The Addison-Wesley Signature Series

A Martin Fowler Signature Book

xUnit Test Patterns

Refactoring Test Code

Gerard Meszaros

Foreword by Martin Fowler

List of Patterns

Assertion Message (370): We include a descriptive string argument in each call to an Assertion Method.

Assertion Method (362): We call a utility method to evaluate whether an expected outcome has been achieved.

Automated Teardown (503): We keep track of all resources that are created in a test and automatically destroy/free them during teardown.

Back Door Manipulation (327): We set up the test fixture or verify the outcome by going through a back door (such as direct database access).

Behavior Verification (468): We capture the indirect outputs of the system under test (SUT) as they occur and compare them to the expected behavior.

Chained Tests (454): We let the other tests in a test suite set up the test fixture.

Configurable Test Double (558): We configure a reusable Test Double with the values to be returned or verified during the fixture setup phase of a test.

Creation Method (415): We set up the test fixture by calling methods that hide the mechanics of building ready-to-use objects behind Intent-Revealing Names.

Custom Assertion (474): We create a purpose-built Assertion Method that compares only those attributes of the object that define test-specific equality.

Data-Driven Test (288): We store all the information needed for each test in a data file and write an interpreter that reads the file and executes the tests.

Database Sandbox (650): We provide a separate test database for each developer or tester.

Delegated Setup (411): Each test creates its own Fresh Fixture by calling Creation Methods from within the Test Methods.

Delta Assertion (485): We specify assertions based on differences between the pre- and post-exercise state of the SUT.

Dependency Injection (678): The client provides the depended-on object to the SUT.

Dependency Lookup (686): The SUT asks another object to return the depended-on object before it uses it.

Derived Value (718): We use expressions to calculate values that can be derived from other values.

Dummy Object (728): We pass an object that has no implementation as an argument of a method called on the SUT.

Fake Object (551): We replace a component that the SUT depends on with a much lighter-weight implementation.

Four-Phase Test (358): We structure each test with four distinct parts executed in sequence.

Fresh Fixture (311): Each test constructs its own brand-new test fixture for its own private use.

Garbage-Collected Teardown (500): We let the garbage collection mechanism provided by the programming language clean up after our test.

Generated Value (723): We generate a suitable value each time the test is run.

Guard Assertion (490): We replace an `if` statement in a test with an assertion that fails the test if not satisfied.

Hard-Coded Test Double (568): We build the Test Double by hard-coding the return values and/or expected calls.

Humble Object (695): We extract the logic into a separate, easy-to-test component that is decoupled from its environment.

Implicit Setup (424): We build the test fixture common to several tests in the `setUp` method.

Implicit Teardown (516): The Test Automation Framework calls our clean up logic in the `tearDown` method after every Test Method.

In-line Setup (408): Each Test Method creates its own Fresh Fixture by calling the appropriate constructor methods to build exactly the test fixture it requires.

In-line Teardown (509): We include teardown logic at the end of the Test Method immediately after the result verification.

Layer Test (337): We can write separate tests for each layer of the layered architecture.

Lazy Setup (435): We use Lazy Initialization of the fixture to create it in the first test that needs it.

Literal Value (714): We use literal constants for object attributes and assertions.

Minimal Fixture (302): We use the smallest and simplest fixture possible for each test.

Mock Object (544): We replace an object the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.

Root Cause

Equality Pollution is caused by a lack of awareness of the concept of test-specific equality. Some early versions of dynamic *Mock Object* (page 544) generation tools forced us to use the SUT's definition of `equals`, which led to *Equality Pollution*.

Possible Solution

When a test requires test-specific equality, we should use a *Custom Assertion* (page 474) instead of modifying the `equals` method just so that we can use a built-in *Equality Assertion* (see *Assertion Method* on page 362).

When using dynamic *Mock Object* generation tools, we should use a Comparator [WWW] rather than relying on the `equals` method supplied by the SUT. We can also implement the `equals` method on a *Test-Specific Subclass* of an *Expected Object* (see *State Verification* on page 462) to avoid adding it to a production class directly.

Further Reading

For Tests Only and *Equality Pollution* were first introduced in a paper at XP2001 called "Refactoring Test Code" [RTC].

Chapter 16

Behavior Smells

Smells in This Chapter

Assertion Roulette

It is hard to tell which of several assertions within the same test method caused a test failure.

Symptoms

A test fails. Upon examining the output of the *Test Runner* (page 377), we cannot determine exactly which assertion failed.

Impact

When a test fails during an automated Integration Build [SCM], it may be hard to tell exactly which assertion failed. If the problem cannot be reproduced on a developer's machine (as may be the case if the problem is caused by environmental issues or *Resource Optimism; see Erratic Test* on page 228) fixing the problem may be difficult and time-consuming.

Causes

Cause: Eager Test

A single test verifies too much functionality.

