

XUnit Basics

Abstract

Revision: 1.2 Date: 2004/11/04 05:44:53

Preface

This text is an excerpt from an upcoming book on patterns of XUnit test automation. It consists of an introductory narrative that introduces the patterns and the corresponding patterns. I've tried organizing the patterns in roughly the order they are encountered in the narrative. Let me know if you would prefer alphabetical order. Only the patterns listed in the Table of Contents are included in the output file for review. Other patterns are available at <http://xunitpatterns.com>.

Notation

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

Table of Contents

[Introductory Narrative](#)

The patterns:

[Test Automation Framework](#)

[Test Runner](#)

[Test Method](#)

[--Four Phase Test](#)

[--Simple Success Test](#)

[--Expected Exception Test](#)

[--Constructor Test](#)

[Testcase Class](#)

[Testcase Object](#)

[Test Suite Object](#)

[Test Suite Factory](#)

[Assertion Message](#)

[Assertion Method](#)

[Automatic Test Method Discovery](#)

[Manual Test Method Discovery](#)

[Test Suite Procedure](#)

Introductory Narrative

[Back to top](#)

XUnit Basics

Revision: 1.15 Date: 2004/11/04 06:09:42

XUnit Basics

Notes to reviewers: This chapter is meant to introduce the XUnit terminology used throughout the book and to explain how the framework operates beneath the covers. These patterns will eventually be rewritten in the new format so please focus on the content rather than the wording.

XUnit is a family of [Test Automation Frameworks](#), one for each programming language to which it has been ported. [?Hand-Scripted Tests?](#) are usually automated using the same programming language as that used for building the [SUT](#). Although this is not strictly necessary, it is usually much easier because your tests have easy access to the [SUT](#) API. By using a programming language with which the developers are familiar, the effort of learning how to automate [Self-Checking Tests](#) is reduced.

See [?Testing Stored Procs with Junit?](#) for an example of using a testing framework in one language to test an [SUT](#) in another language.

Since most members of the XUnit family are implemented using object-oriented programming languages (OOPL), we'll describe them first and then note where the non-OOPL members of the family differ.

All the members of the XUnit family implement a basic set of features. They all provide a way to:

- specify a test as a [Test Method](#),
- specify the expected results within the test method in the form of [Assertion Methods](#),
- aggregate the tests into [test suites](#) and
- run one or more tests to get a report on the results of the [test run](#).

Many members of the family support [Automatic Test Method Discovery](#) so that you don't have to manually add each test method you want to run to a test suite([Manual Test Method Discovery](#).)

Defining Tests

Each test is represented by a [Test Method](#) that implements a single [Four Phase Test](#) by:

- setting up the [test fixture](#) using either [Inline Setup](#), [Delegated Setup](#) or [Implicit Setup](#),
- exercising the [SUT](#) by interacting with it,
- verifying that the expected outcome has occurred using calls to [Assertion Methods](#) and
- tearing down the [test fixture](#) using either [Inline Teardown](#), [Implicit Teardown](#) or [Garbage-Collected Teardown](#).

The most common types of tests are [Simple Success Test](#), [Expected Exception Test](#) and [Constructor Test](#).

The [Test Methods](#) need to live somewhere so we define them as methods of a [TestCase Class](#); we pass the name of the [TestCase Class](#) to the [Test Runner](#) to run our tests.

It is **possible** to use XUnit without any further understanding of how the [Test Automation Framework](#) operates but that is likely to lead to confusion when building and reusing test fixtures. It is much better to understand how XUnit instantiates and runs the [Test Methods](#).

In most members of the XUnit family, each test is represented by a [Testcase Object](#) because it is a lot easier to manipulate tests if they are [?first class?](#) objects. The [Testcase Objects](#) can be aggregated into [test suites](#) that can be used to run many tests with a single user action.

Defining Suites of Tests

By convention, each [Testcase Class](#) acts as a [Test Suite Factory](#) by providing a [class method](#) called [suite](#) that returns a [Test Suite Object](#) containing one [Testcase Object](#) for each [Test Method](#) in the class. In languages that support some form of reflection, XUnit may use [Automatic Test Method Discovery](#) but in other cases it may require the [test automater](#) to implement [Manual Test Method Discovery](#) (which is more likely to lead to [Lost Tests](#)).

[Test Suite Objects](#) implement the same [Standard Test Interface](#) that [Testcase Objects](#) implement. That interface (implicit in languages lacking a type or interface construct) requires provision of a [run](#) method. The expectation is that when [run](#) is invoked, all of the tests *contained* in the receiver will be run. In the case of a [Testcase Object](#), it is itself a test and will be run. In the case of a [Test Suite Object](#), that means invoking [run](#) on all of the [Testcase Objects](#) it contains. The value of defining a [Standard Test Interface](#) is that it makes running one or running many tests exactly the same. This is a classic example of a [?Composite?](#) object. This is particularly useful in the implementation of [Test Runners](#) to assist in the selection and execution of tests.

We often want a way to run all the tests for an entire [SUT](#) and yet be able to run just the tests for subsets of the functionality. To do this, we can define the overall [test suite](#) for the [SUT](#) as an aggregate of several smaller [test suites](#). This is done by defining a [Test Suite Factory](#) object whose suite method returns a [Test Suite Object](#) containing all the [Test Suite Objects](#) to run. This collection of test suites into larger and larger aggregate suites is commonly used as a way to include the unit test suite for a class into the suite for the module which is in turn included in the suite for the entire system. This hierarchical organization supports the running of tests at many levels.

Running Tests

Tests are run by using one of a variety of [Test Runners](#). A [Graphical Test Runner](#) provides a visual way for the user to specify, invoke and observe the results of running a [test suite](#). The [Test Runner](#) typically provides a way for the user to type in or select a [Test Suite Factory](#). Some also provide a graphical [Test Tree Explorer](#) that can be used to select a specific [Testcase Object](#) to execute from within a [Test Suite Object](#).

A [Command-Line Test Runner](#) can be used to run [tests](#) when running the [test suite](#) from the command line. The name of the [Test Suite Factory](#) to be used to create the [Test Suite Object](#) is included as a command line parameter. [Command-Line Test Runners](#) are most commonly used when invoking the [Test Runner](#) from build scripts or sometimes from within an [?IDE?](#).

The [Test Runner](#) calls the suite method of the [Test Suite Factory](#) to get a [Test Suite Object](#). It then calls the run method via the [Standard Test Interface](#). The run method of a [Testcase Object](#) executes the specific [Test Method](#) for which it was instantiated and reports whether it passed or failed. The run method of a [Test Suite Object](#) iterates over all the members of the collection of tests keeping track of which ones passed and which ones failed.

Test Results

Naturally, the main reason for running automated tests is to determine the results. For the results to be meaningful, we need a standard way to describe them. In general, members of the XUnit family follow the principle of [?Silent Success; Noisy Failure?](#). "No news is good news"; the tests will "call you" when there is a problem.

