6

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.

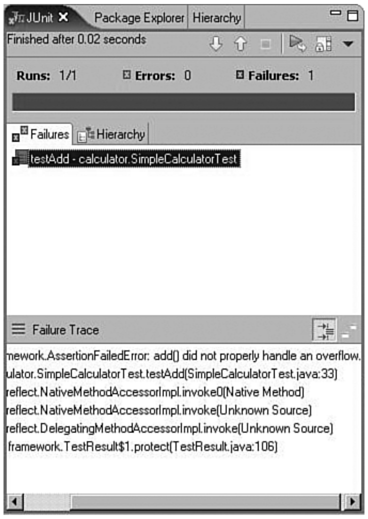
Let's discuss what happens when a test case runs within Eclipse. The JUnit framework starts up, loads the test class selected on the workbench (your test class does not need `main()`), creates as many copies of your test case as there are methods that start with the word `test` (for example, `testFindCustomer()`), 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.



```
Finished after 0.02 seconds
Runs: 1/1    Errors: 0    Failures: 1
testAdd - calculator.SimpleCalculatorTest
mework.AssertionFailedError: add() did not properly handle an overflow.
ulator.SimpleCalculatorTest.testAdd(SimpleCalculatorTest.java:33)
reflect.NativeMethodAccessorImpl.invoke0(Native Method)
reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
framework.TestResult$1.protect(TestResult.java:106)
```

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.

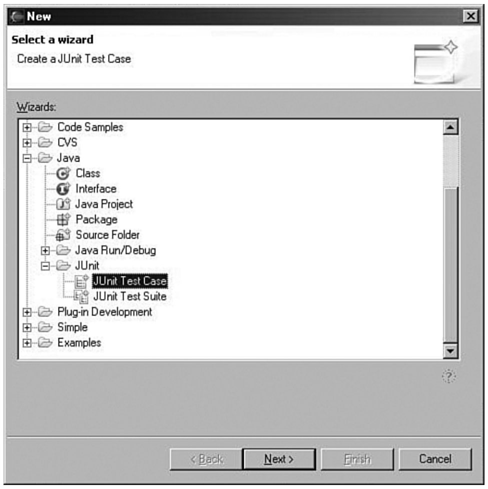


FIGURE 6.2 The New dialog with the JUnit test case selected.

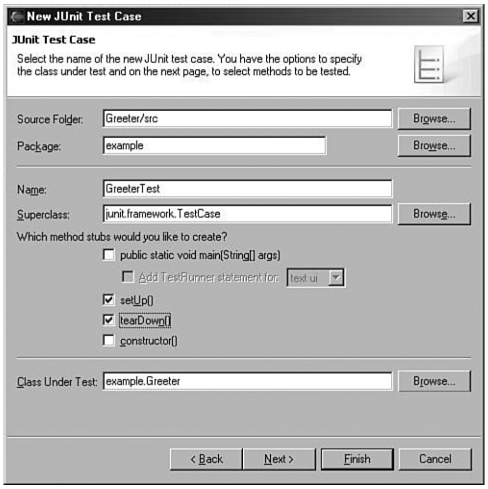
Create a Test Case

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.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

Source Folder: Greeter/src
Package: example
Name: GreeterTest
Superclass: junit.framework.TestCase
Which method stubs would you like to create?
- [ ] public static void main(String[] args)
  - [ ] Add TestRunner statement for: text ui
- [x] setUp()
- [x] tearDown()
- [ ] constructor()

Class Under Test: example.Greeter

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

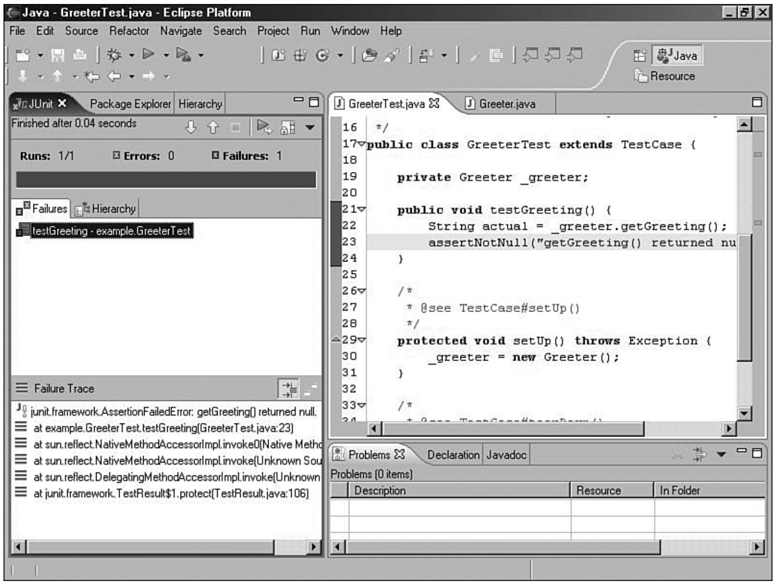
White Box Testing Versus Black Box Testing

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.



