A Pattern Language for Setting Up XUnit Test Fixtures

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

Abstract

Automated unit tests (A.K.A. "developer tests") and functional tests (A.K.A. "customer tests") are a cornerstone of many agile development methods (such as eXtreme Programming). The availability of automated, self-checking tests allows developers to be much bolder in how they modify existing software. Almost every test requires some kind of test fixture. The test fixture is everything you need to have in place to run the test. Typically, it consists of objects (in memory or on disk), records in a database, and/or the state of any other depended-on components (or even entire systems or applications). At first glance, the construction of the test fixture would appear to be a rather simple task. But as the size and complexity of the SUT grows, the complexity of managing the fixture grows in proportion. This is particularly true as we move from unit tests for simple entity objects (which have simple state models) or utility objects (which are typically stateless) to component tests for stateful service objects or functional tests for entire applications. This set of patterns describes the key techniques for addressing the issues around test fixture management.

Introduction

Pattern Templates

Note that I am using two different pattern templates in this submission. Many of the patterns are written in a classic highly-structured pattern form. As I am targeting this content for a book, a few of the patterns have been rewritten in a more "prosey" (and therefore “user-friendly) form based on the format used by Martin Fowler in his book “Patterns of Enterprise Application Architecture”. This pattern format is something I would like specific feedback on.

My Pattern Template

The template starts off with the summary statement and a sketch. The summary statement captures the essence of the pattern in one or two sentences and the sketch provides a visual representation of the pattern.

The next (untitled) section summarizes why you might want to use the pattern in just a few sentences. It includes the *Problem* section from the traditional pattern template. You should be able to get a sense of whether you want to read any further by reading this section.

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

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

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

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

Notation:

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

Scope:

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

Fixture Setup

Table of Contents

Patterns are underlined; pattern categories are not. Patterns labelled (*) are included here for review.

Introductory Narrative

Almost every test requires some kind of test fixture. The test fixture is everything you need to have in place to run the test. It is the "before" state of the system; the state in which the system must be before the actual behaviour we are attempting to verify can be induced. Typically, it consists of objects (in memory or on disk), records in a database, and/or the state of any other depended-on components (or even entire systems or applications).

At first glance, the construction of the test fixture would appear to be a rather simple task. Just instantiate the objects, components or records in the right state and be done with it. Right?

This may be true for very simple ?unit tests? that require just one or two objects for their test fixture. In this case, creating the fixture may truly be as simple as instantiating a single object in memory and setting its state explicitly. And if this is all you need, feel free to skip ahead to Building Clean Slate Fixtures later in this chapter.

But as the size and complexity of the SUT grows, the complexity of managing the fixture grows in proportion. This is particularly true as we move from unit tests for simple entity objects (which have simple state models) or utility objects (which are typically stateless) to ?component tests? for stateful service objects that have complex state. In these cases, the effort to create the fixture goes up considerably and the potential for *?Test Code Duplication?* starts to rise dramatically.

As soon as the SUT has persistent state (such as a database), the time it takes to set up the test fixture and execute the tests increases dramatically. This introduces a further constraint into our automated test design: test execution speed. This is particularly true of ?functional tests? for entire applications.

A common reaction to having slow tests is to reuse the same test fixture to reduce the execution time spent building and tearing down the fixture. This is typically done in one of two ways. The most common is to use a Prebuilt Fixture that is set up before any test suites are run. The second is the use of Chained Tests that depend on the order in which tests execute within the test suite to build the fixture up incrementally. Both of these approaches are fraught with perils that make it harder to achieve robust and maintainable tests.

As the complexity of and consequences of fixture management becomes the predominant complexity factor in our tests, how you manage your test fixtures becomes the single most critical decision you will need to make as you adopt a test automation strategy. Because this is such a large and important topic, we will attack it in three phases.

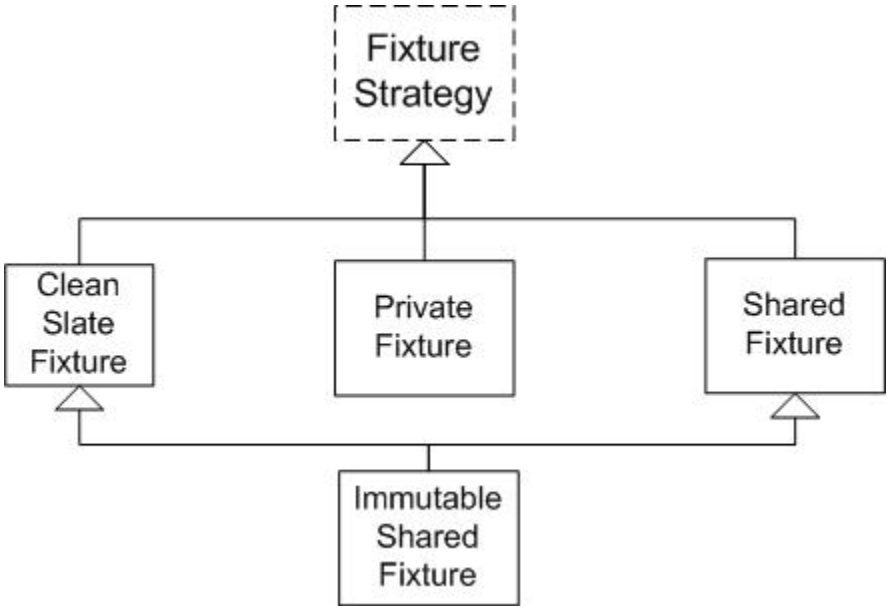
- First, in Fixture Management Decisions, we'll introduce the fundamental decisions we need to make about the test fixture.
- Second, in Test Fixture Strategies we will summarize the two key test fixture management strategies as well as several hybrid strategies.
- Finally, in Building Test Fixtures, we will deal with how you actually construct the test fixture in each of the strategies.