Test results are classified into one of three categories, each of which is treated slightly differently. When a test runs without any errors or failures, it is considered to be successful. In general, XUnit does not do anything special for successful tests as there should be no need to examine any output when a [Self-Checking Test](#) passes.

A test is considered to have failed when an [assertion](#) fails. That is, the test asserts that something should be true by calling an [Assertion Method](#) and it turns out not to be the case. When it fails, an [Assertion Method](#) throws an assertion failure exception (or whatever facimile the language supports.) The [Test Automation Framework](#) increments a counter for each failure and adds the failure details to a list of failures which can be examined after the

test run is complete. The failure of a single test, while significant, does not prevent the remaining test from being run.

A test is considered to have an error when either the [SUT](#) or the test itself fails in an unexpected way. Depending on the language being used, this could be an uncaught exception, a raised error, etc. As with assertion failures, the [Test Automation Framework](#) increments a counter for each error and adds the error details to a list of errors which can be examined after the test run is complete.

For each [test error](#) or [test failure](#), XUnit records information that can be examined to help understand exactly what went wrong. As a minimum, the name of the [Test Method](#) and [Testcase Class](#) are recorded along with the nature of the problem (whether it was a failed [assertion](#) or a software error).

XUnit in the Procedural World

In the absence of objects or classes, [Test Methods](#) must be treated as global procedures. These are typically stored in modules (or whatever modularity mechanism the language supports). If the language supports storing procedure variables in arrays, the [Test Methods](#) can be aggregated into suites using [Manual Test Method Discovery](#). Otherwise, [test suites](#) are defined by writing [Test Suite Procedures](#) that call [Test Methods](#) and/or other [Test Suite Procedures](#). Tests may be initiated by defining a main method on the module.

[Back to top](#)

Test Automation Framework

Revision: 1.4 Date: 2004/11/04 05:44:53

Build a Test Automation Framework that provides all the mechanisms needed to run the test logic so only the test-specific logic needs to be provided by the test writer.

Context

You are building [?Hand-Scripted Tests?](#) in a programming or scripting language.

Problem

How do you make it easy to write and run tests?

Forces

- There is a lot of overhead in automating tests. This makes automating them very time consuming.

- Much of the overhead is common to every test. Including it in each test results in a lot of [?Test Code Duplication?](#) that increases test maintenance effort.
- All this duplication also results in [?Obscurity Through Verbosity?](#) because all the boilerplate swamps the important test-specific logic.
- Intent Revealing [assertions](#) make tests much more readable.
- It takes time and effort to build a *Test Automation Framework* from scratch.
- It takes time and effort to learn how to use an existing *Test Automation Framework* but probably a lot less than building one.
- Once learned, the use of a *Test Automation Framework* is relatively easy and can make test automation a much more cost-effective exercise.

Therefore ...

Find or build a *Test Automation Framework* that provides all the mechanisms needed to run the test logic so only the test-specific logic needs to be provided by the test writer.

The basic features that should be supported by the *Test Automation Framework* include:

- A way to define [test cases](#) as [Test Methods](#).
- A way to aggregate [Test Methods](#) into [test suites](#).
- A way to run a [test suite](#) and provide statistics about how many tests passed and failed.

Implementation Notes

There is not much to say about learning to use an existing *Test Automation Framework* other than to read the documentation and work through the examples. And if you are reading this we've already established that you have the motivation! Some frameworks are easier to learn than others. For example, SUnit is essentially four classes. For a Smalltalker, understanding the basics of SUnit is not very difficult and shouldn't take very long.

Can we say the same about VbUnit, CUnit, or another non-oo XUnit?

The approach to building a *Test Automation Framework* varies depending on the kind of programming language you are using.

In an object-oriented programming language, you'll naturally make [test cases](#) be [Testcase Objects](#). You'll find yourself encoding them as [Test Methods](#) on [Testcase Classes](#). [Test suites](#) will also be objects so you'll likely have a [?Test Suite Class?](#).

You'll want to have a [Test Runner](#) to run the tests and [Assertion Methods](#) to cause them to fail. And if you build a [?Graphical Test Runner?](#) you may also want to build a [?Test Tree Explorer?](#) to allow you to pick out individual tests to run.

Known Uses

Some of the more popular examples of *Test Automation Framework* are JUnit (Java), SUnit (Smalltalk), CppUnit (C++), NUnit (.Net), RUnit (Ruby), PyUnit (Python) and VbUnit (Visual Basic).

How about technology specific XUnit extentions like HttpUnit, Cactus, etc.?

Related Patterns

Related Smells

This pattern helps avoid [?Test Code Duplication?](#). [Back to top](#)

Test Runner

Revision: 1.4 Date: 2004/11/04 05:44:53

Define a *Test Runner* application that instantiates a [Test Suite Object](#) and executes all the [?Test Objects?](#) it contains.

Context

You are using a [Test Automation Framework](#) to automate [?Hand-Scripted Tests?](#).

Problem

How does a user run the tests?

Forces

- You don't want each test automater to have to provide a special means of running their [test suites](#).
- It would be preferable to be able to run all [test suites](#) the same way so people wanting to run tests only have one interface to learn.
- Most users want to interact with an application through a graphical user interface.
- Some users prefer applications with command-line interfaces.
- Build tools (such as Ant in Java) need a command line interface for running tools.

Therefore ...

Define a *Test Runner* application that instantiates a [Test Suite Object](#) and executes all the [?Test Objects?](#) it contains. Provide two versions of the *Test Runner*:

- a [Graphical Test Runner](#) that allows a user to select a [test suite](#) and watch it execute.
- a [Command-Line Test Runner](#) that can be used by power users and build tools.

Implementation Notes

Both types of *Test Runners* take the name of a [Test Suite Factory](#) and use it to build a [Test Suite Object](#). They then tell it to execute, passing it a [?Collecting Parameter?](#) in which to store the test results.

Known Uses

The website <http://junit.org> describes the use of *Test Runner* in Java.

Variations

Graphical Test Runner

The most common feature of the Graphical Test Runner is some sort of real-time progress indicator. This typically includes a running count of test failures and errors and often includes a colored progress bar which starts off green and turns red as soon as an error or failure is encountered. Some members of the XUnit family include a [?Test Tree Explorer?](#) as part of the [Graphical Test Runner](#).

Should include one or two pictures of GTRs.

Command-Line Test Runner

Need a description of how these typically work and maybe a sample console log.

Related Patterns

[Automatic Test Method Discovery](#) and [Manual Test Method Discovery](#).

Related Smells

When using [Manual Test Method Discovery](#), we are more likely to encounter [Lost Tests](#) because we may forgot to add a new test to the [test suite](#). [Back to top](#)

Test Method

Revision: 1.6 Date: 2004/11/04 05:44:53

Encode each test as a single Test Method on some class.

Context

You are using an object-oriented programming language to code your tests and you don't want to have a separate class for each test.

Problem

How do you organize the code of your test cases?

Forces

same as [Testcase Class](#)

- Using global functions for tests makes it hard to keep track of what test verifies what behavior.
- Creating a separate class for each test creates additional overhead and clutters the class name space. It also makes it harder (though not impossible) to reuse functionality between tests.

