Lecture 17: Case Study: JUnit

The JUnit testing framework which you've been using to test your own code in 6.170 is worth studying in its own right. It was developed by Kent Beck and Erich Gamma. Beck is an exponent of patterns and Extreme Programming (XP); Gamma is one of the authors of the celebrated design patterns book. JUnit is open source, so you can study the source code yourself. There's also a nice explanatory article in the JUnit distribution, entitled 'A Cook's Tour', which explains the design of JUnit in terms of design patterns, and from which much of the material in this lecture is drawn.

JUnit has been a great success. Martin Fowler, an insightful and pragmatic proponent of patterns and XP (and also author of a wonderful book on object models called *Analysis Patterns*), says about JUnit:

> *Never in the field of software development was so much owed by so many to so few lines of code.*

JUnit's ease of use is no doubt in large part responsible for its popularity. You might think that. since it doesn't do very much – it just runs a bunch of tests and reports their results – JUnit should be very simple. In fact, the code is rather complicated. The main reason for this is that it has been designed as a framework, to be extended in many unanticipated ways, and so it's full of rather complex patterns and indirections designed to allow an implementer to override some parts of the framework while preserving other parts.

Another complicating influence is a desire to make tests easy to write. There's a clever hack involving reflection that turns methods of a class into individual instances of the type *Test*. Here's another example of a hack that seems unconscionable at first. The abstract class *TestCase* inherits from the class *Assert*, which contains a bunch of static assertion methods, simply to allow a call to the static *assert* method to be written as just *assert (...)*, rather than *Assert.assert (...)*. In no sense is *TestCase* a subtype of *Assert*, of course, so this really makes no sense. But it does allow code within *TestCase* to be written more succinctly. And since all the test cases the user writes are methods of the *TestCase* class, this is actually pretty significant.

The use of patterns is skillful and well motivated. The key patterns we'll look at are: *Template Method*, the key pattern of framework programming; *Command*, *Composite*, and *Observer*. All these patterns are explained at length in Gamma et al, and, with the

exception of *Command*, have been covered already in this course.

My personal opinion is that JUnit, the jewel in the crown of XP, itself belies the fundamental message of the movement – that code alone is enough. It's a perfect example of a program that is almost incomprehensible without some abstract, global representations of the design explaining how the parts fit together. It doesn't help that the code is pretty lean on comments – and where are there comments they tend to dwell on which Swiss mountain the developer was sitting on when the code was written. Perhaps high altitude and thin air explains the coding style.The 'Cook's Tour' is essential; without it, it would take hours to grasp the subtleties of what's going on. And it would be helpful to have even more design representations. The 'Cook's Tour' presents a simplified view, and I had to construct for myself an object model explaining, for example, how the listeners work.

If you're one of those students who's skeptical about design representations, and who still thinks that code is all that matters, you should stop reading here, and curl up in a chair to spend an evening with JUnit's source code. Who knows, it may change your mind…

You can download the source code and documentation for JUnit from

http://www.junit.org/.