Fixture Management Decisions

(Omitted due to limited space.)

Test Fixture Strategies

As we stated earlier, the choice of test fixture strategy is probably the most far-reaching decision you will have to make as you adopt an automated testing strategy using XUnit. The right strategy could make automated testing a "piece of cake" while choosing the wrong strategy could lead to large amounts of wasted effort and frustration. Forewarned is for-armed!



Clean Slate Testing

In this approach, each test creates a temporary Private Test Fixture as it runs. Any objects or records it requires are created by the test itself. Because the?test fixture visibility? is private to the one test alone, we ensure that each test is completely independent because it cannot depend, either accidentally or on purpose, on the output of any other tests that use the same fixture.

We call this "clean slate" because each test starts with a clean slate and builds from there. It does not "inherit" or "reuse" any part of the fixture from other tests or from a Prebuilt Fixture.

The main disadvantage of Clean Slate Fixture is the additional CPU cycles it takes to create all the objects for each test. This can make the tests run slower than a Prebuilt Fixture approach, especially if the test fixture is constructed within a database.

We feel this is best addressed by applying one or more of the following patterns:

1. Construct a Minimal Test Fixture (the smallest possible fixture you possibly can) and
2. Speed up the construction by using ?Stub Out Slow? to bypass anything that takes too long to construct.
3. Minimize the size of the fixture you need to destroy and reconstruct each time by using an Immutable Shared Test Fixture for any objects that are referenced but not modified.

We have found that, on average, our tests run 50 (yes, fifty!) times faster when we use ?In-Memory Database? (a variation of ?Stub Out Slow Component?) to replace the entire database with a set of HashTables so we can do in-memory test execution. This is because a test doesn't just access the database when it exercises the system, but it also writes records during fixture setup and deletes them during fixture teardown. That's a lot of disk accesses in each test.

There is a lot to be said for minimizing the size and complexity of the test fixture. A Minimal Test Fixture is much easier to understand and helps highlight the "cause and effect" relationship between the fixture and the expected outcome. In this regard, it is a major enabler of Tests as Specification. In some cases, you can make the test fixture much smaller by using Entity Chain Snipping to eliminate the need to instantiate objects on which your test depends only indirectly. This will certainly speed up the instantiation of your test fixture.

Ahead of Time Test Fixture Construction

There will be times when you cannot or choose not to use a Clean Slate Fixture test fixture strategy. In these cases, you can use Prebuilt Fixture. This approach involves pre-building the test fixture and then using it over and over again when you run your tests.

The most common example of Prebuilt Fixture is the ?Test Database?. In this approach, a database is populated with some (possibly standardized) test data and all tests are written assuming this set of records. This approach has the advantage that it executes much faster because there is little or no test fixture creation overhead associated with individual tests. The main disadvantage is that it opens the door to a number of smells, both ?code smells? and ?behaviour smells?. Examples of ?code smells? include *?Test Code Duplication?*, *?Obscurity Through Verbosity?*, *?Mystery Guest?* and *?Complex Teardown?*. Examples of ?behaviour smells? this approach breeds includes *Unrepeatable Test*, *Interacting Tests* and *?Test Run War?*.

The *?Mystery Guest?* problem is caused by the use of magic values that refer to particular objects in the pre-built test fixture. The test reader cannot tell by merely reading the test how the referenced object affects the outcome of the test without consulting the database. This can be partially alleviated by using a variable with an Intent Revealing Name to hold the magic value so that the test reader is given some inkling about why the test uses this particular object. This doesn't avoid *Unrepeatable Tests*, *Interacting Tests* and *?Test Run Wars?* but at least we can understand what the test was intended to do when it fails (Emoticon: smiley) . To avoid a *?Test Run War?*, each test runner must have its own copy of the test fixture. This is most commonly done by using a ?Private Database?; either a ?Private Real Database? or ?Private Virtual Database? . The ?Private Virtual Database? can be implemented using built-in database support (?DB Schema per TestRunner?) or by using a ?Database Partitioning Scheme? (i.e., giving each potential test runner its own set of customers, accounts, etc..)

All these approaches solve the problem by ensuring that tests run from different test runners find different objects even though they are using the same identifier (i.e., the key). To avoid having *Unrepeatable Tests*, you must ensure that each test leaves the pre-built test fixture in the same state as it found it (i.e., treat it as an Immutable Shared Test Fixture) This is easier said than done because it must do so regardless of the outcome of the test. It must understand what the SUT would do to the state of the fixture when the test succeeds and undo it upon successful completion. If the test fails, it must accurately assess what changes the SUT did make and undo those. This can very easily lead to *?Complex Teardown?* which will make the tests very hard to write? *?Data Leaks?* are inevitable and just as hard to debug as ?memory leaks? because the tests that fail as a result are not the tests that are leaving behind the leftover test fixture data.

One way to avoid the *?Complex Teardown?* is to use Automated Teardown. In this technique, anything that is newly created or modified by the test is registered with the Automated Teardown mechanism so that it can automatically clean it up in the teardown method (which is run regardless of whether the test passes or fails.) Because the Automated Teardown mechanism can be a separate component (well, at least a separate method) and not embedded in each Test Method, it can be tested using automated Self-Checking Tests.