Therefore ...

Define each [test](#) as a method, procedure or function. Have each *Test Method* implement the steps necessary to realize a [Fully Automated Test](#). Most notably, the *Test Method* should include [assertions](#) to realize a [Self-Checking Test](#).

Implementation Notes

The *Test Methods* have to be put somewhere. In object oriented programming languages the preferred option is to put them on a suitable [Testcase Class](#).

We still need a way to run all the *Test Methods* tests on the [Testcase Class](#). One solution is to define the run method to call each of the test methods. Of course, we would also have to deal with counting the tests and how many passed and failed. Since all this functionality is needed anyway for a [test suite](#), a simple solution is to instantiate a [Test Suite Object](#) to hold the [tests](#). This requires that we create an instance of the *Test Method* for each [Test Method](#) using the `suite()` method as a [Test Suite Factory](#).

Should we include the following here or let the reader find it in [Test Suite Factory](#) or [Test Suite Object](#)?

We can build the [Test Suite Object](#) manually in the suite method or, if our language supports it, we use reflection to discover all the [Test Methods](#) and add them automatically.

Known Uses

The web site <http://mockobjects.org> describes the use of *Test Method* in Java and lists a number of tools for generating *Test Methods*.

Related Patterns

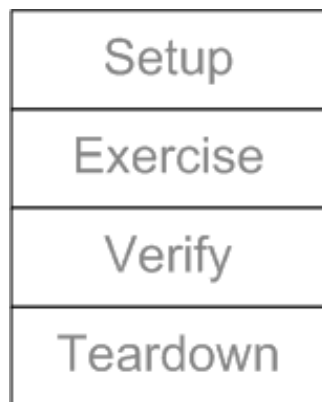
The only real alternative to *Test Method* is [?Data-Driven Test?](#) where tests are encoded as data rather than *Test Methods*.

Related Smells

[Back to top](#)

Four Phase Test

Structure your test with four distinct parts executed in sequence.



Context

You are building [?Hand-Scripted Tests?](#) using a [Test Automation Framework](#) such as XUnit.

Problem

How do you code the tests?
Need to state this more clearly.

Forces

- Many tests require a starting state other than the default state of the [SUT](#).

- Many operations of the [SUT](#) leave it in a different state from that in which they started with.
- It is hard to tell what is being verified if interactions with the [SUT](#) are interspersed with [?outcome verification?](#) logic.
- When one [assertion](#) fails, the rest of the test is not executed. This makes it hard to achieve [Defect Triangulation](#).

Therefore ...

Design each test to have four distinct phases that are executed in sequence. Make the four phases easy to recognize in your test code. The four parts are [?fixture setup?](#), [?exercise the SUT?](#), [?outcome verification?](#) and [?fixture teardown?](#).

- In the first phase, set up the [test fixture](#) (the "before" picture) that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome (such as using a [Test Double](#).)
- In the second phase, interact with the SUT.
- In the third phase, do whatever is necessary to determine whether the expected outcome has been obtained.
- In the fourth phase, teardown the [test fixture](#) to put the world back into the state in which you found it.

Implementation Notes

In the simplest case, each [test](#) is completely free-standing. It does all four phases of the test within the body of the [Test Method](#). This would involve using [Inline Setup](#) and [Inline Teardown](#).

The other main alternative is to take advantage of the [Test Automation Framework's](#) support for [Implicit Setup](#) and [Implicit Teardown](#) outside the body of the [Test Method](#). This involves factoring out common setup and teardown logic into setup and teardown methods on the [Testcase Class](#).

Don't try to test as much functionality as possible in a single test because that can result in [?Eager Test?](#)

Solution Example

Here's an example of a test that is clearly a *Four Phase Test*:

Mostly borrowed from [Simple Success Test](#)

```
public void testSomething() throws Exception {
    // setup
    Integer anArgument = new Integer( 17 );
    // exercise SUT
```

```
        returnValue = sut.doSomething(anArgument);  
        // verify results  
        assertEquals( expectedRV, returnValue);  
        // teardown:  
        anArgument.destroy()  
    }
```

Known Uses

Related Patterns

This pattern helps achieve [Defect Triangulation](#)

Related Smells

[Back to top](#)

Simple Success Test

Setup the fixture, call the SUT and use [Assertion Methods](#) to verify the outcome. Tear down the fixture if necessary. Do not catch any exceptions; let the [Test Automation Framework](#) catch them for you.

Context

You are writing a [Self-Checking Test](#) to verify a success scenario of the SUT (in which you do not expect an exception).

Problem

How do you code the test?

Forces

- There are many possible exceptions that *could* occur while executing the SUT. Catching them all "just in case" would obscure the real expected behavior.
- [Tests as Specification](#) states that the test should describe what *should* occur.

Therefore ...

Define the *Simple Success Test* by coding the four parts of a test. Do *not* include any code to catch any *possible* exceptions or errors as this would convey to the reader that they should occur.

Implementation Notes

Let the [Test Automation Framework](#) catch and report any exceptions. Doing otherwise would result in [?Test Code Duplication?](#) and would mislead the test reader by making it appear that exceptions were expected. See [?Test as Specification?](#) for the rationale behind this.

Motivating Example

The following illustrates a test where the novice [test automater](#) has included code to catch exceptions that he knows might occur (or that he might have encountered while debugging his code.)

```
public void testSomething() throws Exception {
    // setup fixture
    Integer anArgument = new Integer( 17 );
    try {
        // exercise SUT
        returnValue = sut.doSomething(anArgument);
        // verify results
        assertEquals( expectedRV, returnValue);
    } catch( NullPointerException e) {
        fail( e.getMessage());
    } catch( ArrayOverflowException e) {
        fail( e.getMessage());
    }
}
```

Example SimpleSuccessTestOvercomplicated embedded from XUnit Basics/Test Templa

The majority of the code is unnecessary and just obscures the intent of the test.

Refactoring Notes

Luckily for us all this exception handling can be avoided. XUnit has built in support for catching unexpected exceptions. We can rip out all the exception handling code and let the [Test Automation Framework](#) catch any unexpected exception that might be thrown.

Unexpected exceptions are counted as [test errors](#) since the test is terminating in a way we didn't anticipate.

Solution Example

```
public void testSomething() throws Exception {
    // setup fixture
    Integer anArgument = new Integer( 17 );
    // exercise SUT
    returnValue = sut.doSomething(anArgument);
    // verify results
    assertEquals( expectedRV, returnValue);
}
```

Example SimpleSuccessTest embedded from XUnit Basics/Test Templates.java

Known Uses

The JUnit tutorial examples featuring the Money class contain many examples of *Simple Success Test*.

Related Patterns

When the expected result of exercising the SUT is an exception, use an [Expected Exception Test](#). To verify the behavior of object constructors, write a [Constructor Test](#) for each constructor.