There's an open source repository at

http://sourceforge.net/projects/junit/

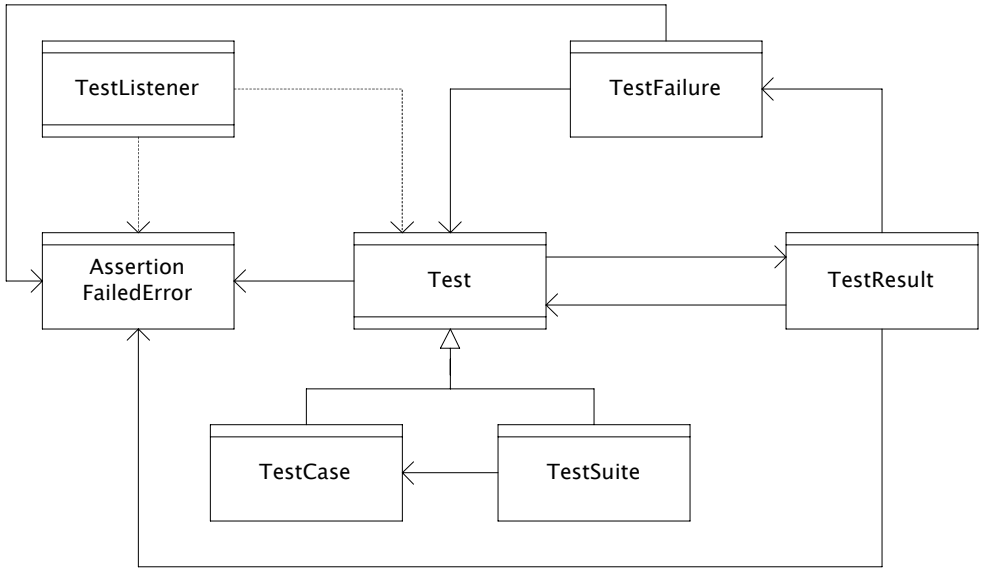
where can view (and contribute) bug reports.