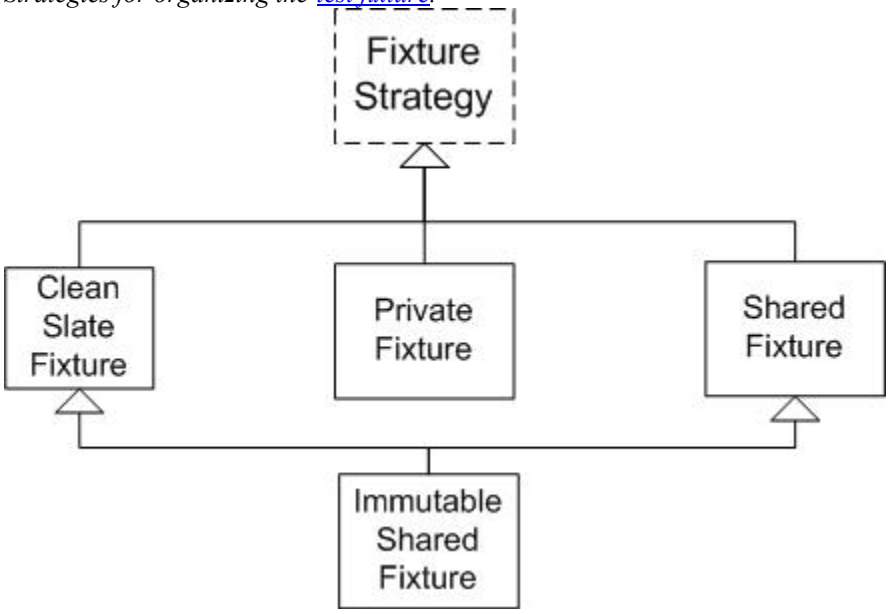
Another problem associated with Prebuilt Fixture is *?Fragile Fixture?*. This occurs when tests fail merely because we've added additional objects/rows to an existing test fixture. This will happen if the tests assume they know what is already in the database (including the assumption that the database is empty) and code their assertions based on that knowledge. As soon as someone extends the fixture to support new tests, any tests that "know" what used to be in that part of the fixture will likely fail. A common strategy for dealing with this is to use ?Before and After Delta Assertions? for verifying the expected results. First, the test queries the results before adding its own objects. Then it adds the objects and queries again. Then it removes the original results from the new results to cancel the impact of any objects/records that already existed. This should handle the straightforward cases but it does increase the complexity of the tests by adding an extra step before the ?test fixture setup? phase of the test. It works best when the newly added test objects are created from scratch because adding existing objects could be affected by changes to the predefined test fixture.

Building Test Fixtures

(Omitted due to limited space.)

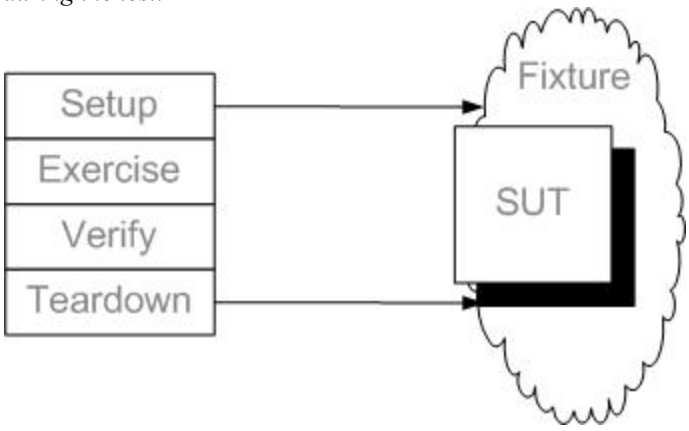
Test Fixture Strategy

Strategies for organizing the test fixture.



Clean Slate Fixture

Make sure each test creates its own private fixture and tears down the fixture and any objects created during the test.



Every test requires some of kind of test fixture that defines the state of the world before it starts to execute the SUT. The test fixture can either be set up in each and every test or they can be "reused" across many tests. In all the but simplest test suites, keeping track of what each test needs in the way of a fixture is complicated and error prone. Even just making sure you don't write *Unrepeatable Tests* is a challenge. And taking into account what other every other test depends on while you are modifying a Shared Test Fixture is not a scalable solution. You are bound to encounter *Interacting Tests* and their shared test environment cousin *?Test Run War?*.

Another key role of the fixture setup logic is to facilitate Tests as Specification. We do this by making the relationship between the test fixture (the preconditions of the test) and the assertions (the expected outcome) very obvious. This enables the test reader to get a good understanding of what the SUT is **expected** to do. Shared Test Fixtures tends to result in *?Mystery Guests?* because they hide the test fixture from the reader.

The solution is to have each test start with a "clean slate" and build its own well-understood Private Test Fixture.

How It Works

When doing "normal" unit testing in a garbage collected environment, *Clean Slate Fixture* is the naturally occurring default behaviour for all tests. You have to try hard to have objects leftover by one test affect another test's outcome.

The problem occurs when the SUT starts to "remember" things between invocations of its API. Once it is "stateful", the likelihood that something is left behind by another test increases. This is particularly true if each test does not (or cannot) create its own instance of the object in question. A good example of this when we use ?Singletons? within our designs. The ?Singleton?, in effect, instantiates itself the first time it is used and hangs around as long as it deems fit. So the tests have lost control over the starting state of the SUT.

The key to the solution is two-fold:

- Have each test build its own fixture, use it, and then clean it up explicitly.
- Define the test fixtures in such a way that it *doesn't matter* if other tests didn't clean up properly.

Both points are equally important and each has its own implementation patterns.

To minimize the effort and maximize the comprehensibility, build a Minimal Test Fixture regardless of whether you use Implicit Setup, Delegated Setup or Inline Setup. For any objects that won't be garbage collected, you can use Automated Teardown to minimize the effort, complexity and the likelihood of test bugs.

To make tests indifferent to other tests messing up the environment, use some kind of ?Database Partitioning Scheme? to give each test its own "virtual address space". One way to achieve this is to use an Anonymous Creation Method to create the "root" object of the fixture (the one all the other objects are associated with) thereby ensuring a different "virtual address space" each time the test is run. When doing outcome verification, only include objects within the private address space. These techniques are particularly important when testing systems with databases underneath them because a shared database instance is the perfect ?Singleton?.

