JUnit A Cook's Tour

Note: this article is based on JUnit 3.8.x.

---

1. Introduction

In an earlier article (see Test Infected: Programmers Love Writing Tests, Java Report, July 1998, Volume 3, Number 7), we described how to use a simple framework to write repeatable tests. In this article, we will take a peek under the covers and show you how the framework itself is constructed.

We carefully studied the JUnit framework and reflected on how we constructed it. We found lessons at many different levels. In this article we will try communicate them all at once, a hopeless task, but at least we will do it in the context of showing you the design and construction of a piece of software with proven value.

We open with a discussion of the goals of the framework. The goals will reappear in many small details during the presentation of the framework itself. Following this, we present the design and implementation of the framework. The design will be described in terms of patterns (surprise, surprise), the implementation as a literate program. We conclude with a few choice thoughts about framework development.

2. Goals

What are the goals of JUnit?

First, we have to get back to the assumptions of development. If a program feature lacks an automated test, we assume it doesn't work. This seems much safer than the prevailing assumption, that if a developer assures us a program feature works, then it works now and forever.

From this perspective, developers aren't done when they write and debug the code, they must also write tests that demonstrate that the program works. However, everybody is too busy, they have too much to do, they don't have enough time, to screw around with testing. I have too much code to write already, how am I supposed write test code, too? Answer me that, Mr. Hard-case Project Manager.

So, the number one goal is to write a framework within which we have some glimmer of hope that developers will actually write tests. The framework has to use familiar tools, so there is little new to learn. It has to require no more work than absolutely necessary to write a new test. It has to eliminate duplicated effort.

If this was all tests had to do, you would be done just by writing expressions in a debugger. However, this isn't sufficient for testing. Telling me that your program works now doesn't help me, because it doesn't assure me that your program will work one minute from now after I integrate, and it doesn't assure me that your program will still work in five years, when you are long gone.

So, the second goal of testing is creating tests that retain their value over time. Someone other than the original author has to be able to execute the tests and interpret the results. It should be possible to combine tests from various authors and run them together without fear of interference.

Finally, it has to be possible to leverage existing tests to create new ones. Creating a setup or fixture is expensive and a framework has to enable reusing fixtures to run different tests. Oh, is that all?

3. The Design of JUnit

The design of JUnit will be presented in a style first used in (see "Patterns Generate Architectures", Kent Beck and Ralph Johnson, ECOOP 94). The idea is to explain the design of a system by starting with nothing and applying patterns, one after another, until you have the architecture of the system. We will present the architectural problem to be solved, summarize the pattern that solves it, and then show how the pattern was applied to JUnit.

3.1 Getting started- TestCase

First we have to make an object to represent our basic concept, the TestCase. Developers often have tests cases in mind, but they realize them in many different ways-

- print statements,
- debugger expressions,
- test scripts.

If we want to make manipulating tests easy, we have to make them objects. This takes a test that was only implicit in the developer's mind and makes it concrete, supporting our goal of creating tests that retain their value over time. At the same time, object developers are used to, well, developing with objects, so the decision to make tests into objects supports our goal of

making test writing more inviting (or at least less imposing).

The Command pattern (see Gamma, E., et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995) fits our needs quite nicely. Quoting from the intent, "Encapsulate a request as an object, thereby letting you… queue or log requests…" Command tells us to create an object for an operation and give it a method "execute". Here is the code for the class definition of TestCase:

```
public abstract class TestCase implements Test {
    …
}
```

Because we expect this class to be reused through inheritance, we declare it "public abstract". For now, ignore the fact that it implements the Test interface. For the purposes of our current design, you can think of TestCase as a lone class.

Every TestCase is created with a name, so if a test fails, you can identify which test failed.

```
public abstract class TestCase implements Test {
    private final String fName;

    public TestCase(String name) {
        fName= name;
    }

    public abstract void run();
        …
}
```

To illustrate the evolution of JUnit, we use diagrams that show snapshots of the architecture. The notation we use is simple. It annotates classes with shaded boxes containing the associated pattern. When the role of the class in the pattern is obvious then only the pattern name is shown. If the role isn't clear then the shaded box is augmented by the name of the participant this class corresponds to. This notation minimizes the clutter in diagrams and was first shown in (see Gamma, E., Applying Design Patterns in Java, in Java Gems, SIGS Reference Library, 1997) Figure 1 shows this notation applied to TestCase. Since we are dealing with a single class and there can be no ambiguities just the pattern name is shown.

