific Subclass* instead of the real DOC.

In either case, we either override existing methods or add new methods to the *Test-Specific Subclass* using our language-specific capabilities (e.g., subclassing or mixins) as required by our tests.

Example: Behavior-Modifying Subclass (Test Stub)

Because the SUT uses a Self-Call to the `getTime` method to ask the `TimeProvider` for the time, we have an opportunity to use a *Subclassed Test Double* to control the time.[6] Based on this idea we can take a stab at writing our tests as follows (I have shown only one test here):

```
   public void testDisplayCurrentTime_AtMidnight() {
      // Fixture setup
      TimeDisplayTestStubSubclass tss = new TimeDisplayTestStubSubclass();
      TimeDisplay sut = tss;
      //   Test Double configuration
      tss.setHours(0);
      tss.setMinutes(0);
      // Exercise SUT
      String result = sut.getCurrentTimeAsHtmlFragment();
```

[6] This decision is enabled by the fact that `getTime` was defined to be `protected`; we would not be able to do this if it was `private`.

```
      // Verify outcome
      String expectedTimeString =
               "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals( expectedTimeString, result );
   }
```

Note that we have used the *Test-Specific Subclass* class for the variable that receives the instance of the SUT; this approach ensures that the methods of the *Configuration Interface* (see *Configurable Test Double*) defined on the *Test-Specific Subclass* are visible to the test.[7] For documentation purposes, we have then assigned the *Test-Specific Subclass* to the variable `sut`; this is a safe cast because the *Test-Specific Subclass* class is a subclass of the SUT class. This technique also helps us avoid the *Mystery Guest* (see *Obscure Test*) problem caused by hard-coding an important indirect input of our SUT inside the *Test Stub* (page 529).

Now that we have seen how it will be used, it is a simple matter to implement the *Test-Specific Subclass*:

```
public class TimeDisplayTestStubSubclass extends TimeDisplay {

   private int hours;
   private int minutes;

   // Overridden method
   protected Calendar getTime() {
      Calendar myTime = new GregorianCalendar();
      myTime.set(Calendar.HOUR_OF_DAY, this.hours);
      myTime.set(Calendar.MINUTE, this.minutes);
      return myTime;
   }
   /*
    * Configuration Interface
    */
   public void setHours(int hours) {
      this.hours = hours;
   }

   public void setMinutes(int minutes) {
      this.minutes = minutes;
   }
}
```

There's no rocket science here—we just had to implement the methods used by the test.

[7] We could have used a *Hard-Coded Test Double* (page 568) subclass instead, but that tactic would have required a different *Test-Specific Subclass* for each time we want to test with. Each subclass would simply hard-code the return value of the `getTime` method.

Example: Behavior-Modifying Subclass (Substituted Singleton)

Suppose our getTime method was declared to be private[8] or static, final or sealed, and so on.[9] Such a declaration would prevent us from overriding the method's behavior in our *Test-Specific Subclass*. What could we do to address our *Nondeterministic Tests* (see *Erratic Test* on page 228)?

Because the design uses a Singleton [GOF] to provide the time, a simple solution is to replace the Singleton during test execution with a *Test Double Subclass*. We can do so as long as it is possible for a subclass to access its soleInstance variable. We use the Introduce Local Extension [Fowler] refactoring (specifically, the subclass variant of it) to create the *Test-Specific Subclass*. Writing the tests first helps us understand the interface we want to implement.

```
   public void testDisplayCurrentTime_AtMidnight() {
      TimeDisplay sut = new TimeDisplay();
      //   Install test Singleton
      TimeProviderTestSingleton timeProvideSingleton =
            TimeProviderTestSingleton.overrideSoleInstance();
      timeProvideSingleton.setTime(0,0);
      //   Exercise SUT
      String actualTimeString = sut.getCurrentTimeAsHtmlFragment();
      // Verify outcome
      String expectedTimeString =
            "<span class=\"tinyBoldText\">Midnight</span>";
      assertEquals( expectedTimeString, actualTimeString );
   }
```

Now that we have a test that uses the *Substituted Singleton*, we can proceed to implement it by subclassing the Singleton and defining the methods the tests will use.

```
public class TimeProviderTestSingleton extends TimeProvider {
   private Calendar myTime = new GregorianCalendar();
   private TimeProviderTestSingleton() {};

   // Installation Interface
   static TimeProviderTestSingleton overrideSoleInstance() {
      // We could save the real instance first, but we won't!
      soleInstance = new TimeProviderTestSingleton();
      return (TimeProviderTestSingleton) soleInstance;
   }

   // Configuration Interface used by the test
```

[8] A private method cannot be seen or overridden by a subclass.

[9] This choice prevents a subclass from overriding the method's behavior.

```
   public void setTime(int hours, int minutes) {
      myTime.set(Calendar.HOUR_OF_DAY, hours);
      myTime.set(Calendar.MINUTE, minutes);
   }

   // Usage Interface used by the client
   public Calendar getTime() {
      return myTime;
   }
}
```

Here the *Test Double* is a subclass of the real component and has overridden the instance method called by the clients of the Singleton.

Example: Behavior-Exposing Subclass

Suppose we wanted to test the `getTime` method directly. Because `getTime` is `protected` and our test is in a different package from the `TimeDisplay` class, our test cannot call this method. We could try making our test a subclass of `TimeDisplay` or we could put it into the same package as `TimeDisplay`. Unfortunately, both of these solutions come with baggage and may not always be possible.