When To Use It

When you are most concerned about test robustness and repeatability, have each test build its own Private Test Fixture from scratch starting with a "clean slate". If, however, you find that *Clean Slate Fixture* results in tests that take too long to execute because of all the extra overhead of setting up and tearing down the test fixture, consider using a Shared Test Fixture, a Prebuilt Fixture or a hybrid approach such as Immutable Shared Test Fixture.

Motivating Example

Come up with a better example.

```
public void testPurchase_firstPurchase() {
   Customer buyer = new Customer(17, "FirstName", "LastName", "G", "ACTIVE");
   ...
}
public void testPurchase_subsequentPurchase() {
   Customer buyer = new Customer(18, "FirstName", "LastName", "G", "ACTIVE");
   ...
}
```

Refactoring Notes

Use Extract Method refactoring to remove the direct call to the constructor and replace the number 17 with a call to a method that returns a value guaranteed to not be the id of any existing Customers.

Solution Example

In the refactored code below, we have extracted the Customer creation code into createAnonymousCustomer

```
public void testInvoice_firstPurchase() {
   Customer buyer = createAnonymousCustomer();
   ...
}
public void testInvoice_firstPurchase_subsequentPurchase() {
   Customer buyer = createAnonymousCustomer();
   ...
}
```

In the implementation of createAnonymousCustomer, we have replaced the Hard-coded Value 17 with a call to getUniqueCustomerId. Different ways of implementing getUniqueCustomerId are discussed in Distinct Value Generation.

```
public Customer createAnonymousCustomer() {
   BigDecimal uniqueid = getUniqueCustomerId()
   return new Customer(uniqueid,"FirstName" + uniqueid, "LastName" + uniqueid);
}
```

Standard Test Fixture

Define a standard fixture (and the procedures to build it) and use it in several tests.

Context

To execute an automated test, you require a text fixture that is well understood and completely deterministic. You want to avoid designing a completely different fixture for every test.

Problem

How do you minimize the effort of designing test fixtures?

Forces

- If other tests have access to the same test fixture, those tests may leave the fixture in an unexpected state which could affect the results of other tests using the same fixture.
- A fixture set up within the test is guaranteed to be there regardless of how the test is run.
- It takes additional design effort to design a specific test fixture for each test.
- A simple test fixture specific to a test is easier to understand than a more general fixture that satisfies the needs of many tests.
- It takes additional computational cycles to build a new test fixture for each test.
- It takes fewer computational cycles to build a simple test fixture for a single test than to build a large general-purpose test fixture.

Therefore ...

Design a *Standard Test Fixture* that can be used by several or many tests. Define a method (function or procedure) that builds the *Standard Test Fixture* and call it from each test.

Implementation Notes

The choice of *Standard Test Fixture* is independent of the choice between Private Test Fixture and Shared Test Fixture because *Standard Test Fixture* focuses on reusing the *design* of the fixture, not when it is built or its visibility. You still need to decide whether each test will build its own instance of the *Standard Test Fixture* (a Private Test Fixture) or whether you will pre-build it as a Shared Test Fixture and reuse it across many tests.

Motivating Example

```
public void testStatus_inital() {
    // setup:
   Flight flight = createAFlight()
   // Exercise SUT & verify outcome:
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}

public void testStatus_cancelled() {
   // setup:
   Flight flight = createAFlight()
   flight.cancel();
   // Exercise SUT & verify outcome:
   assertEquals(FlightState.CANCELLED, flight.state());
   // teardown:
   flight.destroy();
}
```

Refactoring Notes

The most obvious refactoring is from a Private Test Fixture to a *Standard Test Fixture*. Another refactoring is from a Test Method using Inline Setup to *Standard Test Fixture* by using Extract Method refactoring. Extracting the fixture setup into a separate method makes the fixture available to any tests that require it. If all the tests in a Testcase Class require the *Standard Test Fixture*, then Extract Method refactoring can be used to create a Shared Setup Method. Along the way, you will introduce Fixture Holding Instance Variables to hold onto the fixture objects. Within each Test Method, you can refer to the objects in the fixture using the instance variables, or you can access them using Finder Methods. It is not advisable to use Hard-Coded Values to refer to objects in the *Standard Test Fixture* as these will result in *?Mystery Guest?* and *?Fragile Tests?*.

Solution Example

```
Airport departureAirport
Airport destinationAirport
Flight flight

public setup() {
   super.setup();
   departureAirport = new Airport("Calgary", "YYC");
   destinationAirport = new Airport("Toronto", "YYZ");
   flight = new Flight(departureAirport, destinationAirport)
}

public void testStatus_inital() {
   // implicit setup
   // Exercise SUT & verify outcome
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}
public void testStatus_cancelled() {
   // implicit setup partially overriden:
   flight.cancel();
   // exercise SUT & verify outcome
   assertEquals(FlightState.CANCELLED, flight.state());
   // teardown:
   flight.destroy();
}
```


Related Patterns

As stated earlier, the choice of *Standard Test Fixture* is independent of the choice between Private Test Fixture and Shared Test Fixture because *Standard Test Fixture* focuses on reusing the *design* of the fixture, not when it is built or its visibility. This pattern can be at odds with Minimal Test Fixture because the more broadly you plan to share the fixture, the larger it tends to get.

Related Smells

As you make a *Standard Test Fixture* more reusable across many tests, it tends to get larger and more complex. This can lead to *?Fragile Fixture?* as the needs of new tests introduce changes that break existing clients of the *Standard Test Fixture*. Depending on how you go about building the *Standard Test Fixture*, you may also find yourself entertaining a *?Mystery Guest?* if the cause/effect of the SUT behaviour is not easy to discern because the fixture setup is hidden from the test.

Minimal Test Fixture

Use the smallest possible fixture for each test.

Context

To execute an automated test, you require a text fixture that is well understood and completely deterministic. You want to achieve Tests as Specification.

Problem

How do you minimize the effort of designing test fixtures?