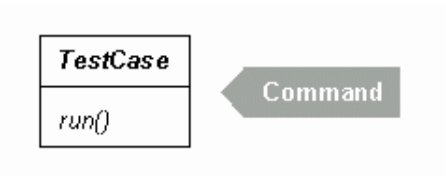

Figure 1 TestCase applies Command

3.2 Blanks to fill in- run()

The next problem to solve is giving the developer a convenient "place" to put their fixture code and their test code. The declaration of TestCase as abstract says that the developer is expected to reuse TestCase by subclassing. However, if all we could do was provide a superclass with one variable and no behavior, we wouldn't be doing much to satisfy our first goal, making tests easier to write.

Fortunately, there is a common structure to all tests- they set up a test fixture, run some code against the fixture, check some results, and then clean up the fixture. This means that each test will run with a fresh fixture and the results of one test can't influence the result of another. This supports the goal of maximizing the value of the tests.

Template Method addresses our problem quite nicely. Quoting from the intent, "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." This is exactly right. We want the developer to be able to separately consider how to write the fixture (set up and tear down) code and how to write the testing code. The execution of this sequence, however, will remain the same for all tests, no matter how the fixture code is written or how the testing code is written.

Here is the template method:

```
public void run() {
    setUp();
    runTest();
    tearDown();
}
```

The default implementations of these methods do nothing:

```
protected void runTest() {
}
```

```
protected void setUp() {
}

protected void tearDown() {
}
```

Since setUp and tearDown are intended to be overridden but will be called by the framework we declare them as protected. The second snapshot of our tour is depicted in Figure 2.

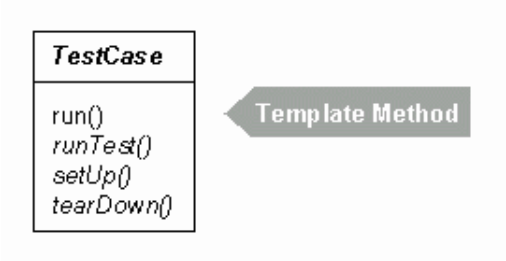

Figure 2 TestCase.run() applies Template Method

3.3 Reporting results- TestResult

If a TestCase runs in a forest, does anyone care about the result? Sure- you run tests to make sure they run. After the test has run, you want a record, a summary of what did and didn't work.

If tests had equal chances of succeeding or failing, or if we only ever ran one test, we could just set a flag in the TestCase object and go look at the flag when the test completed. However, tests are (intended to be) highly asymmetric- they usually work. Therefore, we only want to record the failures and a highly condensed summary of the successes.

The Smalltalk Best Practice Patterns (see Beck, K. Smalltalk Best Practice Patterns, Prentice Hall, 1996) has a pattern that is applicable. It is called *Collecting Parameter*. It suggests that when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you. We create a new object, TestResult, to collect the results of running tests.

```
public class TestResult extends Object {
   protected int fRunTests;

   public TestResult() {
      fRunTests= 0;
   }
}
```

This simple version of TestResult only counts the number of tests run. To use it, we have to add a parameter to the TestCase.run() method and notify the TestResult that the test is running:

```
public void run(TestResult result) {
   result.startTest(this);
   setUp();
   runTest();
   tearDown();
}
```

And the TestResult has to keep track of the number of tests run:

```
public synchronized void startTest(Test test) {
   fRunTests++;
}
```

We declare the TestResult method startTest as synchronized so that a single TestResult can collect the results safely when the tests are run in different threads. Finally, we want to retain the simple external interface of TestCase, so we create a no-parameter version of run() that creates its own TestResult:

```
public TestResult run() {
   TestResult result= createResult();
   run(result);
   return result;
}

protected TestResult createResult() {
```

```
    return new TestResult();
}
```

Figure 3 shows our next design snapshot.

