From.

Eclipse 3.0 Kick Start

by Carlos Valcarcel

SAMS

High-Grade Testing Using JUnit

IN THIS CHAPTER

"Test, code, refactor, design. Repeat."

—Anonymous

Overview

Test-driven development. Testing frameworks. JUnit, Cactus, StrutsTestCase, HttpUnit, HtmlUnit, JWebUnit, SwingUnit. These days there is almost as much to be learned about testing APIs as the libraries with which you are writing your code. JUnit, the most popular of the current testing frameworks, was created by Erich Gamma and Kent Beck. You may remember Erich Gamma as one of the authors of the groundbreaking book *Design Patterns*. Kent Beck, in addition to various contributions in software analysis and design, is responsible in part for a small earthquake called *Extreme Programming* (sometimes called *Agile Programming* so as not to upset the faint of heart).

Documentation, source code, and binaries for JUnit can be found at `http://www.junit.org`, which will point you to SourceForge (`http://sourceforge.net/projects/junit/`) where the latest version can be found. However, you are one of the select few who have downloaded a tool that already has the latest version of the most popular testing framework on the face of the planet. The real issue is why you should use it.

Eclipse makes the creation and use of tests almost trivial (almost because you still have to decide what to test and how to test it). Having JUnit built in is not unique to Eclipse as a Java IDE, but as an additional piece to an already feature-rich IDE, it makes the decision to go with Eclipse all the more easy.

I am not going to spend time trying to convince you that writing tests is good for you, your programs, and your paycheck. What I will say is that without tests to prove your code works, you don't know for sure that it does work. Do tests prove that your code works in every instance? Of course not. Your users will be certain to remind you of that. What tests will do for you is guarantee that the situations you've planned for work, the situations that don't work fail gracefully, and bugs that have been squashed stay that way. Let's take a look at how you can do all this using Eclipse.

The JUnit Framework

The JUnit framework is made of a number of classes that take care of the nitty-gritty details of running your tests, as well as running a GUI to visually display which of your tests have passed and which have failed. From your perspective, you need to learn one class: `TestCase`. Once you become comfortable with `TestCase` then you may, on occasion, use `TestSuite`.

The process of using JUnit can be summed up as follows:

1. Write a test class that subclasses `TestCase`. It should not be able to compile because you have not yet written the class to be tested. Your first test has just failed.
2. Write the class that needs to be tested. Your first test now passes.
3. Add to the test class a test of functionality you have not yet written in the other class. As your second test, this also fails.
4. Write the smallest amount of functionality in the other class. Your second test now passes.
5. Test another bit of nonexistent functionality. This test fails.
6. Add the missing functionality. The test now passes.

By now you should see a pattern emerging. Write the test, which fails, and then write the functionality for which the test is checking, which causes the test to pass. You start with the simplest functionality and work your way up. As the functionality gets more complex, you know that prior functionality continues to work. If prior functionality stops working, you know it right away and you can fix it before the complexity grows out of hand.

Writing JUnit tests in Eclipse is only marginally different. In order to take advantage of the JUnit Wizard, the class to be tested must exist. In Eclipse's case, the previous list would look like this:

1. Create an empty class to be tested.
2. Create a subclass of `TestCase` to test the other class.
3. Write a test of simple functionality of the other class. The test fails.
4. Write the simplest piece of functionality for the other class. The test should now pass.

5. Write another test and watch it fail.
6. Add the functionality that will cause the test to pass.

Perform steps 5 and 6 until it is time to go home, until you hit a milestone date, or until you have to deliver your code. If you used use cases to drive your schedule and test-driven development to prove that the use cases pass the good scenarios and know how to behave in the bad scenarios, you win.

TestCase

When you write a test case, you subclass `junit.framework.TestCase`. In the same way that a servlet subclasses `HttpServlet` so that the servlet engine knows how to manage its life cycle, you are responsible for subclassing `TestCase` so the JUnit framework can walk your test object through its life cycle. The TestCase API includes assert methods that you will use to confirm the validity of results returned by the object under testing.