Related Smells

This pattern helps avoid [?Obscurity Through Verbosity?](#) and [?Test Code Duplication?](#). [Back to top](#)

Expected Exception Test

Revision: 1.7 Date: 2004/11/04 05:44:53

*Exercise the [SUT](#) in ways that should result in errors and catch **only** the exceptions you expect it to raise.*

Context

You are writing a [Self-Checking Test](#) to verify an error scenario of the [SUT](#).

Problem

How do you code a test to verify that an expected exception is thrown without making the test appear to have terminated with an error?

Forces

- Code that hasn't been tested can be broken accidentally without being noticed.
- If code is worth writing, it is worth testing to prove that it works.
- There are many possible exceptions that *could* occur while executing the SUT.
- [Tests as Specification](#) states that the test should describe what *should* occur.

Therefore ...

Write a [Self-Checking Test](#) for each exception or error that you would expect the [SUT](#) to raise. Raising these exceptions should be part of the requirements you expect the SUT to fulfil. If it is only a *possibility* that these exceptions *might* be raised, do not write tests to verify they are thrown and do not include them in your tests.

Implementation Notes

Set up the fixture to be a scenario where you would expect the [SUT](#) to raise a particular error. Invoke the [SUT](#) inside an error-handling block and if the flow of control continues with the next statement, use a [Single Outcome Assertion](#) to fail the test.

If the error is raised, flow will pass to the error-handling block. This may be enough to let the test pass, but if the class, type, or message contents of the exception or error is important (such as when the error message will be shown to a user), use an [Equality Assertion](#) to verify it.

Exception tests are among the most interesting because of the different ways the XUnit frameworks express them. JUnit provides a special `ExpectedException` class to inherit from but you can only have a single [Test Method](#) per class so it really isn't worth using. NUnit (for .NET) provides a special "ExpectedException" method attribute to tell the [Test Automation Framework](#) to fail the test if that exception **isn't** raised.

Motivating Example

The following is a partially complete test to verify an exception case. The novice [test automater](#) has setup up the right test condition to cause the [SUT](#) to raise an error.

```
public void testSomething() throws Exception {
    // setup fixture
    Integer invalidInput = new Integer( -17 );
    // exercise SUT
    returnValue = sut.doSomething(invalidInput);
    // verify results
    assertEquals( expectedRV, returnValue);
}
```

Example beforeException embedded from XUnit Basics/Test Templates.java

Because the [Test Automation Framework](#) will catch the exception and fail the test, the [Test Runner](#) will not exhibit the [?Green Bar?](#) even though the [SUT's](#) behavior is correct.

Refactoring Notes

Introduce an error-handling block around the exercise phase of the test and use it to invert the pass/fail criteria (pass when the exception *is* thrown and fail when it is not.)

Solution Example

Here's how to verify that the [SUT](#) fails as expected in JUnit:

```
public void testSomething_invalidInput() throws Exception {
    // setup fixture
    String expectedMessage = ... ;
    Integer invalidInput = new Integer( -17 );
    try {
```

```

    // exercise SUT
    returnValue = sut.doSomething(anArgument);
    fail( "Should have thrown InvalidInputException");
} catch( InvalidInputException e) {
    // verify results
    assertEquals( expectedMessage, e.getMsg());
}
}

```

Example ExpectedException embedded from XUnit Basics/Test Templates.java

NUnit provides an attribute that does the same thing without forcing you to code a try/catch block explicitly:

```

[Test]
[ExpectedException(typeof(InvalidOperationException))]
public void PopEmptyStack() {
    Stack stack = new Stack();
    stack.pop();
}

```

Example ExpectedException embedded from XUnit Basics/Test Templates.cs

Smalltalk's SUnit provides another mechanism to achieve the same thing:

```

need to include an example from SUnit using should:raise:
Example ExpectedException embedded from XUnit Basics/Test Templates.st

```

Known Uses

The JUnit tutorial examples featuring the Money class contain many examples of *Expected Exception Test*.

Related Patterns

For [?success scenarios?](#) use an [Simple Success Test](#). To verify the behavior of object constructors, write a [Constructor Test](#) for each constructor.

Binder has a related pattern called [?Catch all Exceptions?](#).

Related Smells

This pattern helps avoid [Untested Requirement](#) by giving us a way to express that an error is expected. [Back to top](#)

Constructor Test

Revision: 1.7 Date: 2004/11/04 05:44:53

Define a special Constructor Test that verifies that object construction was done correctly.

Context

You are verifying the logic of a constructor method of class or the initialization logic of a module. You can predict the value of all attributes (of interest) of the object before the SUT is exercised.

Problem

How do you verify that the initialization is done correctly?

Forces

- Writing a separate *Constructor Test* is extra work.
- Other tests may fail if the object is not constructed properly.
- Having a specific *Constructor Test* provides better [Defect Triangulation](#).

Therefore ...

For each object constructor offered by your class, write a *Constructor Test* that instantiates an object by calling that constructor and verify each attribute using the appropriate [Assertion Method](#).

Variations

Stubbable Initialization Test

When we need to be able to replace a depended-on component with a [Test Double](#), we also need to make sure that the attribute that holds the reference to the [DOC](#) is initialized to the real [DOC](#) when the software is run in production.

A "Stubbable Initialization Test" is a *Constructor Test* that asserts that this attribute is initialized correctly. It is often kept separate from the normal *Constructor Tests* so improve its visibility.

Implementation Notes

Verify each attribute of the object or data structure regardless of whether you expect it to be initialized or not. For attributes that should be initialized, use an [Equality Assertion](#) to specify the correct value. For attributes that should not be initialized, you can use a [Stated Outcome Assertion](#) appropriate to the type of the attribute (e.g., `assertNull(anObjectReference)` for object variables or pointers.)

You may find it useful to use an [Expected Object](#) to specify what the resulting object should look like.

Motivating Example

In this example, we need to build a flight to be able to test the conversion of the flight distance from miles to kilometers. But we'll just make sure the Flight is constructed properly first.

```
public void testFlightMileage_asKm() throws Exception {
    // setup fixture
    BigDecimal flightNumber = new BigDecimal( 1023 )
    Flight newFlight = new Flight(flightNumber);
    assertEquals(flightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    flight.setMileage(1122);
    // exercise SUT
    actualKilometres = flight.getMileageAsKm();
    // verify results
    assertEquals( expectedKilometres, actualKilometres);
}
```

Example EagerTest embedded from XUnit Basics/Test Templates.java

This test is not a [Single Condition Test](#) because it is testing both object construction as well as distance conversion behavior. If object construction fails, we won't know that was the problem until we start debugging.

Refactoring Notes

It would be better to separate this test into two tests each of which is a [Single Condition Test](#). This is most easily done by cloning the [Test Method](#), renaming each copy to reflect what it would do if it were a [Single Condition Test](#) and then removing any code that doesn't satisfy that goal.