Symptoms

A test exercises several methods of the SUT or calls the same method several times interspersed with fixture setup logic and assertions.

```
public void testFlightMileage_asKm2() throws Exception {
   // set up fixture
   // exercise constructor
   Flight newFlight = new Flight(validFlightNumber);
   // verify constructed object
   assertEquals(validFlightNumber, newFlight.number);
   assertEquals("", newFlight.airlineCode);
   assertNull(newFlight.airline);
   // set up mileage
   newFlight.setMileage(1122);
   // exercise mileage translator
   int actualKilometres = newFlight.getMileageAsKm();
   // verify results
   int expectedKilometres = 1810;
   assertEquals( expectedKilometres, actualKilometres);
```

```
    // now try it with a canceled flight
    newFlight.cancel();
    try {
       newFlight.getMileageAsKm();
       fail("Expected exception");
    } catch (InvalidRequestException e) {
       assertEquals( "Cannot get cancelled flight mileage",
                     e.getMessage());
    }
}
```

Another possible symptom is that the test automater wants to modify the *Test Automation Framework* (page 298) to keep going after an assertion has failed so that the rest of the assertions can be executed.

Root Cause

An *Eager Test* is often caused by trying to minimize the number of unit tests (whether consciously or unconsciously) by verifying many test conditions in a single *Test Method* (page 348). While this is a good practice for manually executed tests that have "liveware" interpreting the results and adjusting the tests in real time, it just doesn't work very well for *Fully Automated Tests* (see page 26).

Another common cause of *Eager Tests* is using xUnit to automate customer tests that require many steps, thereby verifying many aspects of the SUT in each test. These tests are necessarily longer than unit tests but care should be taken to keep them as short as possible (but no shorter!).

Possible Solution

For unit tests, we break up the test into a suite of *Single-Condition Tests* (see page 45) by teasing apart the *Eager Test*. It may be possible to do so by using one or more Extract Method [Fowler] refactorings to pull out independent pieces into their own *Test Methods*. Sometimes it is easier to clone the test once for each test condition and then clean up each *Test Method* by removing any code that is not required for that particular test conditions. Any code required to set up the fixture or put the SUT into the correct starting state can be extracted into a *Creation Method* (page 415). A good IDE or compiler will then help us determine which variables are no longer being used.

If we are automating customer tests using xUnit, and this effort has resulted in many steps in each test because the work flows require complex fixture setup, we could consider using some other way to set up the fixture for the latter parts of the test. If we can use *Back Door Setup* (see *Back Door Manipulation* on page 327) to create the fixture for the last part of the test independently of the

first part, we can break one test into two, thereby improving our *Defect Localization* (see *Goals of Test Automation*). We should repeat this process as many times as it takes to make the tests short enough to be readable at a single glance and to *Communicate Intent* (see page 41) clearly.

Assertion Roulette

Cause: Missing Assertion Message

Symptoms

A test fails. Upon examining the output of the *Test Runner*, we cannot determine exactly which assertion failed.

Root Cause

This problem is caused by the use of *Assertion Method* (page 362) calls with identical or missing *Assertion Messages* (page 370). It is most commonly encountered when running tests using a *Command-Line Test Runner* (see *Test Runner*) or a *Test Runner* that is not integrated with the program text editor or development environment.

In the following test, we have a number of *Equality Assertions* (see *Assertion Method*):

```
public void testInvoice_addLineItem7() {
   LineItem expItem = new LineItem(inv, product, QUANTITY);
   // Exercise
   inv.addItemQuantity(product, QUANTITY);
   // Verify
   List lineItems = inv.getLineItems();
   LineItem actual = (LineItem)lineItems.get(0);
   assertEquals(expItem.getInv(), actual.getInv());
   assertEquals(expItem.getProd(), actual.getProd());
   assertEquals(expItem.getQuantity(), actual.getQuantity());
}
```

When an assertion fails, will we know which one it was? An *Equality Assertion* typically prints out both the expected and the actual values—but it may prove difficult to tell which assertion failed if the expected values are similar or print out cryptically. A good rule of thumb is to include at least a minimal *Assertion Message* whenever we have more than one call to the same kind of *Assertion Method*.

Possible Solution

If the problem occurred while we were running a test using a *Graphical Test Runner* (see *Test Runner*) with IDE integration, we should be able to click on the appropriate line in the stack traceback to have the IDE highlight the failed

assertion. Failing this, we can turn on the debugger and single-step through the test to see which assertion statement fails.

If the problem occurred while we were running a test using a *Command-Line Test Runner*, we can try running the test from a *Graphical Test Runner* with IDE integration to determine the offending assertion. If that doesn't work, we may have to resort to using line numbers (if available) or apply a process of elimination to deduce which of the assertions it couldn't be to narrow down the possibilities. Of course, we could just bite the bullet and add a unique *Assertion Message* (even just a number!) to each call to an *Assertion Method*.

Further Reading

Assertion Roulette and *Eager Test* were first described in a paper presented at XP2001 called "Refactoring Test Code" [RTC].