Finished after 0.04 seconds

Runs: 1/1 Errors: 0 Failures: 1

testGreeting - example.GreeterTest

```
junit.framework.AssertionFailedError: getGreeting() returned null.
  at example.GreeterTest.testGreeting(GreeterTest.java:23)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Metho
  at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Sou
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown
  at junit.framework.TestResult$1.protect(TestResult.java:106)
```

```
 */
public class GreeterTest extends TestCase {

    private Greeter _greeter;

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

    /*
     * @see TestCase#setUp()
     */
    protected void setUp() throws Exception {
        _greeter = new Greeter();
    }

    /*
```

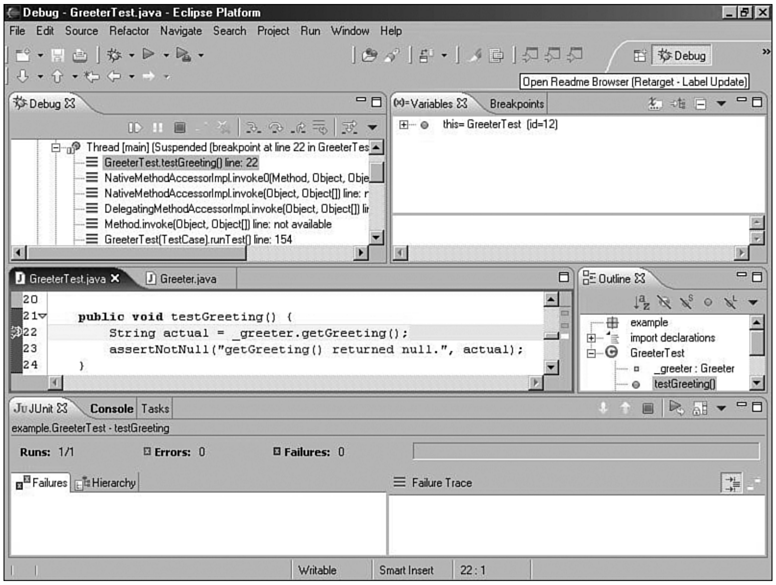
FIGURE 6.4 The TestRunner GUI with the expected red bar and the message passed in as the first argument to `assertNotNull()`.

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).



```
Thread [main] (Suspended (breakpoint at line 22 in GreeterTes
    GreeterTest.testGreeting() line: 22
    NativeMethodAccessorImpl.invoke0(Method, Object, Obje
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: r
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) li
    Method.invoke(Object, Object[]) line: not available
    GreeterTest(TestCase).runTest() line: 154
```

```
this= GreeterTest (id=12)
```

```

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

```
example.GreeterTest - testGreeting
Runs: 1/1        Errors: 0        Failures: 0
```

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:

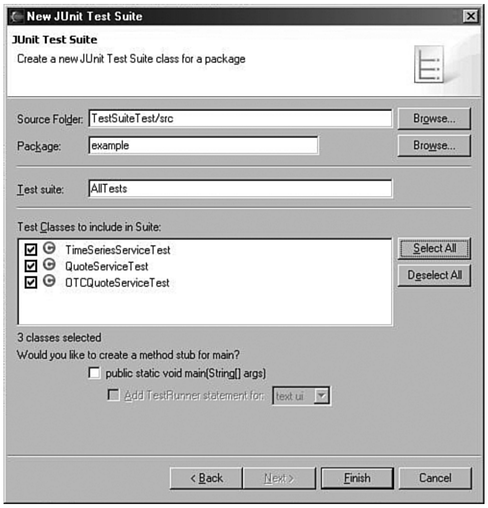
```
package example;