Let's discuss what happens when a test case runs within Eclipse. The JUnit framework starts up, loads the test class selected on the workbench (your test class does not need `main()`), creates as many copies of your test case as there are methods that start with the word `test` (for example, `testFindCustomer()`), and begins to run the test objects one at a time. Before the test method is run, an initialization method called `setUp()` is called to give you the opportunity to create whatever objects you need to make your test work. When your test method completes, either with a success or failure, a cleanup method called `tearDown()` is called so your code can safely dispose of used resources.

The Granularity of Tests

A question that always comes up in discussions about tests and testing frameworks is, how many tests (meaning assertions) should you put in a test method? The quick answer would be to put one assertion per test method because it gives JUnit the opportunity to run all your tests rather than just the ones that passed prior to the one that failed. However, the number of tests can get rather high, in which case you would have an immense number of test methods, which would be hard to maintain (the second argument against tests). The longer answer is, group associated tests together as long as they make sense. Start with one test method per method being tested. As you find that the methods are getting too long, break them up into functional test areas. In other words, use refactoring as a way of controlling the inevitable onslaught of success.

TestSuite

A suite of tests can be run by creating a subclass of `junit.framework.TestSuite`. Within this class, you would list the various subclasses of `TestCase` that you would like JUnit to run by overriding `suite()`. When you create a test suite using Eclipse, the code generator uses code markers so that it can regenerate the suite as often as you like. After you complete the example, you will get a chance to create, and re-create, a test suite.

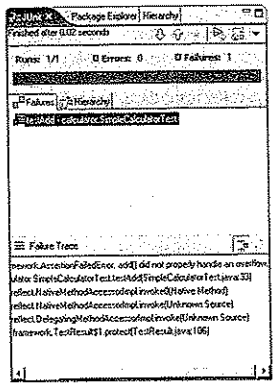

FIGURE 6.1 JUnit's TestRunner displaying the red bar, signaling it is not time to go home yet.

TestRunner: The JUnit GUI

So how does JUnit run within Eclipse? Do you simply select a test and run it like a Java application? When JUnit, or one of its variants, runs within Eclipse, it opens TestRunner, the GUI used by JUnit to display the results of completed tests (see Figure 6.1). The most visible part of TestRunner is the colored bar that runs below the title bar. You will learn to question a red bar when you expected a test to pass and a green bar when you expected a test to fail. Below the colored bar are three status values: how many tests of the available number of tests have run, how many errors (non-JUnit exceptions) occurred, and how many failures (assertion failures) occurred. Below these counters are two tabs: The Failures tab lists the methods that failed, and the Hierarchy tab lists all the methods with either a red X (error), a grey X (failure), or a green check (success).

Below the method list is the output of the currently selected method. If the method succeeds, there should be no output. If the result bar is red, expect to find output.

The title bar of TestRunner has five convenience buttons:

- Next or Previous Failed Test—Either of these buttons will take you directly to the method that encountered a failure or error.
- Stop JUnit Test Run—In case the test class has encountered an infinite loop or has simply decided to hang, you can kill JUnit from here.
- Rerun Last Test—The same as pressing Ctrl+F11, except it only runs the last JUnit test, not the last class that was run.
- Scroll Lock—Used to keep the selected test and its output in sync.

Creating a Test Case

In order for you to get a good feel for how to use JUnit, let's create a class that simply prints out a greeting. The greeting is changeable, as is the name of the person being greeted.

Create a Class

Start Eclipse and create a Java project called Greeter. Due to the way the JUnit Wizard works, you need to create the class to be tested first. Create a new class called `Greeter` in package `example` (you can create the package at the same time you are creating the class). The class should be quite empty after the code generator is done creating it.

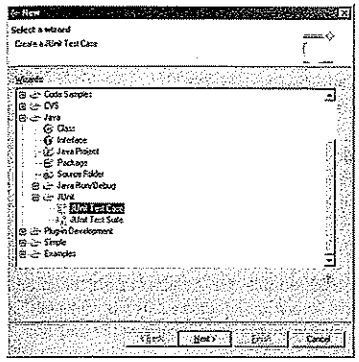

FIGURE 6.2 The New dialog with the JUnit test case selected.