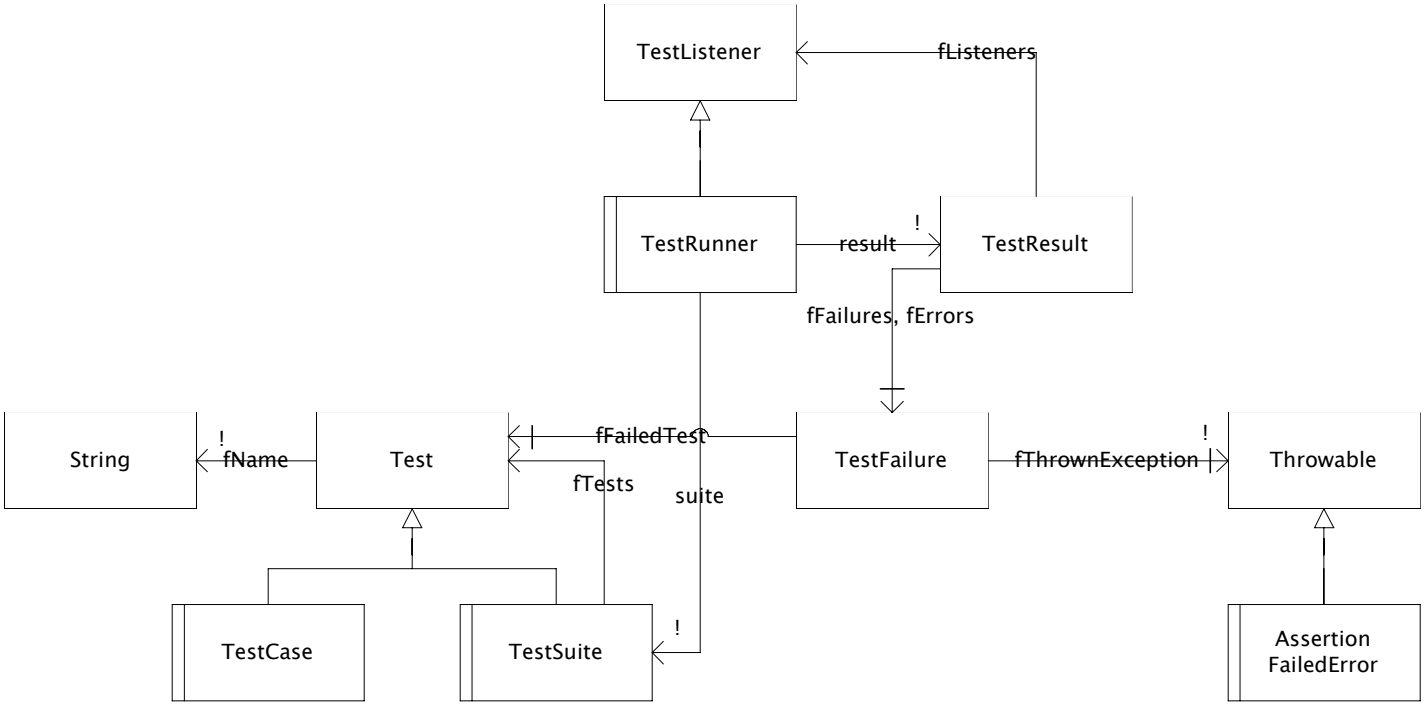
17.1 Overview

JUnit has several packages: *framework* for the basic framework, *runner* for some abstract classes for running tests, *textui* and *swingui* for user interfaces, and *extensions* for some useful additions to the framework. We'll focus on the *framework* package.

The diagrams show the object model and module dependences. You may want to follow along with these diagrams as you read our discussion. Both of these include only the framework modules, although I've included *TestRunner* in the object model to show how the listeners are connected; it's relations, *suite* and *result* are local variables of its *doRun* method.

Note that the module dependency diagram is almost fully connected. This is not surprising for a framework; the modules are not intended to be used independently.


TestListener
TestFailure
Assertion
FailedError
Test
TestResult
TestCase
TestSuite


TestListener
fListeners
TestRunner
result
!
TestResult
fFailures, fErrors
String
!
fName
Test
fFailedTest
fTests
suite
TestFailure
fThrownException
!
Throwable
TestCase
TestSuite
!
Assertion
FailedError

17.2 Command

The command pattern encapsulates a function as an object. It's how you implement a closure – remember that from 6.001? – in an object-oriented language. The command class typically has a single method with a name like *do*, *run* or *perform*. An instance of a subclass is created that overrides this method, and usually also encapsulates some state (in 6.001 lingo, the environment of the closure). The command can then be passed around as an object, and 'executed' by calling the method.

In JUnit, test cases are represented as command objects that implement the interface *Test*:

```
public interface Test {
    public void run();
    }
```

Actual test cases are instances of a subclass of a concrete class *TestCase*:

```
public abstract class TestCase implements Test {
    private String fName;
    public TestCase(String name) {
        fName= name;
    }

    public void run() {
        ...
        }
}
```

In fact, the actual code isn't quite like this, but starting from this simplified version will allow us to explain the basic patterns more easily. Note that the constructor associates a name with the test case, which will be useful when reporting results. In fact, all the classes that implement *Test* have this property, so it might have been good to add a method

```
public String getName ()
```

to the *Test* interface. Note also that the authors of JUnit use the convention that identifiers that begin with a lowercase *f* are fields of a class (that is, instance variables).

We'll see a more elaborate example of the command pattern when we study the *Tagger* program next week.

17.3 Template Method

One might make *run* an abstract method, thus requiring all subclasses to override it. But most test cases have three phases: setting up the context, performing the test, then tearing down the context. We can factor out this common structure by making *run* a *template method*:

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

The default implementations of the hook methods do nothing:

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

They are declared as protected so that they are accessible from subclasses (and can thus be overridden) but not accessible from outside the package. It would be nice to be able to prevent access except from subclasses, but Java doesn't offer such a mode. A subclass can selectively override these methods; if it overrides only *runTest*, for example, there will be no special *setUp* or *tearDown* behaviour.

We saw this same pattern in the last lecture in the skeletal implementations of the Java collections API. It is sometimes referred to in a rather corny way as the *Hollywood Principle*. A traditional API provides methods that get called by the client; a framework, in contrast, makes calls to the methods of its client: 'don't call us, we'll call you'. Pervasive use of templates is the essence of framework programming. It's very powerful, but also easy to write programs that are completely incomprehensible, since method implementations make calls at multiple levels in the inheritance hierarchy.