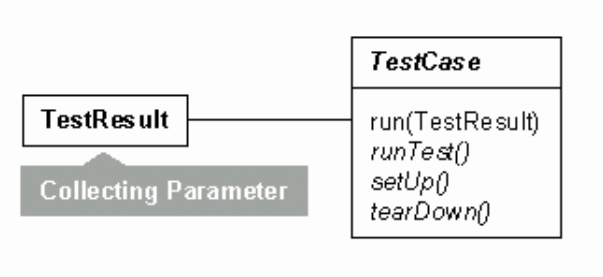

Figure 3: TestResult applies Collecting Parameter

If tests always ran correctly, then we wouldn't have to write them. Tests are interesting when they fail, especially if we didn't expect them to fail. What's more, tests can fail in ways that we expect, for example by computing an incorrect result, or they can fail in more spectacular ways, for example by writing outside the bounds of an array. No matter how the test fails we want to execute the following tests.

JUnit distinguishes between *failures* and *errors*. The possibility of a failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Failures are signaled with an AssertionFailedError error. To distinguish an unanticipated error from a failure, failures are caught in an extra catch clause (1). The second clause (2) catches all other exceptions and ensures that our test run continues..

```
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
        runTest();
    }
    catch (AssertionFailedError e) { //1
        result.addFailure(this, e);
    }

    catch (Throwable e) { // 2
        result.addError(this, e);
    }
    finally {
        tearDown();
    }
}
```

An AssertionFailedError is triggered by the assert methods provided by TestCase. JUnit provides a set of assert methods for different purposes. Here is the simplest one:

```
protected void assertTrue(boolean condition) {
    if (!condition)
        throw new AssertionFailedError();
}
```

The AssertionFailedError is not meant to be caught by the client (a testing method inside a TestCase) but inside the Template Method TestCase.run(). We therefore derive AssertionFailedError from Error.

```
public class AssertionFailedError extends Error {
    public AssertionFailedError () {}
}
```

The methods to collect the errors in TestResult are shown below:

```
public synchronized void addError(Test test, Throwable t) {
    fErrors.addElement(new TestFailure(test, t));
}

public synchronized void addFailure(Test test, Throwable t) {
    fFailures.addElement(new TestFailure(test, t));
}
```

TestFailure is a little framework internal helper class to bind together the failed test and the signaled exception for later reporting.

```
public class TestFailure extends Object {
    protected Test fFailedTest;
    protected Throwable fThrownException;
}
```

The canonical form of collecting parameter requires us to pass the collecting parameter to each method. If we followed this advice, each of the testing methods would require a parameter for the TestResult. This results in a "pollution" of these method signatures. As a benevolent side effect of using exceptions to signal failures we can avoid this signature pollution. A test case method, or a helper method called from it, can throw an exception without having to know about the TestResult. As a refresher here is a sample test method from our MoneyTest suite. It illustrates how a testing method doesn't have to know anything about a TestResult:

```
public void testMoneyEquals() {
    assertTrue(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assertTrue(!f12CHF.equals(f14CHF));
}
```

JUnit comes with different implementations of TestResult. The default implementation counts the number of failures and errors and collects the results. TextTestResult collects the results and presents them in a textual form. Finally, UITestResult is used by the graphical version of the JUnit Test Runner to update the graphical test status.

TestResult is an extension point of the framework. Clients can define their own custom TestResult classes, for example, an HTMLTestResult reports the results as an HTML document.

3.4 No stupid subclasses - TestCase again

We have applied Command to represent a test. Command relies on a single method like execute() (called run() in TestCase) to invoke it. This simple interface allows us to invoke different implementations of a command through the same interface.

We need an interface to generically run our tests. However, all test cases are implemented as different methods in the same class. This avoids the unnecessary proliferation of classes. A given test case class may implement many different methods, each defining a single test case. Each test case has a descriptive name like testMoneyEquals or testMoneyAdd. The test cases don't conform to a simple command interface. Different instances of the same Command class need to be invoked with different methods. Therefore our next problem is make all the test cases look the same from the point of view of the invoker of the test.