Forces

Same as Private Test Fixture unless otherwise noted.

- It takes additional design effort to design a specific test fixture for each test.
- A simple test fixture specific to a test is easier to understand than a more general fixture that satisfies the needs of many tests.
- It takes fewer computational cycles to build a simple test fixture for a single test than to build a large general-purpose test fixture.

Therefore ...

Design a *Minimal Test Fixture* that includes only those objects that are absolutely necessary to express the behaviour that the test verifies. Another way to phrase this is "If the object is not important to the test, it is important not to have it included in the test." This is important for achieving ?Test as Specification? and for avoiding *?Slow Tests?*.

Implementation Notes

To build a *Minimal Test Fixture*, ruthlessly remove anything from the fixture that does not help the test communicate how the SUT should behave. You can hide the attributes of objects that don't affect the outcome of the test but which are needed for constructing the object by using Creation Methods to fill in all the "don't care" attributes with meaningful default values. You can also hide the creation of necessary depended-on objects within the Creation Methods. Or, you can eliminate them entirely by installing a Test Double to do Entity Chain Snipping. This is truly an example of creating a *Minimal Test Fixture*!

Some good "Ocam's Razors" for determining whether an attribute or object is really needed:

- An attribute shouldn't affect the behaviour of the SUT in the specific circumstance, and
- the attribute is not referenced during assertions on other objects.

It is one thing to minimize the size of the test fixture itself. It is also good to minimize the amount of code required to create the test fixture. Creation Methods are a common way of achieving this goal. When

writing tests that require badly formed objects as input (for testing with invalid inputs) a common approach is to use One Bad Attribute.

Refactoring Notes

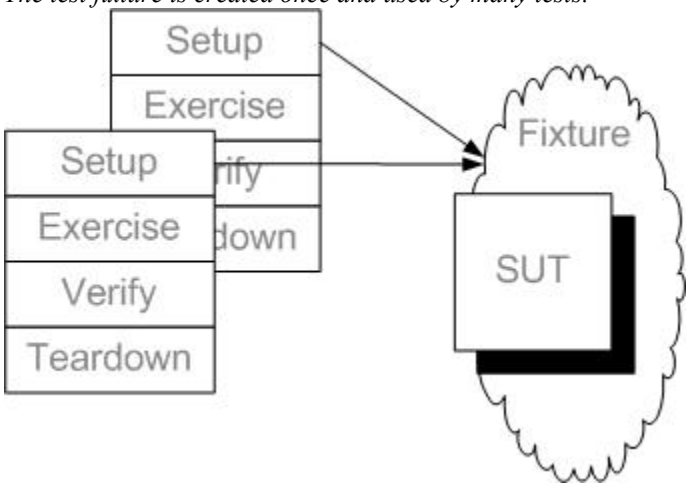
Solution Example

Related Patterns

Related Smells

Shared Test Fixture

The test fixture is created once and used by many tests.



To execute an automated test, you require a text fixture that is well understood and completely deterministic. Your tests are taking too long to execute mostly due to the time it takes to set up a Private Test Fixture in each test.

How do you make your tests run faster?

How It Works

The concept is pretty simple: create a test fixture that outlasts the lifetime of a single Testcase Object. This allows multiple tests to use the same test fixture without destroying it and recreating it between tests.

When To Use It

Use a *Shared Test Fixture* when you cannot afford to use a Clean Slate Fixture. Typically, this is because it takes too long to build a new fixture for each test.

The biggest issue is that a *Shared Test Fixture* can lead to *Interacting Tests* or even a *?Test Run War?* since tests may depend on the outcomes of other tests. Another issue is that a fixture designed to serve many tests is bound to be much more complicated than the Minimal Test Fixture needed for a single test. This will typically take more effort to design and can lead to a *?Fragile Fixture?* down the road when you need to modify it.

A *Shared Test Fixture* will often result in a *?Mystery Guest?* because the fixture is not constructed inside the test. This can be mitigated by the use of Finder Methods to access the relevant parts of the fixture.

Implementation Notes

Among the things you will need to decide on is just how far and wide you want to share the fixture. It is one thing to share the fixture between all the tests in a single Testcase Class and an entirely different thing to share it with any and all users and tests that happen to be running at the same time.

There are two key questions to decide upon:

- Fixture Setup Strategy - How do you construct the *Shared Test Fixture*?
- Shared Fixture Access - How do you access the *Shared Test Fixture*?

This leads to several options for implementing *Shared Test Fixture*.

- SuiteFixture Setup
- Lazy Fixture Initialization
- Prebuilt Fixture

If the fixture will only be shared by tests in the same Testcase Class, you can simply move fixture creation to the ?Shared Setup Method?. If you want to share it across many Testcase Classes, or you have resources that need to be cleaned up, you'll need to use a ?Test Setup Decorator? to hold the setup and teardown methods. A third option is to prebuild the fixture at some point before the tests are run. A common way to do this is with a Data Loader or ?Database Population Script?. The latter is an example of a Back Door Fixture Setup wherein we bypass the SUT and interact with its database directly.

Motivating Example

This example shows a Testcase Class which is setting up the test fixture using Implicit Setup. Each Test Methods uses a Fixture Holding Instance Variable to access the contents of the fixture.

```
Airport departureAirport
Airport destinationAirport
Flight flight

public setup() {
   super.setup();
   departureAirport = new Airport("Calgary", "YYC");
   destinationAirport = new Airport("Toronto", "YYZ");
   flight = new Flight(departureAirport, destinationAirport)
}

public void testStatus_inital() {
   // implicit setup
   // Exercise SUT & verify outcome
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}
public void testStatus_cancelled() {
   // implicit setup partially overriden:
   flight.cancel();
   // exercise SUT & verify outcome
   assertEquals(FlightState.CANCELLED, flight.state());
   // teardown:
   flight.destroy();
}
```

Note that the setup method is run once for each Test Method. If the fixture setup is fairly complex and involves accessing a database, this could result in *?Slow Tests?*.