It can be difficult to know what's expected of a subclass in a framework. An analog of pre- and post-conditions hasn't been developed, and the state of the art is rather crude. You usually have to read the source code of the framework to use it effectively. The Java collections API does better than most frameworks, by including in the specifications of template methods some careful descriptions of how they are implemented. This is of course anathema to the idea of abstract specification, but it's unavoidable in the context of a framework.

17.4 Composite

As we discussed in Lecture 11, test cases are grouped into test suites. But what you do with a test suite is essentially the same as what you do with a test: you run it, and you report the result. This suggests using the *Composite* pattern, in which a composite object shares an interface with its elementary components.

Here, the interface is *Test*, the composite is *TestSuite*, and the elementary components are members of *TestCase*. *TestSuite* is a concrete class that implements *Test*, but whose *run* method, unlike the *run* method of *TestCase*, calls the *run* method of each test case that the suite contains. Instances of *TestCase* are added to a *TestSuite* instance with the method *addTest*; there's also a constructor that creates a *TestSuite* with a whole bunch of test cases, as we'll see later.

The example of *Composite* in the Gamma book has the interface include all the operations of the composite. Following this approach, *Test* should include methods like *addTest*, which apply only to *TestSuite* objects. The implementation section of the pattern description explains that there is a tradeoff between transparency – making the composite and leaf objects look the same – and safety – preventing inappropriate operations from being called. In terms of our discussion in the subtyping lecture, the question is whether the interface should be a true supertype. In my opinion it should be, since the benefits of safety outweigh those of transparency, and, moreover, the inclusion of composite operations in the interface is confusing. JUnit follows this approach, and does not include *addTest* in the interface *Test*.

17.5 Collecting Parameter

The *run* method of *Test* actually has this signature:

public void run(TestResult result);

It takes a single argument that is mutated to record the result of running the test. Beck calls this a 'collecting parameter' and views it as a design pattern in its own right.

There are two ways in which a test can fail. Either it produces the wrong result (which may include not throwing an expected exception), or it throws an unexpected exception (such as *IndexOutOfBoundsException*). JUnit calls the former 'failures' and the latter 'errors'. An instance of *TestResult* contains a sequence of failures and a sequence of errors, each failure or error being represented as an instance of the class *TestFailure*, which contains a reference to a *Test* and a reference to the exception object generated by the failure or error. (Failures always produce exceptions, since even when an unexpected result is produced without an exception, the *assert* method used in the test con-

verts the mismatch into an exception).

The *run* method in *TestSuite* is essentially unchanged; it just passes the *TestResult* when invoking the *run* method of each of its tets. The *run* method in *TestCase* looks something like this:

```
public void run (TestResult result) {
    setUp ();
    try {
        runTest ();
        }
    catch
        (AssertionFailedError e) {
            result.addFailure (test, e);
            }
        (Throwable e) {
            result.addError (test, e);
            }
    tearDown ();
    }
```

In fact, the control flow of the template method *run* is more complicated than we have suggested. Here are some pseudocode fragments showing what happens. It ignores the *setUp* and *tearDown* activities, and considers a use of *TestSuite* within a textual user interface:

```
junit.textui.TestRunner.doRun (TestSuite suite) {
    result = new TestResult ();
    result.addListener (this);
    suite.run (result);
    print (result);
    }

junit.framework.TestSuite.run (TestResult result) {
    forall test: suite.tests
        test.run (result);
    }

junit.framework.TestCase.run (TestResult result) {
    result.run (this);
    }
```

```
junit.framework.TestResult.run (Test test) {
    try {
        test.runBare ();
        }
    catch (AssertionFailedError e) {
        addFailure (test, e);
        }
    catch (Throwable e) {
        addError (test, e);
        }
    }

junit.framework.TestCase.runBare (TestResult result) {
    setUp();
    try {
        runTest();
        }
    finally {
        tearDown();
        }
    }
```

TestRunner is a user interface class that calls the framework and displays the results. There's a GUI version *junit.swingui* and a simple console version *junit.textui*, which we've shown an excerpt from here. We'll come to the listener later; ignore it for now.