Reviewing the problems addressed by available design patterns, the Adapter pattern springs to mind. Adapter has the following intent "Convert the interface of a class into another interface clients expect". This sounds like a good match. Adapter tells us different ways to do this. One of them is a class adapter, which uses subclassing to adapt the interface. For example, to adapt testMoneyEquals to runTest we implement a subclass of MoneyTest and override runTest to invoke testMoneyEquals.

```
public class TestMoneyEquals extends MoneyTest {
    public TestMoneyEquals() { super("testMoneyEquals"); }
    protected void runTest () { testMoneyEquals(); }
}
```

The use of subclassing requires us to implement a subclass for each test case. This puts an additional burden on the tester. This is against the JUnit goal that the framework should make it as simple as possible to add a test case. In addition, creating a subclass for each testing method results in class bloat. Many classes with only a single method are not worth their costs and it will be difficult to come up with meaningful names.

Java provides anonymous inner classes which provide an interesting Java-specific solution to the class naming problem. With anonymous inner classes we can create an Adapter without having to invent a class name:

```
TestCase test= new MoneyTest("testMoneyEquals ") {
    protected void runTest() { testMoneyEquals(); }
};
```

This is much more convenient than full subclassing. It preserves compile-time type checking at the cost of some burden on the developer. Smalltalk Best Practice Patterns describes another solution for the problem of different instances behaving differently under the common heading of *pluggable behavior*. The idea is to use a single class which can be parameterized to perform different logic without requiring subclassing.

The simplest form of pluggable behavior is the *Pluggable Selector*. Pluggable Selector stores a Smalltalk method selector in an instance variable. This idea is not limited to Smalltalk. It is also applicable to Java. In Java there is no notion of a method selector. However, the Java reflection API allows us to invoke a method from a string representing the method's name. We can use this feature to implement a pluggable selector in Java. As an aside, we usually don't use reflection in ordinary application code. In our case we are dealing with an infrastructure framework and it is therefore OK to wear the reflection hat.

JUnit offers the client the choice of using pluggable selector or implementing an anonymous adapter class as shown above. To do so, we provide the pluggable selector as the default implementation of the runTest method. In this case the name of the test case has to correspond to the name of a test method. We use reflection to invoke the method as shown below. First we look up the Method object. Once we have the method object we can invoke it and pass its arguments. Since our test methods take no arguments we can pass an empty argument array:

```
protected void runTest() throws Throwable {
    Method runMethod= null;
    try {
        runMethod= getClass().getMethod(fName, new Class[0]);
    } catch (NoSuchMethodException e) {
        assertTrue("Method \""+fName+"\" not found", false);
    }
    try {
        runMethod.invoke(this, new Class[0]);
    }
    // catch InvocationTargetException and IllegalAccessException
}
```

The JDK 1.1 reflection API only allows us to find public methods. For this reason you have to declare the test methods as public, otherwise you will get a NoSuchMethodException.

Here is the next design snapshot, with Adapter and Pluggable Selector added.

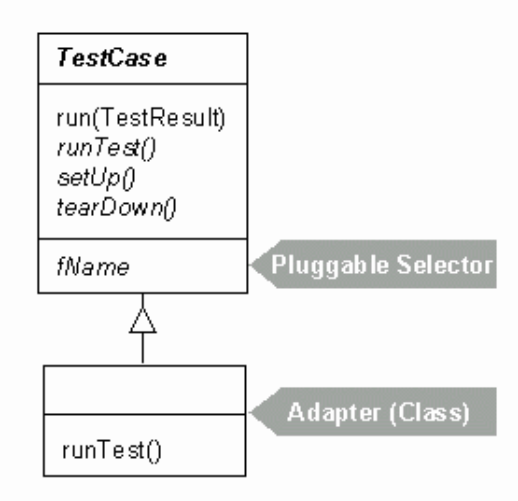

Figure 4: TestCase applies either Adapter with an anonymous inner class or Pluggable Selector

3.5 Don't care about one or many - TestSuite

To get confidence in the state of a system we need to run many tests. Up to this point JUnit can run a single test case and report the result in a TestResult. Our next challenge is to extend it so that it can run many different tests. This problem can be solved easily when the invoker of the tests doesn't have to care about whether it runs one or many test cases. A popular pattern to pull out in such a situation is Composite. To quote its intent "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." The point about part-whole hierarchies is of interest here. We want to support suites of suites of suites of tests.