Create a Test Case

Let's look at creating a JUnit test. With the `Greeter` class selected in the Package Explorer, press Ctrl+N (or you can right-click the `Greeter` class and select New, Other). When the New dialog appears, select Java, JUnit, JUnit Test Case (see Figure 6.2). Click Next. Because this is the first time you are creating a JUnit test, the JUnit Wizard will ask if you would like it to add the JUnit JAR file to your class path. Click Yes to go to the JUnit Test Case page.

If you selected the `Greeter` class before you pressed Ctrl+N, the JUnit Test Case page should be almost completely filled in. If the page does not appear to be populated, click Cancel and start again.

This page lists all the information needed to generate the class:

- The name of the source folder to which the class should be written.
- The package into which the test class will go. You could put the test class into a different package, but conventionally the test classes live in the same package as the class they are testing. This gives the test class access to package/protected methods that may need to be tested.
- The name of the class to be tested.
- The name of the test class. The wizard uses the convention of appending the word *Test* to the end of any test classes. Other IDEs use the word *Test* as a prefix for the JUnit classes they generate. If you come up with another convention, just be consistent.
- The super class of the test class. This will almost always be `junit.framework.TestCase`, unless you come up with another class that you would prefer to extend. Be aware, though: if the class does not ultimately extend `TestCase`, it will not work within the JUnit framework.

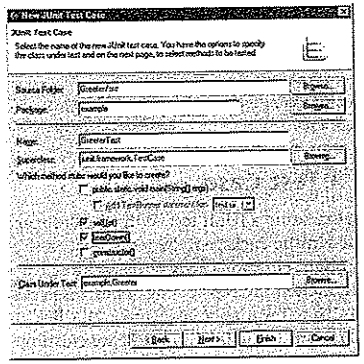

FIGURE 6.3 The JUnit Test Case page with the two main life cycle methods selected.

The only missing information needed is which of the lifecycle methods you want the class to include. Check `setUp()` and `tearDown()` and then click Finish (see Figure 6.3).

When the `GreeterTest` class opens in the Java editor, go to `setUp()` and remove the call to `super.setUp()`. In its place create a `Greeter` object and store it in an instance field called `_greeter`:

```
protected void setUp() throws Exception {
  _greeter = new Greeter();
}
```

A light bulb and red X appear in the left margin on the line where `_greeter` is declared. Click once on the light bulb and, when the content assist window opens, double-click Create Field _greeter. Save the file once the instance field is created.

The code in `setUp()` affords you the opportunity to single-source the creation of objects used through the test class. Because all your tests will need a `Greeter` object, `setUp()` is an ideal place to create it.

Next, empty out the `tearDown()` method and set `_greeter` to null:

```
protected void tearDown() throws Exception {
  _greeter = null;
}
```

Now, no matter what happens, you will always have a good object to work with before the test starts, and the object will always be released to garbage collection when the test completes. This may not seem like a big deal for a single test, but you can safely assume that you will have dozens, hundreds, maybe even thousands of tests running as your code development progresses.

One of the things we want the `Greeter` class to do is to return a greeting for display. A string returned as a result for display should not be null. (Yes, I know there are plenty of reasons why you might want a null returned from a method, but not in this case.) Our first test is now defined: a call to the method that returns the greeting should not return null. Above `setUp()`, define the method `testGreeting()` as follows:

```
public void testGreeting() {
}
```


Within `testGreeting()`, make a call to the `Greeter` method `getGreeting()`:

```
public void testGreeting() {
    String actual = _greeter.getGreeting();
}
```

Yes, it is true that there is no method in the `Greeter` class called `getGreeting()`. Remember step 3 of the development of JUnit tests? Your first test just failed. In order to make it succeed, you need to add the method to the `Greeter` class. Eclipse makes this a trivial task. Just single-click the light bulb and double-click the suggested fix in the content assist window:

```
Create method 'getGreeting()' in Greeter.java
```

The `Greeter` class comes forward, showing you the new method that has just been added. Save the `Greeter.java`.