A more general solution is to expose the behavior using a *Behavior-Exposing Subclass*. We can do so by defining a *Test-Specific Subclass* and adding a `public` method that calls this method.

```
public class TimeDisplayBehaviorExposingTss extends TimeDisplay {

   public Calendar callGetTime() {
      return super.getTime();
   }
}
```

We can now write the test using the *Behavior-Exposing Subclass* as follows:

```
   public void testGetTime_default() {
      // create SUT
      TimeDisplayBehaviorExposingTss tsSut =
               new TimeDisplayBehaviorExposingTss();
      // exercise SUT
      //  want to do
      //    Calendar time = sut.getTime();
      //  have to do
      Calendar time = tsSut.callGetTime();
      // verify outcome
      assertEquals( defaultTime, time );
   }
```

Example: Defining Test-Specific Equality (Behavior-Modifying Subclass)

Here is an example of a very simple test that fails because the object we pass to `assertEquals` does not implement test-specific equality. That is, the default `equals` method returns `false` even though our test considers the two objects to be equals.

```
   protected void setUp() throws Exception {
      oneOutboundFlight = findOneOutboundFlightDto();
   }

   public void testGetFlights_OneFlight() throws Exception {
      // Exercise System
      List flights = facade.getFlightsByOriginAirport(
                     oneOutboundFlight.getOriginAirportId());
      // Verify Outcome
      assertEquals("Flights at origin - number of flights: ",
                   1,
                   flights.size());
      FlightDto actualFlightDto = (FlightDto)flights.get(0);
      assertEquals("Flight DTOs at origin",
                   oneOutboundFlight,
                   actualFlightDto);
   }
```

One option is to write a *Custom Assertion* (page 474). Another option is to use a *Test-Specific Subclass* to add a more appropriate definition of equality for our test purposes alone. We can change our fixture setup code slightly to create the *Test-Specific Subclass* as our *Expected Object* (see *State Verification*).

```
   private FlightDtoTss oneOutboundFlight;

   private FlightDtoTss findOneOutboundFlightDto() {
      FlightDto realDto = helper.findOneOutboundFlightDto();
      return new FlightDtoTss(realDto) ;
   }
```

Finally, we implement the *Test-Specific Subclass* by copying and comparing only those fields that we want to use for our test-specific equality.

```
public class FlightDtoTss extends FlightDto {
   public FlightDtoTss(FlightDto realDto) {
      this.destAirportId = realDto.getDestinationAirportId();
      this.equipmentType = realDto.getEquipmentType();
      this.flightNumber = realDto.getFlightNumber();
      this.originAirportId = realDto.getOriginAirportId();
   }
   public boolean equals(Object obj) {
      FlightDto otherDto = (FlightDto) obj;
      if (otherDto == null) return false;
      if (otherDto.getDestAirportId()!= this.destAirportId)
         return false;
      if (otherDto.getOriginAirportId()!= this.originAirportId)
         return false;
      if (otherDto.getFlightNumber()!= this.flightNumber)
         return false;
      if (otherDto.getEquipmentType() != this.equipmentType )
         return false;
      return true;
   }
}
```

In this case we copied the fields from the real DTO into our *Test-Specific Subclass*, but we could just as easily have used the *Test-Specific Subclass* as a wrapper for the real DTO. There are other ways we could have created the *Test-Specific Subclass*; the only real limit is our imagination.

This example also assumes that we have a reasonable `toString` implementation on our base class that prints out the values of the fields being compared. It is needed because `assertEquals` will use that implementation when the `equals` method returns `false`. Otherwise, we will have no idea of why the objects are considered unequal.

Example: State-Exposing Subclass

Suppose we have the following test, which requires a `Flight` to be in a particular state:

```
   protected void setUp() throws Exception {
      super.setUp();
      scheduledFlight = createScheduledFlight();
   }

   Flight createScheduledFlight() throws InvalidRequestException{
      Flight newFlight = new Flight();
      newFlight.schedule();
      return newFlight;
   }

   public void testDeschedule_shouldEndUpInUnscheduleState()
                  throws Exception {
      scheduledFlight.deschedule();
      assertTrue("isUnsched", scheduledFlight.isUnscheduled());
   }
```

Setting up the fixture for this test requires us to call the method `schedule` on the flight:

```
public class Flight{
   protected FlightState currentState = new UnscheduledState();

   /**
    * Transitions the Flight from the <code>unscheduled</code>
    * state to the <code>scheduled</code> state.
    * @throws InvalidRequestException when an invalid state
    *          transition is requested
    */
   public void schedule() throws InvalidRequestException{
      currentState.schedule();
   }
}
```

The `Flight` class uses the State [GOF] pattern and delegates handling of the `schedule` method to whatever State object is currently referenced by `currentState`. This test will fail during fixture setup if `schedule` does not work yet on the default content of `currentState`. We can avoid this problem by using a *State-Exposing Subclass* that provides a method to move directly into the state, thereby making this an *Independent Test* (see page 42).

```
public class FlightTss extends Flight {

   public void becomeScheduled() {
      currentState = new ScheduledState();
   }
}
```

By introducing a new method `becomeScheduled` on the *Test-Specific Subclass*, we ensure that we will not accidentally override any existing behavior of the SUT. Now all we have to do is instantiate the *Test-Specific Subclass* in our test instead of the base class by modifying our *Creation Method* (page 415).

```
   Flight createScheduledFlight() throws InvalidRequestException{
      FlightTss newFlight = new FlightTss();
      newFlight.becomeScheduled();
      return newFlight;
   }
```

Note how we still declare that we are returning an instance of the `Flight` class when we are, in fact, returning an instance of the *Test-Specific Subclass* that has the additional method.