Composite introduces the following participants:

- Component: declares the interface we want to use to interact with our tests.
- Composite: implements this interface and maintains a collection of tests.
- Leaf: represents a test case in a composition that conforms to the Component interface.

The pattern tells us to introduce an abstract class which defines the common interface for single and composite objects. The primary purpose of the class is to define an interface. When applying Composite in Java we prefer to define an interface and not an abstract class. Using an interface avoids committing JUnit to a specific base class for tests. All that is required is that the tests conform to this interface. We therefore tweak the pattern description and introduce a Test interface:

```
public interface Test {
    public abstract void run(TestResult result);
}
```

TestCase corresponds to a Leaf in Composite and implements this interface as we have seen above.

Next, we introduce the Composite participant. We name the class TestSuite. A TestSuite keeps its child tests in a Vector:

```
public class TestSuite implements Test {
    private Vector fTests= new Vector();
}
```

The run() method delegates to its children:

```
public void run(TestResult result) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ) {
        Test test= (Test)e.nextElement();
        test.run(result);
    }
}
```

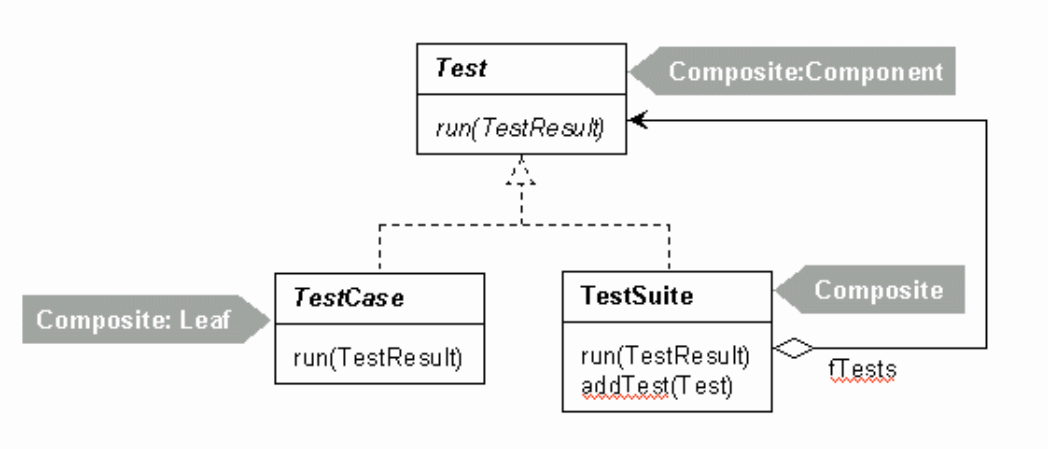

Figure 5: TestSuite applies Composite

Finally, clients have to be able to add tests to a suite, they can do so with the method addTest:

```
public void addTest(Test test) {
    fTests.addElement(test);
}
```

Notice how all of the above code only depends on the Test interface. Since both TestCase and TestSuite conform to the Test interface we can recursively compose suites of test suites. All developers can create their own TestSuites. We can run them all by creating a TestSuite composed of those suites.