Refactoring Notes

To convert a Testcase Class from a Standard Test Fixture to a *Shared Test Fixture*, you simply use a ?Convert Instance Variable to Static? refactoring to make the fixture outlast the creating test instance and use ?Lazy Initialization? to avoid recreating it for each Test Method.

To introduce a ?Test Setup Decorator?, just use the ?Move Method? refactoring to transfer the Shared Setup Method to it. You'll also need to create a suite method on some object that instantiates the ?Test Setup Decorator? as a wrapper for your Test Suite Object.

Solution Example

This example shows the fixture converted to a *Shared Test Fixture* and the Test Methods modified to use the Fixture Holding Class Variable to access the contents of the fixture.

```
static Airport departureAirport
static Airport destinationAirport
static Flight flight

public setup() {
   super.setup();
   if ( departureAirport == nil) {
      departureAirport = new Airport("Calgary", "YYC");
      destinationAirport = new Airport("Toronto", "YYZ");
      flight = new Flight(departureAirport, destinationAirport)
   }
}

public void testStatus_inital() {
   // implicit setup
   // Exercise SUT & verify outcome
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}

public void testStatus_cancelled() {
   // implicit setup partially overriden:
   flight.cancel();
   // exercise SUT & verify outcome
   assertEquals(FlightState.CANCELLED, flight.state());
   // teardown:
   flight.destroy();
}
```


Shared Fixture Construction

Techniques for constructing the Shared Test Fixture.

Prebuilt Fixture

The test fixture is created once and used by many tests possibly over many test runs.

Context

Your tests are taking too long to execute mostly due to the time it takes to set up a test fixture in each test. Even the time it takes to create a Shared Test Fixture each time you run the tests is still slowing down your tests too much for your liking.

Problem

When is the Test Fixture constructed?

Forces

- If other tests have access to the same test fixture, those tests may leave the fixture in an unexpected state which could affect the results of other tests using the same fixture.
- A fixture set up within the test is guaranteed to be there regardless of how the test is run.
- It takes additional design effort to design a specific test fixture for each test.
- A simple test fixture specific to a test is easier to understand than a more general fixture that satisfies the needs of many tests.
- It takes additional computational cycles to build a new test fixture for each test.
- It takes fewer computational cycles to build a simple test fixture for a single test than to build a large general-purpose test fixture.
- Setting up the fixture ahead of time may result in *Unrepeatable Tests* if the tests modify the fixture.

Therefore ...

If you cannot afford to build a new test fixture for each test run, pre-build the fixture and reuse the fixture in many test runs.

Implementation Notes

A common way to do *Prebuilt Fixture* is with a Data Loader or ?Database Population Script?. The latter is an example of a Back Door Fixture Setup wherein we bypass the SUT and interact with its database directly.

When using *Prebuilt Fixture*, the tests must be Self-Cleaning Tests otherwise they will become *Unrepeatable Tests* as their leftover detritus starts interfering with subsequent test runs

The fixture you create using *Prebuilt Fixture* is a Shared Test Fixture but you still need to decide how broadly you want to share it. You can keep it relatively private if you ensure that each developer has their own ?Private Database?. This gives the developer control over how many test runners can access it at one time.

You'll need to decide how often you want to refresh the Shared Test Fixture as it is bound to decay over time as various tests modify it and forget to put it back the way they found it. (You'll know when it does happen because you'll start to see tests failing when they should pass.)

Motivating Example

TBD

Find example

Refactoring Notes

The most obvious refactoring is from a Shared Test Fixture (shared between tests within a single test run) to one shared between multiple test runs.

Solution Example

There is no "typical" example of a *Prebuilt Fixture* because the reason for prebuilding the fixture often dictates how it must be prebuilt. Most often, it is not built in code (at least not the same kind of code our XUnit tests are written in!). It may be built manually using the user interfaces of other applications or by other applications based on input files or database contents. As far as we are concerned, the main thing is that it was there before we ran our first test run and it will still be there well after the test run is done.

A common way to prebuild the fixture is to use database tools to reload snapshots taken earlier. The databases may be restored using these snapshots on a regular schedule (say, every night or weekend) or on an "as needed" basis.

Because the fixture exists before the test executes, the test must find the relevant objects in the fixture. The most robust way is by using Finder Methods as shown here:

```
public void testPurchase_firstPurchase() {
   Customer buyer = findCreditworthyCustomer();
   ...
}
public void testPurchase_subsequentPurchase() {
   Customer buyer = findCreditworthyCustomer();
   ...
}
```

Related Patterns

The most obvious alternative is ?Just In Time Fixture Construction? (A.K.A. "Clean Slate Setup") of a Private Test Fixture.

Related Smells

While Shared Test Fixture opens the door to *Interacting Tests*, *Prebuilt Fixture* is particularly likely to result in *Unrepeatable Tests*.

Care has to be taken when coding the tests to ensure that you don't end up with a *?Fragile Fixture?* as a result of tests depending too heavily on assumptions about what exists in the fixture.

Lazy Fixture Initialization

Let the first test that needs the shared test fixture create it.

Context

You have chosen to use a Shared Test Fixture and want to be able to run each test independently. (That is, you want to avoid the *?Lonely Test?* problem.

Problem

How do you know it is time to build the shared test fixture?

Forces

- If several tests are sharing the same test fixture, they can't all build the fixture otherwise it won't be shared.
- The test fixture needs to be built before the first test executes.

- The test fixture should be cleaned up after the last test completes otherwise the test fixture will be left dangling.
- If the test fixture is set up outside the test, the test cannot be run unless the fixture setup code is also run. This introduces a dependency.

Therefore ...

Use ?Lazy Initialization? to construct the fixture in the first test that needs it. Let all the other tests discover that the fixture is already created and that they can reuse it thus avoiding the effort of constructing it.