Solution Example

```
public void testFlightConstructor() throws Exception {
    // setup fixture
    BigDecimal flightNumber = new BigDecimal( 1023 )

    // exercise SUT
    Flight newFlight = new Flight(flightNumber);

    // verify results
    assertEquals(flightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
}
```

Example ConstructorTest embedded from XUnit Basics/Test Templates.java

Known Uses

Resulting Context

Related Patterns

When the expected result of exercising the SUT is an exception, use an [Expected Exception Test](#). For [?success scenarios?](#) use a [Simple Success Test](#).

Related Smells

A regular test that also verifies object construction as an afterthought is an example of an [?Eager Test?](#). [Back to top](#)

Testcase Class

Revision: 1.8 Date: 2004/11/04 05:44:53

Group a set of related [Test Methods](#) on a single Testcase Class.

Context

You are using an object-oriented programming language to code [?Hand-Scripted Tests?](#) as [Test Methods](#) and you don't want to have a separate class for each test.

Problem

How do you organize the code of your test cases?

Forces

same as [Test Method](#)

- Using global functions for tests makes it hard to keep track of what test verifies what behavior.
- Creating a separate class for each test creates additional overhead and clutters the class name space. It also makes it harder (though not impossible) to reuse functionality between tests.

Therefore ...

Define each [test](#) as [Test Methods](#) on a suitable class. At runtime, create a [Testcase Object](#) for each [Test Method](#) and add them to a [Test Suite Object](#) that the [Test Runner](#) will use to run them all.

Implementation Notes

We still need a way to run all the [Test Methods](#) on the *Testcase Class*. One solution is to define the run method to call each of the test methods. Of course, we would also have to deal with counting the tests and how many passed and failed. Since all this functionality is needed anyway for a [test suite](#), a simple solution is to instantiate a [test suite](#) to hold the [tests](#). This requires that we create an instance of the *Testcase Class* for each [Test Method](#).

We can build the [test suite](#) manually in the suite method or, if our language supports it, we use reflection to discover all the [Test Methods](#) and add them automatically.

Known Uses

The web site <http://mockobjects.org> describes the use of *Testcase Class* in Java and lists a number of tools for generating *Testcase Classes*.

Related Patterns

Testcase Class is also a [Test Suite Factory](#) because it returns a [Test Suite Object](#) containing all its [Testcase Objects](#). It can create its [Test Suite Object](#) using [Automatic Test Method Discovery](#) and or [Manual Test Method Discovery](#).

Related Smells

When using [Manual Test Method Discovery](#), we are more likely to encounter [Lost Tests](#) because we may forgot to add a new test to the [test suite](#). [Back to top](#)

Testcase Object

Revision: 1.7 Date: 2004/11/04 05:44:53

Create a Testcase Object for each test and call the run method when you wish to execute it.

Context

Problem

How do you organize the code of your test cases?

Forces

- Objects are easy to deal with because they can be manipulated more easily than procedures.
- Most programming languages support procedures; many programming languages do not support objects.

same as [Test Method](#)

- Using global functions for tests makes it hard to keep track of what test verifies what behavior.
- Creating a separate class for each test creates additional overhead and clutters the class name space. It also makes it harder (though not impossible) to reuse functionality between tests.

Therefore ...

If your programming language supports objects, instantiate an individual [?Command?](#) object [?\[GOF\]?](#) for each test you want to run and add them to a [Test Suite Object](#) that the [Test Runner](#) will use to run them. Have each *Testcase Object* implement a [Standard Test Interface](#) so that the [Test Runner](#) doesn't need to know the specific interface for each test.

Implementation Notes

Have each *Testcase Object* implement a [Standard Test Interface](#) so that the [Test Runner](#) doesn't need to know the specific interface for each test. In effect, each *Testcase Object* acts as a [?Command?](#) object [?\[GOF\]?](#). This allows us to build collections of these *Testcase Objects* which we can iterate across to do counting, running displaying etc..

In most programming languages, we need to create a class to define the behavior of the *Testcase Objects*. We could create a separate [Testcase Class](#) for each *Testcase Object* but most variants of XUnit host many [Test Methods](#) on a single [Testcase Class](#).

Known Uses

Most variants of XUnit for object-oriented programming languages implement *Testcase Object*.

Related Patterns

Testcase Object is an example of a [?Command?](#) object. The *Testcase Objects* are typically instances of a [Testcase Class](#) with one [Test Method](#) per [test case](#) and are instantiated into a [test suite](#) using either [Automatic Test Method Discovery](#) or [Manual Test Method Discovery](#).

Related Smells

[Back to top](#)

Test Suite Object

Revision: 1.4 Date: 2004/11/04 05:44:53

Define a collection class that implements the [Standard Test Interface](#) and use it to run a set of related [Testcase Objects](#).

Context

You are using an object-oriented programming language to code your [tests](#) as [Test Methods](#) on [Testcase Class](#).

Problem

How do you run more than one test with a single invocation.

Forces

- If each test needs to be invoked separately, you'll run fewer tests.
- If it is easy to run all the tests, you are more likely to run them all.
- It is easier to build the [Test Runner](#) if it doesn't have to know the difference between individual [Testcase Objects](#) and collections of them.

Therefore ...

Define a [?Composite??\[GOF\]?Testcase Objects](#) class that implements the [Standard Test Interface](#). Use it to hold a collection of [Testcase Objects](#) that you want to run as a group. When the standard interface method `run` is called, iterate over all the [Testcase Objects](#) in the collection and call `run` on each [Testcase Object](#).

Implementation Notes

To instantiate the *Test Suite Object*, use a static [?Factory Method?](#) on a [Test Suite Factory](#) class.

Provide an `add` method that can be called by the [?Factory Method?](#) to add another test to the *Test Suite Object*. It shouldn't matter whether the test is a [Testcase Object](#) or another [Test Suite Object](#).

For convenience, some variants of XUnit provide additional `add` methods that take the name of a [Testcase Class](#) and use [Automatic Test Method Discovery](#) to instantiate [Testcase Objects](#) and place them in the suite.

Known Uses

Pretty well all the object-oriented implementations of XUnit include the concept of using a

Test Suite Object to hold and run multiple tests.

Related Patterns

Test Suite Object is an example of [?Composite?](#). The suite of a [Testcase Class](#) returns a *Test Suite Object*.

Related Smells

[Back to top](#)

Test Suite Factory

Revision: 1.5 Date: 2004/11/04 05:44:53

Define a [?Factory?](#) class which implements a class method that returns a [Test Suite Object](#).

Context

You are using an object-oriented programming language to code your tests based on a [Test Automation Framework](#). You run your test by building a [Test Suite Object](#) that contains all the [Testcase Objects](#) you wish to run as a group.

Problem

How do you instantiate your test suite?

Forces

- Having a single [?Factory Method?](#) to call regardless of the class makes it simpler for a user to tell the [Test Runner](#) which [test suite](#) they want to run.
 same as [Test Method](#)
- Creating a separate class for each test creates additional overhead and clutters the class name space. It also makes it harder (though not impossible) to reuse functionality between tests.

Therefore ...