Here is an example of creating a TestSuite:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
}
```

This works fine, but it requires us to add all the tests to a suite manually. Early adopters of JUnit told us this was stupid. Whenever you write a new test case you have to remember to add it to a static suite() method, otherwise it will not be run. We added a convenience constructor to TestSuite which takes the test case class as an argument. Its purpose is to extract the test methods and create a suite containing them. The test methods must follow the simple convention that they start with the prefix "test" and take no arguments. The convenience constructor uses this convention, constructing the test objects by using reflection to find the testing methods. Using this constructor the above code is simplified to:

```
public static Test suite() {
    return new TestSuite(MoneyTest.class);
}
```

The original way is still useful when you want to run a subset of the test cases only.

3.6 Summary

We are at the end of our cook's tour through JUnit. The following figure shows the design of JUnit at a glance explained with patterns.

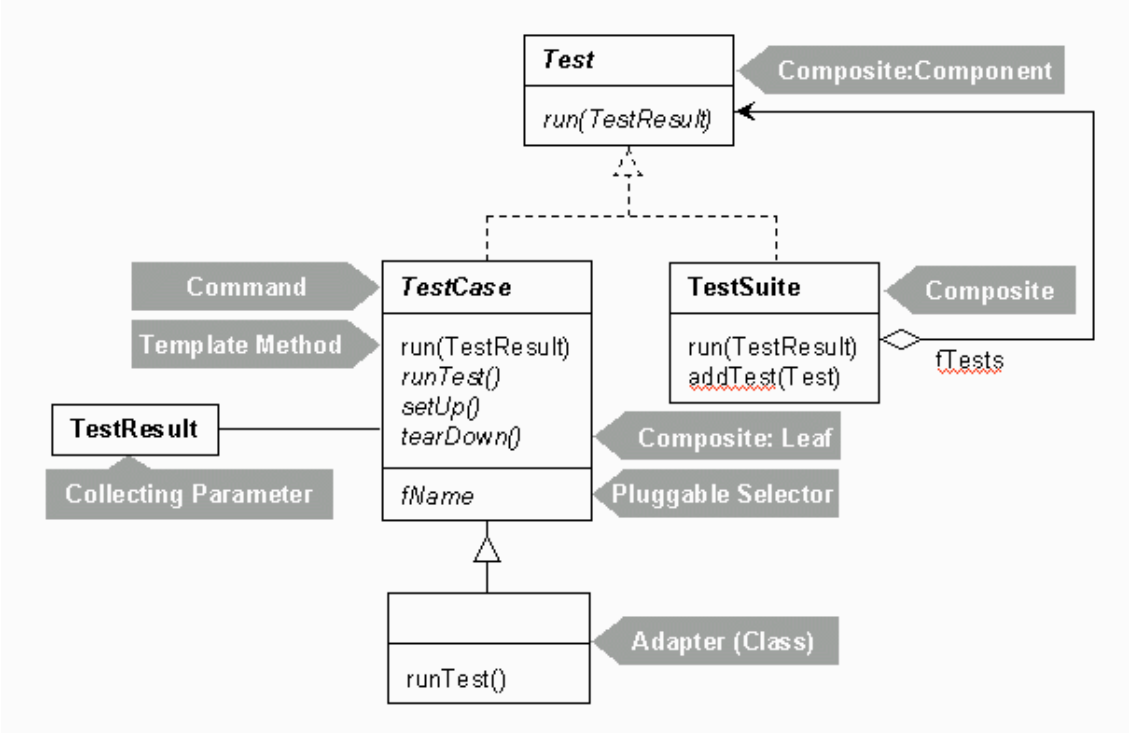

Figure 6: JUnit Patterns Summary

Notice how TestCase, the central abstraction in the framework, is involved in four patterns. Pictures of mature object designs show this same "pattern density". The star of the design has a rich set of relationships with the supporting players.

Here is another way of looking at all of the patterns in JUnit. In this storyboard you see an abstract representation of the effect of each of the patterns in turn. So, the Command pattern creates the TestCase class, the Template Method pattern creates the run method, and so on. (The notation of the storyboard is the notation of figure 6 with all the text deleted).

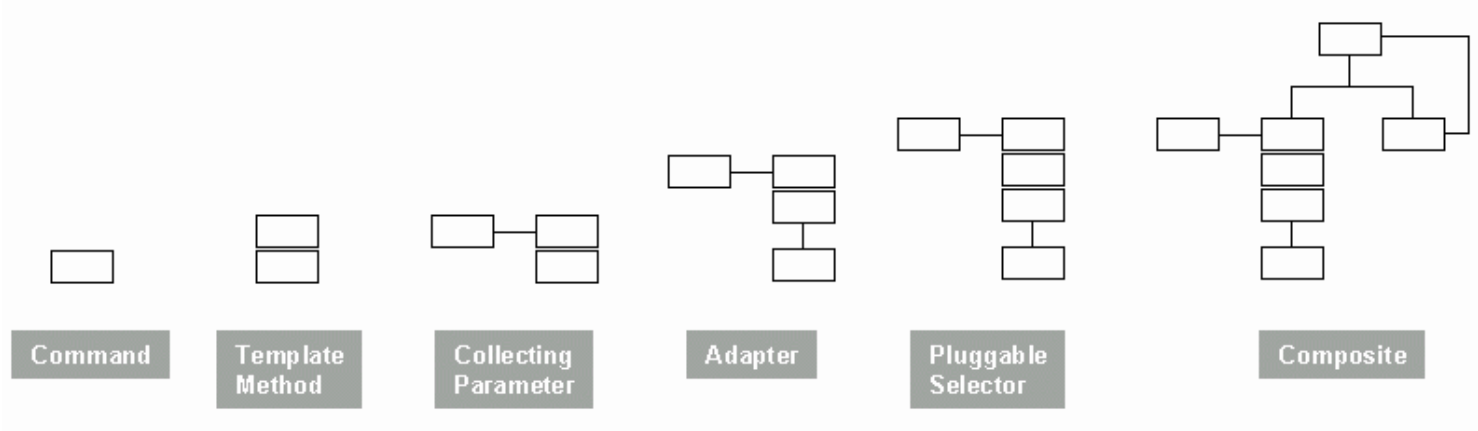

Figure 7: JUnit Pattern Storyboard

One point to notice about the storyboard is how the complexity of the picture jumps when we apply Composite. This is pictorial corroboration for our intuition that Composite is a powerful pattern, but that it "complicates the picture." It should therefore be used with caution.

4. Conclusion

To conclude, let's make some general observations:

- *Patterns*

  We found discussing the design in terms of patterns to be invaluable, both as we were developing the framework and as we try to explain it to others. You are now in a perfect position to judge whether describing a framework with patterns is effective. If you liked the discussion above, try the same style of presentation for your own system.