Implementation Notes

Since the point of Shared Test Fixture is to save execution time and effort by having multiple tests use the same instance of the test fixture, we'll need to keep a reference to the fixture we create so that we can find the fixture if it already exists and inform other tests that it now exists once we have constructed it. If the test fixture is only being shared within a single Testcase Class, the simplest solution is to use a Fixture Holding Class Variable for each fixture object we need to hold a reference to. But if we want to share the test fixture between many Testcase Classes we'll need some other solution.

A common solution is to use a Test Fixture Singleton to access the fixture. In some cases, the singleton is the shared ?Test Database?. But for in-memory test fixtures we'll have to create a real ?Singleton? object to access the fixture.

Similarly, if the *Lazy Fixture Initialization* will be done from several different Testcase Classes, we would prefer not to duplicate the ?Lazy Initialization? logic in each class. The two obvious places to put the setup and teardown logic are a ?Test Helper? or within the Test Fixture Singleton. The latter eliminates the need for a ?Test Helper? class but may introduce some type dependencies if you want to use the same Test Fixture Singleton for different types of fixtures.

The biggest detracting factor for *Lazy Fixture Initialization* is that fact that while it is easy to discover that you are the first test and you need to construct the fixture, it is difficult to determine that you are the last test and that the fixture should be destroyed. Most members of the XUnit family of Test Automation Frameworks do not provide a way to determine this other than using a Fixture Setup Decorator for the entire test suite.

Motivating Example

Rework this example

In this example, we have been building a new fixture for each Testcase Object.

```
Airport departureAirport
Airport destinationAirport
Flight flight

public setup() {
   super.setup();
   departureAirport = new Airport("Calgary", "YYC");
   destinationAirport = new Airport("Toronto", "YYZ");
   flight = new Flight(departureAirport, destinationAirport)
}

public void testStatus_inital() {
   // implicit setup
   // Exercise SUT & verify outcome
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}
public void testStatus_cancelled() {
   // implicit setup partially overriden:
   flight.cancel();
```



```
      // exercise SUT & verify outcome
      assertEquals(FlightState.CANCELLED, flight.state());
      // teardown:
      flight.destroy();
   }
```

Note that we are using Implicit Fixture Setup (where the Test Automation Framework calls the setup and teardown methods for each Testcase Object.) This means we already have the fixture setup and teardown logic nicely packaged up in the setup method.

Refactoring Notes

To use *Lazy Fixture Initialization* at the Testcase Class level, we merely insert the ?Lazy Initialization? logic into the setup method so that only the first test will cause it to be run.

Don't forget to remove the teardown logic as it will render the ?Lazy Initialization? logic useless if it removes the fixture after each Test Method has run! Sorry, but there is nowhere that you can move this logic to so that it will be run after the *last* Test Method has completed.

If we want to do use *Lazy Fixture Initialization* at the Suite of Suites level, we will need to do a bit more work. First, we'll need a way to hold a reference to the test fixture independently of any one Testcase Class. And we'll need a place to put the fixture setup logic; let's assume we can use a Test Fixture Singleton for both roles. We'll use ?Rename Method? refactoring to rename the setup method to better reflect what it is doing (e.g. setupAirportsFixture.) Then we'll use ?Move Method? refactoring to move it to the newly created AirportFixture Test Fixture Singleton class. Now, we'll add a new setup to our Testcase Class that calls setupAirportsFixture on AirportFixture. We'll also do a ?Move Variable? refactoring to move any variables we were referencing from the setup logic to AirportFixture and make them public so we can still access them from the tests.

Solution Example

Here, we are creating the fixture using ?Lazy Initialization? by checking if our Fixture Holding Class Variable is empty before calling methods to setup the fixture. We then use the variables to access the objects in the fixture from our tests. Note that while there is a teardown method on AirportFixture, there is no way to know when to call it! That's the main consequence of using *Lazy Fixture Initialization*. And since the variables are static, they will not go out of scope so the fixture will not be garbage collected.

Some IDEs (such as Eclipse) and some Test Runners automatically reload your classes every time the test suite is run. This causes the original class variable to go out of scope and the fixture **will** be garbage collected before the new version of the class is run. In these cases there is no negative consequence of using *Lazy Fixture Initialization*.

```
static Airport departureAirport
static Airport destinationAirport
static Flight flight

public setup() {
   super.setup();
   if ( departureAirport == nil) {
      departureAirport = new Airport("Calgary", "YYC");
      destinationAirport = new Airport("Toronto", "YYZ");
      flight = new Flight(departureAirport, destinationAirport)
   }
}

public void testStatus_inital() {
   // implicit setup
   // Exercise SUT & verify outcome
   assertEquals(FlightState.PROPOSED, flight.state());
   // teardown:
   flight.destroy();
}
```



```
public void testStatus_cancelled() {
   // implicit setup partially overriden:
   flight.cancel();
   // exercise SUT & verify outcome
   assertEquals(FlightState.CANCELLED, flight.state());
   // teardown:
   flight.destroy();
}
   public teardown() {
   super.teardown();
   // Cannot do much else here because we don't know
   // whether or not the last test has run yet!
   }
```

Related Patterns

If you need to ensure that the test fixture has been cleaned up after the last test completes, you'll want to use a Fixture Setup Decorator instead of *Lazy Fixture Initialization*.

Related Smells

The main reason for using a Shared Test Fixture and hence *Lazy Fixture Initialization* is a *?Slow Test?* caused by too many test fixture objects being created each time every test is run.

Fixture Setup Decorator

Wrap the test suite with a Decorator that sets up the shared test fixture before running the tests and tears it down after all the tests are done.

Sketch goes here.