Return to the GreeterTest editor window and save the file. The light bulb is gone, and our test-by-implication test is now successful. Because your first real test is to make sure you don't get a null from the call to `getGreeting()`, you need to take the result of the call and compare it against an expected result. By extending `TestCase`, you have an extensive selection of methods to check the result of a call and either do nothing if the result was as expected or complain if the result was invalid.

`TestCase` extends the `Assert` class, which has the following assertion methods:

- `assertEquals()`
- `assertTrue()`
- `assertFalse()`
- `assertNotNull()`
- `assertNull()`
- `assertNotSame()`
- `assertSame()`
- `fail()`

All the `assertXXX()` methods will throw an `AssertionFailedError` if the value they are passed is false. The `fail()` method throws an `AssertionFailedError` as soon as it is called. Each one gives you the choice of using its default error message or one you supply. For the test in `testGreeting()`, you could make a call to `assertEquals()` and compare the result to `null`, but the `assertNotNull()` method will work much better:

```
public void testGreeting() {
    String actual = _greeter.getGreeting();
    assertNotNull("getGreeting() returned null.", actual);
}
```

Running the Test

Save your files and make sure that GreeterTest is the active editor. From the main menu, select Run, Run As, JUnit Test. You already know that the `getGreeting()` method is going to return a null, so running TestRunner should give you a red bar (see Figure 6.4). If a green bar appears, make sure you are calling the proper assert method and that you are passing in the result of the call to `getGreeting()`.

Let's fix `getGreeting()` so it does not return a null:

```
public String getGreeting() {
    return "";
}
```

Simply returning an empty string should cause the test to be successful, and pressing Ctrl+F11, which runs the last thing you executed, causes TestRunner to return a green bar. Success!

SHOP TALK

White Box Testing Versus Black Box Testing

A *black box test* is a test run on code without you knowing its internal makeup. A *white box test* is a test where you know exactly what is going on in the code. JUnit tests are usually white box tests, but you could also write black box tests against third-party vendor libraries.

When I first started writing tests, I found it a little disconcerting to know that I was writing tests to prove that code would behave in the fashion I had written it. Checking whether a method returns (or doesn't return) an expected result seemed trivial, especially when I knew what the code was doing. Why should I check, for example, that a method does not return a null when I know it will never return a null? There was no code anywhere (in the code I had written) that could possibly return a null.

I found out the reason as soon as I used someone else's code in my algorithm.

If the set of acceptable values is known, a test needs to be written to check that the result does not fall outside of this set or, if the value does fall outside the acceptable set, that the bad value is returned under known conditions.

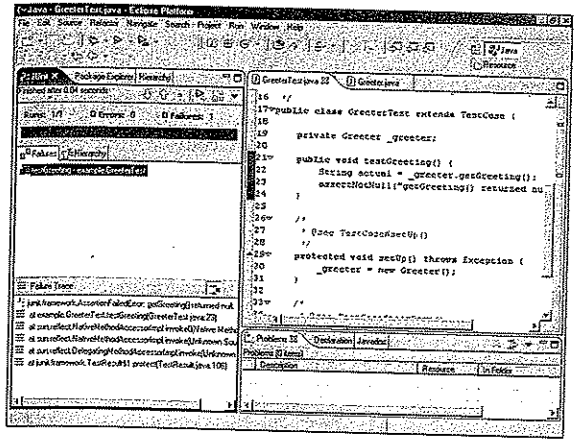

FIGURE 6.4 The TestRunner GUI with the expected red bar and the message passed in as the first argument to `assertNotNull()`.

If this were a chapter on writing tests in an incremental fashion, here are the next tests you would try to write:

1. Test for a default greeting.
2. Test the ability to change the current greeting with a null (throw an exception), a blank string (throw an exception), and a nonblank string.
3. Test whether the greeting can be reset to the default.
4. Test for a default greeting that can be personalized.
5. Test the ability to change the personalizable greeting with a null (throw an exception), a blank string (throw an exception), and a nonblank string.
6. Test whether the personalizable greeting can be reset to the default.