- *Pattern density*

  There is a high pattern "density" around TestCase, which is the key abstraction of JUnit. Designs with high pattern density are easier to use but harder to change. We have found that such a high pattern density around key abstractions is common for mature frameworks. The opposite should be true of immature frameworks - they should have low pattern density.

Once you discover what problem you are really solving, then you can begin to "compress" the solution, leading to a denser and denser field of patterns where they provide leverage.

- *Eat your own dog food*

  As soon as we had the base unit testing functionality implemented, we applied it ourselves. A TestTest verifies that the framework reports the correct results for errors, successes, and failures. We found this invaluable as we continued to evolve the design of the framework. We found that the most challenging application of JUnit was testing its own behavior.
- *Intersection, not union*

  There is a temptation in framework development to include every feature you can. After all, you want to make the framework as valuable as possible. However, there is a counteracting force- developers have to decide to use your framework. The fewer features the framework has, the easier it is to learn, the more likely a developer will use it. JUnit is written in this style. It implements only those features absolutely essential to running tests- running suites of tests, isolating the execution of tests from each other, and running tests automatically. Sure, we couldn't resist adding some features but we were careful to put them into their own extensions package (test.extensions). A notable member of this package is a TestDecorator allowing execution of additional code before and after a test.
- *Framework writers read their code*

  We spent far more time reading the JUnit code than we spent writing it, and nearly as much time removing duplicate functionality as we spent adding new functionality. We experimented aggressively with the design, adding new classes and moving responsibility around in as many different ways as we could imagine. We were rewarded (and are still being rewarded) for our monomania by a continuous flow of insights into JUnit, testing, object design, framework development, and opportunities for further articles.

The latest version of JUnit can be downloaded from http://www.junit.org.

5. Acknowledgements

Thanks to John Vlissides, Ralph Johnson, and Nick Edgar for careful reading and gentle correction.