Define a [?Factory?](#) class that implements the suite method. When the suite method is called by the [Test Runner](#), the *Test Suite Factory* builds a [Test Suite Object](#) by adding the appropriate [Testcase Objects](#) and/or [Test Suite Objects](#) to it.

Implementation Notes

Note that while the *Test Suite Factory* builds and returns a [Test Suite Object](#), it doesn't need to subclass a [Test Suite Class](#) or any particular XUnit class. It does need to implement a [Standard Test Suite Factory Interface](#) (typically by providing a [class method](#) called `suite`). While this interface exists in concept, it is rarely defined in the actual class hierarchy because Java does not allow you to define abstract [static](#) methods on interfaces and therefore the variants inspired by JUnit didn't implement one either. (Also, many other programming and scripting languages do not provide a way to define this interface.)

It is possible to define more than one [Factory Method](#) on a *Test Suite Factory* but most [Test Runners](#) only know how to invoke the standard [Factory Method](#) (typically called something like `suite()`).

Known Uses

Pretty well all the object-oriented implementations of XUnit include the concept of using a *Test Suite Factory* as a single name to run multiple tests. The concept is so common that some IDEs (e.g., IBM WebSphere Studio) will create them for you from classes in a project.

Related Patterns

Test Suite Factory is an example of an object [Factory](#). The `suite` method is an example of a [Factory Method](#). By convention, the [AllTests Suite](#) class, which is an example of a *Test Suite Factory* is provided in each test package, module, or other unit of software organization as a way to run all the tests it contains.

Related Smells

Forgetting to include a [Testcase Object](#) in a suite will result in [Lost Test](#).

[Back to top](#)

Assertion Message

Do not AutoFormat; contains PRE tags.

Revision: 1.3 Date: 2004/05/26 05:14:14

For each call to an [Assertion Method](#), include a descriptive string parameter that will be included in the failure message and which will help the test maintainer determine why the test failed.

Sketch goes here.

Assertions are included in tests to make them [self-checking](#). The whole point of them is to report failures. So when a test fails, we want to be able to read the [?test failure report?](#) and have a pretty good idea of at least the symptoms of the problem.

How It Works

Every [Assertion Method](#) takes an optional string parameter that is included in the failure log. Take a moment as you write each assertion and ask yourself what the person reading the failure log would hope to get out of it. As a minimum, the *Assertion Message* should at least be unique within the [Test Method](#) so that the reader is not playing [?Assertion Roulette?](#).

Even better is being able to tell from the message itself (without having to look at the source code of the test) what was being asserted.

When To Use It

Always! The only excuse for not doing this is laziness! If you can't be bothered making a meaningful *Assertion Message*, you have only yourself to blame when you spend too much time in debuggers just trying to determine which assertion was failing.

Variations

Argument Describing Assertion Message

Some types of [Assertion Methods](#) provide less helpful failure messages than others. Among the worst are [Stated Outcome Assertions](#) such as `assertTrue(aBooleanExpression)`. When these fail, all you know is that the stated outcome did not occur. In these cases you should include the expression that was being evaluated (including the actual values) as part of the *Assertion Message* text so that the test maintainer can see from the failure log what was being evaluated and why it caused the test to fail.

Motivating Example

Need to redo these examples

```
    assertTrue( a > b );
    assertTrue( b > c );
Example assertTrueWithoutMessage embedded from ResultVerification/Assertions.java
```

This emits a failure message something like "Assertion Failed". From this, we cannot even tell which of the two *Assertion Messages* has failed. Not very useful, is it?

Refactoring Notes

Fixing this is a simple matter of adding one more parameter to each [Assertion Method](#) call.

Solution Example

In this case, we want to communicate first of all that we are expecting 'a' to be greater than 'b'. But it would also be useful to be able to see what the values of 'a' and 'b' actually were.

```
assertTrue("Expected a > b but a was '" + a.toString() + "' and b was '"
    assertTrue("Expected b > c but b was '" + b.toString() + "' and c was '"
Example assertTrueWithMessage embedded from ResultVerification/Assertions.java
```

This will now result in a useful failure message:

Assertion Failed. Expected a > b but a was '17' and b was '19'.

Of course, this would be even more meaningful if the variables had [Intent Revealing Names!](#)

[Back to top](#)

Assertion Method

Revision: 1.9 Date: 2004/08/08 06:22:20

Call a utility method provided by the [Test Automation Framework](#) (or a [Custom Assertion](#) you may have written yourself) to evaluate whether an expected condition is true.

Context

You are verifying the behaviour of the SUT in a [Self-Checking Test](#).

Problem

How does a test define the expected behavior of a system?

Forces

- Conditional logic makes tests hard to understand.
- Declarative statements that must evaluate to true are easier to understand.
- You don't want each test to know the details of how failures are handled in the [Test Automation Framework](#).
- Intent revealing [assertions](#) make tests much more readable.

Therefore ...

Encode the expected outcome of the test as a series of [assertions](#) that state what should be true

for the test to pass. The [assertions](#) are realized as calls to *Assertion Methods* that encapsulate the mechanism that causes the test to fail. The *Assertion Methods* may be provided by the [Test Automation Framework](#) or by the [test automater](#) as a [Custom Assertions](#).

Implementation Notes

Be sure to call the most appropriate *Assertion Method* to make the test easier to understand and more expressive. While the syntax and naming conventions vary from one member of the XUnit family to the next, most members provide a basic set of assertions that fall into a few basic categories:

- [Single Outcome Assertions](#) such as fail,
- [Stated Outcome Assertions](#) such as `assertNotNull(anObjectReference)` and `assertTrue(aBooleanExpression)`,
- [Equality Assertions](#) that compare two objects or values for equality,
- [Fuzzy Equality Assertions](#) that determine whether two values are "close enough" to each other

Assertion Methods should take an optional [Assertion Message](#) as a text parameter that is included in the output when the assertion fails to allow the user to be able to determine exactly which *Assertion Method* failed and to better explain what should have occurred. The error detected by the test will be much easier to debug if the *Assertion Method* provides more information about why it failed. We call this a [?Diagnostic Assertion?](#) and most of the built-in *Assertion Methods* in most members of the XUnit family implement this pattern.

Variations

Single Outcome Assertion

A Single Outcome Assertion always behaves the same. Since there are only 3 possible outcomes of an XUnit test (success, failure, and error), there are only 3 possible S Outcome Assertions. However, success and error assertion methods are generally not provided. An error assertion makes no sense since an error indicates an unexpected exception. A success assertion is essentially a no-op, but is often used as documentation, especially in the exception handling block of [Expected Exception Test](#). If you want to use success then it is trivially defined as a method with an empty body.

The only Single Outcome Assertion that provides concrete value is fail which causes a test to be treated as a failure.

A Single Outcome Assertion is most commonly used in two circumstances:

- As a [Test Not Implemented Assertion](#) when a test is first identified and implemented as an nearly empty [Test Method](#). By including a call to fail, you can get the [Test Runner](#) to remind you that you still have a test to write.