The number of things that can be implemented in the `Greeter` class is not insignificant, but you can check them reliably by adding slightly more complex tests in each iteration. But I digress.

When you discover a test that fails and the code has reached a complexity level where simply eye-balling it does not suggest where the problem may be, you need to debug the code through your test. Debugging a JUnit test is the same as debugging any other Java code: set a breakpoint in the test code or in the class being tested and from the main menu select Run, Debug As, JUnit Test. The Debug perspective opens and displays TestRunner at the bottom of the screen after the first full run, with the debug views and editor above it (see Figure 6.5).

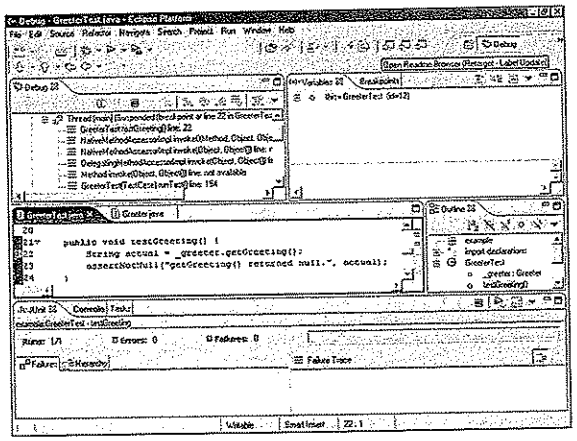

FIGURE 6.5 The Debug perspective with the JUnit TestRunner displaying the methods to be run.

Everything discussed so far would give you the impression that you can only run tests at the object level. In fact, you can select one method for execution to the exclusion of all the other test methods. Select the method to be executed from the Package Explorer or the Outline view, or double-click the method name in the Java editor. From the main menu, select Run, Run As, JUnit Test. Only the selected method will be run within JUnit. Unfortunately, you can only select one method at a time; this is not an arbitrary selection.

Creating and Running a Test Suite

As mentioned before, a JUnit test suite contains one or more tests that should be run as a unit. `TestSuite` is a composite object that contains `Test` objects (including other test suites) that are prepared to run all or one of their tests. When the JUnit framework starts, it uses reflection to make a call to the `suite()` method of the object created from the incoming class type. Because JUnit uses reflection, the only thing the class has to define to work within the framework is `suite()`. The incoming class does not have to be a type of `Test`. The `suite()` method must return an object of type `Test` so that the framework can begin running tests. The `suite()` method returns a `Test` object that contains whatever test objects you decide. Internally, JUnit performs the same operation, only it creates enough objects of your test type to run each individual method.

To make this explanation clearer, let's look at a class that defines `suite()` and returns a `TestSuite` object composed of a combination of single-method tests and multiple-method tests:

```
package example;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("Test for example");

        // Only run the named methods.
        suite.addTest(new TimeSeriesServiceTest("testGetTimeSeries"));
        suite.addTest(new OTCQuoteServiceTest("testGetOTCQuote"));
        suite.addTest(new QuoteServiceTest("testGetQuote"));

        // Run all of the tests contained within each class.
        //$JUnit-BEGIN$
        suite.addTestSuite(OTCQuoteServiceTest.class);
        suite.addTestSuite(QuoteServiceTest.class);
        suite.addTestSuite(TimeSeriesServiceTest.class);
        //$JUnit-END$
        return suite;
    }
}
```

The `AllTests` class has the following features:

- It does not extend any JUnit class.
- It declares a `suite()` method that will return an object of type `Test`.
- Within `suite()`, a `TestSuite` object is created.
- The `TestSuite` object has three method-specific tests added to it through calls to `addTest()`. Because method names are being passed into the test class constructors, only those tests will be run.
- The `TestSuite` object has three test classes added to its internal list through calls to `addTestSuite()`, and all the tests within each test class will be executed.