If you have chosen to use a Shared Test Fixture whether it is for reasons of convenience or necessity and you have chosen not to use a Prebuilt Fixture, you'll need to ensure that the fixture gets built before each Test Run. Lazy Fixture Initialization is one strategy you could employ to create the test fixture "just in time" for the first test. But if it is critical to teardown the fixture after the last test, how do you know that all the tests have been completed?

How It Works

Fixture Setup Decorator works by "bracketing" the execution of the entire test suite with a set of matching setup and teardown "bookends". The pattern ?Decorator? is just what we need to make this happen. We construct a *Fixture Setup Decorator* that holds a reference to the Test Suite Object we wish to "bracket" and pass it to the Test Runner. When it is time to run the test, the Test Runner calls the run method on our *Fixture Setup Decorator* rather than the one on the actual Test Suite Object. The *Fixture Setup Decorator* does the fixture setup before calling the run method and tears down the fixture after it is done. Because it is calling the run method on the suite, it knows exactly when the suite has finished (because the suite returns control to the caller).

The basic approach is to create some code that sets up the fixture, delegates test execution to the test suit to be run, and then executes the code to tear down the fixture. To better line up with the normal XUnit calling conventions, we typically put the code that constructs the test fixture into a method called setup and the code the tears down the fixture into teardown. Then our *Fixture Setup Decorator's* run logic consists of three lines of code:

```
void run() {
   setup();
   theSuite.run();
   teardown();
}
```

When To Use It

Use a *Fixture Setup Decorator* whenever it is critical that a Shared Test Fixture be set up before every Test Run and that it is torn down after the run is complete. It may be critical because tests are using Hard-coded Values that would cause the tests to fail if run again without cleaning up after each run (*Unrepeatable Tests*.) Or it may just be the issue of the database slowly filling up with data from repeated test runs. Another reason if the tests need to change some global parameter before running and want to change it back when they are done. Stubbing out the database to avoid *?Slow Tests?* is one common reason for this; setting global switches to a particular configuration is another.

Motivating Example

In this example, we have a set of tests that use Lazy Fixture Initialization to build the Shared Test Fixture and Finder Methods to find the objects in the fixture. We have discovered that the leftover fixture is causing problems so we want to clean up properly after the last test has finished running.

```
TBD
```

Since there is no easy way to do this with Lazy Fixture Initialization, we will have to change our fixture setup logic to use a *Fixture Setup Decorator* instead.

Refactoring Notes

When creating a *Fixture Setup Decorator*, we can reuse the exact same fixture setup logic; we just need to call it at a different time. So this refactoring consists mostly of moving the call to the fixture setup logic (ideally just a call to a Creation Method) from the Testcase Class(es) to the setup method of a *Fixture Setup Decorator* class. This class is usually a subclass of a generic *Fixture Setup Decorator* class that implements the setup/run/teardown sequence as a ?Template Method?. We just provide a concrete implementation of the setup and teardown methods.

If our instance of XUnit does not support *Fixture Setup Decorator* directly, we can create our own *Fixture Setup Decorator* super class by building a single-purpose *Fixture Setup Decorator* and then introducing a constructor parameter and instance variable to hold the test suite to be run. Finally, we do an ?Extract Superclass? refactoring to create our reusable super class.

Solution Example

In this example, we have moved all the setup logic to the setup method of a *Fixture Setup Decorator*. We have also written some fixture teardown logic.

```
class TypicalFixtureSetupDecorator extends FixtureSetup {
   void setup() {
      MyTestHelper.setupStandardFixture();
   }

   void setup() {
      MyTestHelper.teardownStandardFixture();
   }
}
```

Variations

Reusable Setup Decorator

The ?simplest thing that could possibly work? is to hard-coded the name of the decorated class in the suite method of the decorator. This causes the decorator to act as the Test Suite Factory for the decorated suite.

```
   static TestSuite suite() {
      return new SuiteDecorator(TestSuiteToBeDecorated.suite());
   }
```

This is certainly simple but we will need a different *Fixture Setup Decorator* for each class we want to decorate.

Reusable Setup Decorator

If we want to reuse the *Fixture Setup Decorator* for different test suites, we can parameterize the *Fixture Setup Decorator's* constructor method with the Test Suite Object to be run; this means that the setup and teardown logic can be coded within the *Fixture Setup Decorator* eliminating the need for a separate ?Test Helper Class? class just to reuse the setup logic across tests. To make it easy for the Test Runner to create our test suite, we'll also need to create a Test Suite Factory that calls the *Fixture Setup Decorator's* constructor with the Test Suite Object to be decorated.

Many members of the XUnit family of Test Automation Frameworks provide a reusable super class that implements *Fixture Setup Decorator*. All you have to do is subclass this class and implement the setup and teardown methods as you would in a normal Testcase Class. When instantiating your *Fixture Setup Decorator* class, pass the Test Suite Object you are decorating as the constructor argument.

```
static TestSuite suite(testSuite) {
   return new SuiteDecorator(testSuite);
}
```

Pushdown Decorator

One of the main drawbacks of using a *Fixture Setup Decorator* is that that tests cannot be run by themselves because they depend on the *Fixture Setup Decorator* to set up the fixture. One way we have circumvented this drawback is to provide a means to push the decorator down to the level of the individual tests rather than the whole test suite. This required a few modifications to the TestSuite class to allow the *Fixture Setup Decorator* to be passed down to where the individual TestCase Objects are constructed during the ?Automated Test Method Discovery? process. As each object is created from the Test Method it is first wrapped in the *Fixture Setup Decorator* before it is added to the ?TestSuite Object's? collection of tests.

Of course, this negates one of the main sources of a speed advantage of using a *Fixture Setup Decorator* by forcing a new test fixture to be built for each test. In our case, this wasn't a problem because we were using the decorator to replace the real database infrastructure with an in-memory database so we got our speed improvement that way. (See Memory Improves Speed.)

Acknowledgements

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

References

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

[McKinnon] Endo Testing (Mock Objects)