- As part of a Try/Catch (or equivalent) block in an [Expected Exception Test](#) by including a call to fail in the Try block right after the call that is expected to throw an exception. If you don't want to assert something about the exception that was caught so you have an empty catch block, you can use success to document that you didn't forget to finish the test.

One circumstance in which you should **not** be using Single Outcome Assertions is in [?Conditional Test Logic?](#). There is almost never a good reason to do this and there is usually a more declarative way to do it using other styles of *Assertion Methods* (see [Guard Assertion](#)) that results in tests that are more easily understood and less likely to yield incorrect results.

Need an example.

Stated Outcome Assertion

Stated Outcome Assertions are a way of saying exactly what the outcome should be. The outcome must be common enough to warrant a special *Assertion Method*. The most common examples of this are:

- `assertTrue(aBooleanExpression)` which fails if the expression evaluates to FALSE and
- `assertNotNull(anObjectReference)` which fails if the objectReference doesn't refer to a valid object.

Need to include an example.

Stated Outcome Assertions are often used as [Guard Assertions](#) to avoid [?Conditional Test Logic?](#).

Equality Assertion

These are the most common examples of *Assertion Methods*. They are used to compare the actual outcome with an expected outcome which is expressed in the form of a variable, constant or [Expected Object](#).

Need an example.

Fuzzy Equality Assertion

When one cannot guarantee an exact match due to variations in precision or expected variations in value, it may be appropriate to use a Fuzzy Equality Assertion. Typically, these look just like an [Equality Assertion](#) with the addition of an extra "fuzz" parameter that specifies how close the actual argument must be to the expected one. The most common

example is the comparison of floating point numbers where the limitations of arithmetic precision needs to be accounted for by providing a tolerance (the maximum acceptable distance between the two values.) We have used the same approach when comparing XML documents where direct string comparisons may result in failure due to certain fields having unpredictable content. In this case, the "fuzz" specification is a "comparison schema" that specifies which fields need to match or which ones should be ignored. This particular kind of equality assertion is very similar to asserting that a string conforms to a regular expression or other form of pattern matching.

Need an example.

Diagnostic Assertion

So you've run some tests and an *Assertion Method* has failed. Now what? Of course, you'll need to figure out why it is failing. But before you can even start on that, you must figure out how the actual result varies from what was expected. For this, you want the *Assertion Method* to provide you with useful diagnostic information. Some built-in XUnit *Assertion Methods* already do this by including the expected and actual values in the [?failure report?](#); `assertEquals` is a good example of this. Others are less helpful; `assertTrue` merely states "Assertion failed". To get useful information from these *Assertion Methods* we need to use a [Argument Describing Assertion Message](#).

When writing [Custom Assertions](#), strive to make them self-diagnosing. On one project, we were comparing string variables containing XML. Whenever a test failed, we had to bring up two String inspectors and scroll through them looking for the difference. Finally, we got smart and included the logic in a [Custom Assertion](#) that told us where the first difference between the two XML strings occurred. The small amount of time we spent writing the diagnostic custom assertion was paid back many times over as we ran our tests.

Need an example.

Motivating Example

The following illustrates the kind of code that would be required for each item we wanted to verify if we did not have *Assertion Methods*. All we really want to do is

```
if (x.equals(y)) {
    throw new AssertionFailedException( "expected: <" + x.toString + "> b
} else { // OK, continue
    :
}
```

Example UnsafeChecking embedded from ResultVerification/Assertions.java

but this will cause a `NullPointerException` if `x` is null and it would be hard to distinguish this from an error in the [SUT](#). So we have to put some guard clauses around this so that we always

throw an `AssertionFailedException`:

```
if (x == null) { // Oops, cannot compare them using equals() in Java
    if (y != null ) { // If y is also null, let it go otherwise throw the
        throw new AssertionFailedException( "expected null but found: <" +
    } // else both null so equal
} else if (!x.equals(y)) { // comparable but not equal !
    throw new AssertionFailedException( "expected: <" + x.toString + "> b
} // equal
```

Example `BetterChecking` embedded from `ResultVerification/Assertions.java`

Yikes! That got pretty messy. And we'll have to do this for every attribute we want to verify? This is not good. There must be a better way.

Refactoring Notes

Luckily for us, the inventors of XUnit realized this and have already done the requisite [Extract Method](#) refactoring to create a library of *Assertion Method* that we can call instead. Simply replace the mess of in-line if statements and thrown exceptions with a call to the appropriate *Assertion Method* method.

Solution Example

To compare two objects, use an Equality Assertion. (By convention, the expected value is specified first and the actual value follows it.)

```
assertEquals( x, y);
assertEquals( "abc", firstThreeLetters);
```

Example `EqualityAssertion` embedded from `ResultVerification/Assertions.java`

To compare two floating point numbers (which are rarely ever really equal, specify the acceptable different using a Fuzzy Equality Assertion:

```
assertEquals ( 3.1415, diameter/2/radius, 0.001);
```

Example `FuzzyEqualityAssertion` embedded from `ResultVerification/Assertions.java`

To insist that a particular outcome has occurred, use a Stated Outcome Assertion such as:

```
assertNotNull( x );
assertTrue( a > b );
assertTrue( aBooleanExpression );
```

Example `StatedOutcomeAssertion` embedded from `ResultVerification/Assertions.java`

To fail the test, use the Single Outcome Assertion:

```
fail( "Expected an exception" );
```

Example `SingleOutcomeAssertion` embedded from `ResultVerification/Assertions.java`

`failNotEquals` is a [Test Utility Method](#) that fails the test and provides a [Diagnostic Assertion Message](#)

The example below is the code for the JUnit `assertEquals` method. Notice that although the

intent is the same as the code we wrote, it has been written in terms of guard clauses that identify when things are equal.

```
/**
 * Asserts that two objects are equal. If they are not
 * an AssertionError is thrown with the given message.
 */
static public void assertEquals(Object expected, Object actual) {
    if (expected == null && actual == null)
        return;
    if (expected != null && expected.equals(actual))
        return;
    failNotEquals(expected, actual);
}
```

Example EqualityAssertionImplementation embedded from ResultVerification/Assert

Known Uses

The JUnit tutorial examples featuring the Money class contain many examples of creating the *Assertion Method* ahead of time.

Related Patterns

The main alternative for verifying outputs is [Procedural Behavior Verification](#). The equivalent technique for verifying the behavior (i.e., "side-effects") of the SUT is [Expected Behavior](#). Another way to remove the details of how verification is done is through the use of [Verification Methods](#).

Related Smells

This pattern helps avoid [?Obscurity Through Verbosity?](#). If you don't include a unique enough *Assertion Method*, you can end up playing [?Assertion Roulette?](#) [Back to top](#)

Automatic Test Method Discovery

Revision: 1.3 Date: 2004/04/26 03:37:56

The [Test Automation Framework](#) discovers the [Test Methods](#) automatically using the reflection facilities of a programming language.

Context

You are building a [Test Automation Framework](#) using an programming language that supports reflection.

Problem

How do you build a [Test Suite Object](#) containing all the [Testcase Objects](#) for a [Testcase Class](#)?