If the three test classes mentioned in the `AllTests` class have two test methods each, the call sequence might look something like this:

```
In testGetTimeSeries().
In testGetOTCQuote().
In testGetQuote().
In testGetOTCQuote().
In testGetOTCQuoteName().
In testGetQuote().
In testGetQuoteSymbol().
In testGetTimeSeries().
In testGetTimeSeriesFloat().
```

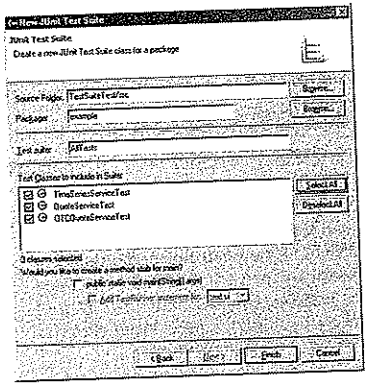

FIGURE 6.6 The JUnit Test Suite page with all the available tests checked.

The three single tests were run first. Next the test classes were called, in turn, and all their tests were run before moving on to the next test object. All these methods have a `System.out.println()` line, but normally the tests are silent if they succeed.

To generate a class that creates a `TestSuite` object, you open the New dialog and select Java, JUnit, TestSuite and then click Next. The JUnit Test Suite page will display the source folder in which the code will be generated, the package to which the class will belong, and the name of the class, which defaults to `AllTests` (see Figure 6.6). In the list below the test suite name are the various JUnit tests the builder recognizes. You can select zero or more JUnit tests for inclusion in `TestSuite`.

Also for the purposes of the example, I have deleted the comments from the code that was generated by the JUnit Wizard. Let's look at what was created:

```
package example;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("Test for example");
        //$JUnit-BEGIN$
```



```
        suite.addTestSuite(OTCQuoteServiceTest.class);
        suite.addTestSuite(QuoteServiceTest.class);
        suite.addTestSuite(TimeSeriesServiceTest.class);
        //$JUnit-END$
        return suite;
    }
}
```

The first line in `AllTests.suite()` is the instantiation of a `TestSuite` object. `TestSuite`, as a composite object, is the container of the various tests you want to run. The next line is a code marker used by the JUnit builder in case you decide to regenerate the suite. Anything outside of the code markers will be saved, whereas anything within the markers will disappear when you regenerate the code. The next line of code adds a class definition to the `TestSuite` object using `addTestSuite()`. This has the effect of creating a new `TestSuite` object and adding all the methods that start with *test* to the new `TestSuite` object, which is then added to your topmost suite. At the end of all this, `suite()` returns the `TestSuite` object.

What happens as you add and remove individual tests in the course of development? Regenerate the test suite class. You can do this in one of two ways:

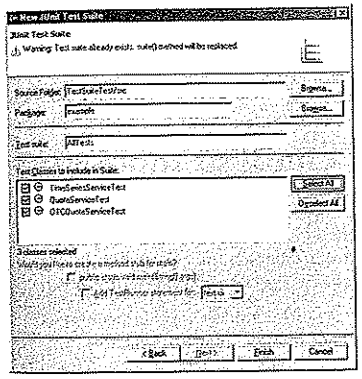

FIGURE 6.7 The JUnit Test Suite page displaying the warning about `suite()` being replaced.

- Press Ctrl+N (which opens the New dialog), select Java, JUnit, Test Suite, and then click Next. The JUnit Test Suite page will display a warning that `suite()` already exists and that it will be replaced unless you give the test suite class a new name (see Figure 6.7).
- Right-click the test suite class in the Package Explorer and select Recreate Test Suite from the pop-up menu. The Recreate Test Suite dialog will list the available JUnit tests for you to choose from. Select the tests you want to have appear in the code and click OK (see Figure 6.8).

Running the test suite is no different from running a regular JUnit test (you select Run, Run As, JUnit Test). If you try to run the test suite as a regular Java class, it will not work (unless you add `main()` and a call to TestRunner).

I have been very careful not to say that the JUnit Wizard creates a `TestSuite` class. The wizard does not. The wizard generates a class that contains the `suite()` method, which will instantiate a `TestSuite` object and return it to any callers.