Here's how it works. The *TestRunner* object creates a new *TestResult* to hold the results of the test; it runs the suite, and prints the results. The *run* method of *TestSuite* calls the *run* method of each of its constituent tests; these may themselves be *TestSuite* objects, so the method may be called recursively. This is a nice illustration of the simplicity that *Composite* brings. Eventually, since there is an invariant that a *TestSuite* cannot contain itself – not actually specified, and not enforced by the code of *TestSuite* either – the method will bottom out by calling the *run* methods of objects of type *TestCase*.

The *run* method of *TestCase* now has the receiver *TestCase* object swap places with the *TestResult* object, and calls the *run* method of *TestResult* with the *TestCase* as an argument. (Why?). The *run* method of *TestResult* then calls the *runBare* method of *TestCase*, which is the actual template method that executes the test. If the test fails, it

throws an exception, which is caught by the *run* method in *TestResult*, which then packages the test and exception as a failure or error of the *TestResult*.

17.6 Observer

For an interactive user interface, we'd like to show the results of the test as it happens incrementally. To achieve this, JUnit uses the *Observer* pattern.

The *TestRunner* class implements an interface *TestListener* which has methods *addFailure* and *addError* of its own. It plays the role of *Observer*. The class *TestResult* plays the role of *Subject*; it provides a method

```
public void addListener(TestListener listener)
```

which adds an observer. When the *addFailure* method of *TestResult* is called, in addition to updating its list of failures, it calls the *addFailure* method on each of its observers:

```
public synchronized void addFailure(Test test, AssertionFailedError e) {
    fFailures.addElement(new TestFailure(test, e));
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
            ((TestListener)e.nextElement()).addFailure(test, e);
            }
}
```

In the textual user interface, the *addFailure* method of *TestRunner* simply prints a character *F* to the screen. In the graphical user interface, it adds the failure to a list display and changes the colour of the progress bar to red.

17.7 Reflection Hacks

Recall that a test case is an instance of the class *TestCase*. To create a test suite in plain old Java, a user would have to create a fresh subclass of *TestCase* for each test case, and instantiate it. An elegant way to do this is to use anonymous inner classes, creating the test case as an instance of a subclass that has no name. But it's still tedious, so JUnit provides a clever hack.

The user provides a class for each test suite – called *MySuite* say – that is a subclass of *TestCase*, and which contains many test methods, each having a name beginning with the string 'test'. These are taken to be individual test cases.

```
public class MySuite extends TestCase {
```

```
void testFoo () {
   int x = MyClass.add (1, 2);
   assertEquals (x, 3);
   }
void testBar () {
   ...
   }
}
```

The class object *MySuite* itself is passed to the *TestSuite* constructor. Using reflection, the code in *TestSuite* instantiates *MySuite* for each of its methods beginning with 'test', passing the name of the method as an argument to the constructor. As a result, for each test method, a fresh *TestCase* object is created, with its name bound to the name of the test method. The *runTest* method of *TestCase* calls, again using reflection, the method whose name matches the name of the *TestCase* object itself, roughly like this:

```
void runTest () {
   Method m = getMethod (fName);
   m.invoke ();
   }
```

This scheme is obscure, and dangerous, and not the kind of thing you should emulate in your code. Here it's justifiable, because it's limited to a small part of the JUnit code, and it brings a huge advantage to the user of JUnit.

17.8 Questions for Self-Study

These questions arose when I constructed the object model for JUnit. They don't all have clear answers.

- Why are listeners attached to *TestResult*? Isn't *TestResult* already a kind of listener itself?
- Can a *TestSuite* contain no tests? Can it contain itself?
- Are *Test* names unique?
- Does the *fFailedTest* field of *TestFailure* always point to a *TestCase*?