Forces

- Adding each [Test Method](#) to the [Test Suite Object](#) takes effort.
- If you forget to add the [Test Method](#) to the [Test Suite Object](#) your test won't get run resulting in a [Lost Test](#).
- Automating the discovery of the [Test Methods](#) requires an understanding of the reflection mechanism provided by the programming language.
- Automating the discovery of the [Test Methods](#) adds complexity to the [Test Automation Framework](#).
- Automating the discovery of the [Test Methods](#) requires a way to identify which methods of a [Testcase Class](#) are actually [Test Methods](#).

Therefore ...

When you have language-level support for reflection, use it to discover all the [Test Methods](#) so that you can create the corresponding [Testcase Objects](#) and add them to the [Test Suite Object](#). The effort involved will be more than compensated by the trouble it saves you. And you'll have fun learning how to use reflection (but don't tell your manager that *we* encouraged you!)

Implementation Notes

There are two basic ways indicate that a method of a [Testcase Class](#) is a [Test Method](#). The more traditional way is the use of a test method naming convention such as "starts with 'test'". The [Test Automation Framework](#) iterates over all the methods of the [Testcase Class](#), selects those that start with the string 'test' (e.g., testCounters) and calls the one-argument constructor to create the [Testcase Object](#) for that [Test Method](#).

The other alternative, used in the .Net languages, is to use a method attribute (e.g. "[Test]" to identify the [Test Method](#).

Motivating Example

The following illustrates the kind of code that would be required for each [Test Method](#) without *Automatic Test Method Discovery*

```
include the code in the suite method required to do manual
                                discovery.
```

Refactoring Notes

Luckily for us users of existing XUnit family members, the inventors of XUnit realized the importance of *Automatic Test Method Discovery* and all we have to do is follow their advice on how to identify our test methods. If they implemented it using a naming convention, we may have to do a [?Rename Method?](#) refactoring refactoring to get XUnit to discover our [Test Method](#). If they implemented method attributes, we just add the appropriate attribute to our [Test Methods](#).

Solution Example

The following example is notable more for the code that is missing rather than that which is present. Note that there is *no* code to add the [Test Methods](#) to the [Test Suite Object](#)!

Replace the following with actual code from JUnit (test name convention) and C# (method attributes).

Known Uses

JUnit, VbUnit, SUnit, PyUnit and RUnit all support *Automatic Test Method Discovery* using the naming convention approach. NUnit uses the method attribute approach.

Related Patterns

Related Smells

This pattern helps avoid [Lost Tests](#).[Back to top](#)

Manual Test Method Discovery

The [test automater](#) manually writes the code that enumerates all the [Test Methods](#) that belong to the [test suite](#).

Context

You are building a [Test Automation Framework](#) using an programming language that does *not* support reflection.

Problem

How do you build a [Test Suite Object](#) containing all the [Testcase Objects](#) for a [Testcase Class](#)? Or, in the procedural world, how do you know which [Test Methods](#) to invoke?

Forces

Same as [Automatic Test Method Discovery](#)

- Adding each [Test Method](#) to the [Test Suite Object](#) takes effort.
- If you forget to add the [Test Method](#) to the [Test Suite Object](#) your test won't get run resulting in a [Lost Test](#).

If your language does not support reflection are the following actually forces?

- Automating the discovery of the [Test Methods](#) requires an understanding of the reflection mechanism provided by the programming language.
- Automating the discovery of the [Test Methods](#) adds complexity to the [Test Automation Framework](#).
- Automating the discovery of the [Test Methods](#) requires a way to identify which methods of a [Testcase Class](#) are actually [Test Methods](#).

Therefore ...

When you do not have language-level support for reflection, manually write the code that enumerates all the [Test Methods](#) that belong to a [test suite](#).

Implementation Notes

If you are programming in an object-oriented programming language and have decided to represent [test cases](#) as [Testcase Objects](#), implement the suite method by instantiating a [Testcase Object](#) for each [Test Method](#) and adding them to the [Test Suite Object](#). Beyond this, everything will be the same as for [Automatic Test Method Discovery](#).

Motivating Example

The following illustrates the kind of code that would be required for each [Test Method](#) without *Manual Test Method Discovery*

```
include the code in the suite method required to do manual
                                discovery.
```

Refactoring Notes

tbd

Solution Example

```
public:
    static CppUnit::Test *suite()
    {
        CppUnit::TestSuite *suite = new CppUnit::TestSuite( "ComplexNumberTest" );
        suite>addTest( new CppUnit::TestCaller<ComplexNumberTest>(
                        "testEquality",
```

```

        &ComplexNumberTest::testEquality ) );
suite>addTest( new CppUnit::TestCaller<ComplexNumberTest>(
        "testAddition",
        &ComplexNumberTest::testAddition ) );
return suite;
}

```

Cite source as CppUnit Documentation.

Known Uses

CppUnit uses *Manual Test Method Discovery* but also includes a number of useful macros to reduce the tedium of building suites. It also provides a mechanism for test suites to register themselves with a TestFactoryRegistry. CUnit uses [Test Suite Procedures](#).

Related Patterns

In the pure procedural world where you cannot treat a [Test Method](#) as an object or data item, you have no choice but to hand-code a [Test Suite Procedure](#) for each [test suite](#). This is a rather different approach in that how tests are invoked is completely different.

Related Smells

Manual Test Method Discovery may result in [Lost Tests](#) if the [test automater](#) forgets to add a newly written test to the [test suite](#). [Back to top](#)

Test Suite Procedure

Define a Test Suite Procedure that is hard-coded to call each Test Method in turn.

Context

You are using a [Test Automation Framework](#) to automate tests using an programming language that does *not* support objects or procedure variables. Therefore, you cannot manipulate them at run-time.

Problem

How do you run more than one test with a single invocation?

Forces

- Adding each [Test Method](#) to the *Test Suite Procedure* takes effort.

- If you forget to add a [Test Method](#) to the *Test Suite Procedure* then that test won't get run resulting in a *Lost Test*.
- Objects are easy to deal with because they can be manipulated more easily than procedures.
- Most programming languages support procedures; many programming languages do not support objects.

Therefore ...

When you do not have language-level support for objects, run multiple tests by calling the [Test Methods](#) directly from a *Test Suite Procedure*. Define a *Test Suite Procedure* for each set of tests that you would like to be able to run as a group.

Implementation Notes

Test Suite Procedures may call other *Test Suite Procedures* to implement [Suite of Suites](#).

Motivating Example

The following illustrates the kind of code that would be required for each [Test Method](#) without *Test Suite Procedure*

example ?

Refactoring Notes

tbd

Solution Example

Need a sample *Test Suite Procedure*.

Known Uses

To be verified: CUnit may use this approach?

Related Patterns

In the object world, we prefer to use a [Test Suite Object](#) constructed by a [Test Suite Factory](#).

Related Smells

This pattern may result in [*Lost Tests*](#) if the [test automater](#) forgets to add a newly written test to the [test suite](#).