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().
```


New JUnit Test Suite

JUnit Test Suite
Create a new JUnit Test Suite class for a package

Source Folder: TestSuiteTest/src
Package: example
Test suite: AllTests
Test Classes to include in Suite:
[✓] TimeSeriesServiceTest
[✓] QuoteServiceTest
[✓] OTCQuoteServiceTest
3 classes selected
Would you like to create a method stub for main?
[ ] public static void main(String[] args)
[ ] Add TestRunner statement for: text ui

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:

- 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).

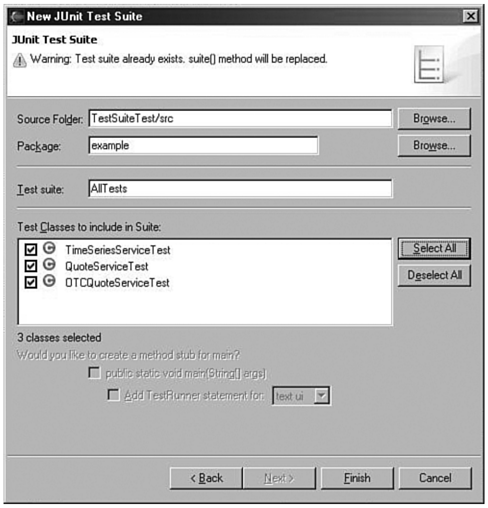
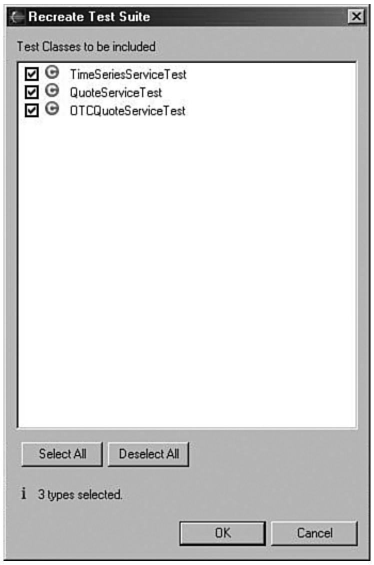

New JUnit Test Suite
JUnit Test Suite
Warning: Test suite already exists. suite() method will be replaced.
Source Folder: TestSuiteTest/src
Package: example
Test suite: AllTests
Test Classes to include in Suite:
☑ TimeSeriesServiceTest
☑ QuoteServiceTest
☑ OTCQuoteServiceTest
3 classes selected
Would you like to create a method stub for main?
☐ public static void main(String[] args)
☐ Add TestRunner statement for: text ui

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

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.


Recreate Test Suite

Test Classes to be included

- [✓] TimeSeriesServiceTest
- [✓] QuoteServiceTest
- [✓] OTCQuoteServiceTest

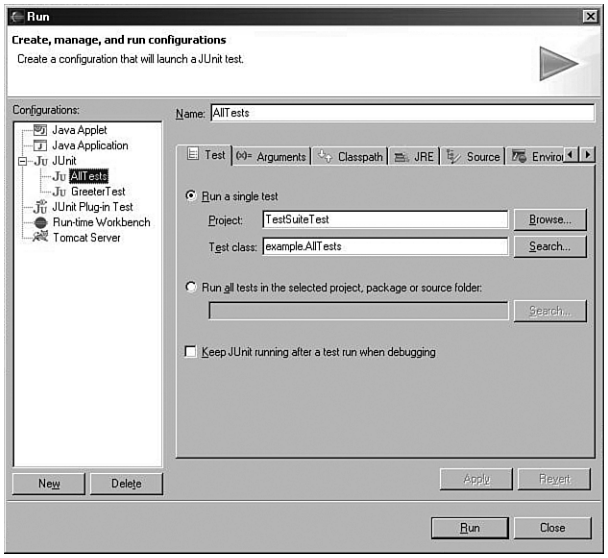
3 types selected.

FIGURE 6.8 The Recreate Test Suite dialog listing the available JUnit tests that can be added to `suite()`.

In addition, the JUnit Wizard only creates test suites made up of complete tests. If you want to have the `suite()` method call certain test methods, you have the task of adding by hand the calls to `addTest()` in `suite()`.

Custom JUnit Launch Configurations

By selecting to run the JUnit test (which is either a subclass of `TestCase` or a test suite) using Run, Run As, JUnit Test, you allow the Launcher to create a default launch configuration and execute the TestRunner GUI using this default configuration. If you were to open the Run dialog (by selecting Run, Run), you would see in the Configurations list to the left the JUnit category with `GreeterTest` as a configuration entry (you did not get the opportunity to create all the classes for the test suite example, so there is no launch configuration for it). Running a JUnit test with the default launch configuration will do the job for most runs; however, creating a custom launcher lets you decide whether you want to run one test or many (see Figure 6.9).


Run

Create, manage, and run configurations

Create a configuration that will launch a JUnit test.

Configurations:
- Java Applet
- Java Application
- JUnit
  - AllTests
  - GreeterTest
- JUnit Plug-in Test
- Run-time Workbench
- Tomcat Server

Name: AllTests

(●) Run a single test

Project: TestSuiteTest

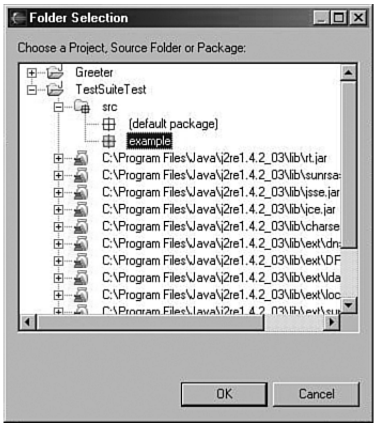
Test class: example.AllTests

( ) Run all tests in the selected project, package or source folder:

[ ] Keep JUnit running after a test run when debugging

FIGURE 6.9 The Run dialog of the Eclipse Launcher.

The Launcher Run dialog has the base functionality discussed in previous chapters (configurable name, configurable environment variables, and sharable launch configurations as well as the ability to create new configurations, pass arguments both to the VM and the running class, select a JRE, and have a custom source path). The Test tab allows you to do the following:



Choose a Project, Source Folder or Package:

```
Greeter
TestSuiteTest
    src
        (default package)
        example
    C:\Program Files\Java\j2re1.4.2_03\lib\rt.jar
    C:\Program Files\Java\j2re1.4.2_03\lib\sunrsa
    C:\Program Files\Java\j2re1.4.2_03\lib\jsse.jar
    C:\Program Files\Java\j2re1.4.2_03\lib\jce.jar
    C:\Program Files\Java\j2re1.4.2_03\lib\charse
    C:\Program Files\Java\j2re1.4.2_03\lib\ext\dn
    C:\Program Files\Java\j2re1.4.2_03\lib\ext\DF
    C:\Program Files\Java\j2re1.4.2_03\lib\ext\lda
    C:\Program Files\Java\j2re1.4.2_03\lib\ext\loc
```

FIGURE 6.10 The Folder Selection dialog of the Eclipse Launcher Test tab.

- **Run a single test**—You can use the Browse button to select a new project, or you can use the Search button to display the list of available tests. You can only select one.
- **Run all the tests from either a project, a package, or a folder**—Selecting the radio button for this choice enables the Select button, which, when clicked, displays a dialog listing the available projects, packages, and folders (see Figure 6.10).

Finally, the check box labeled Keep JUnit Running After a Test Run When Debugging is useful only if you are using a JDK that supports the hot-swapping of code. When you are debugging a JUnit test and you modify and recompile the code, the hot swap will only work if you have checked the Keep JUnit Running… box.

To create a new JUnit configuration, remember to select either the JUnit category in the Configurations list or one of the JUnit configurations and then click New.

Extensions to JUnit

Not to be outdone in the open-source marketplace for testing frameworks, a number of additional packages have emerged that allow testing of Java technologies that are not easily tested due to their inclusion in application frameworks. For example, servlets, JSPs, and EJBs only run within a container framework that the average JUnit test cannot easily test against.

Developers abhor a vacuum. Almost before anyone realized how cool JUnit really was, extensions were already being written to allow for the testing of client-side and server-side Java components in their native environment. Here are some examples:

- **Cactus**—A server-side testing framework for the testing of servlets, EJBs, and other server-side Java technologies.
- **HttpUnit/HtmlUnit/jWebUnit**—Various frameworks for testing Web applications. The best thing about testing at this level is that the tests are really user-level tests. Pick a use case and write your HttpUnit, HtmlUnit, or jWebUnit test as proof that the use-case scenario works based on how the user acts and reacts to the system and that the system behaves predictably when the use-case failure scenarios occur.

- **JMXUnit**—A testing framework for JMX Beans.
- **StrutsTestCase**—A testing framework for Struts.
- **VirtualMock**—An AOP-based framework that uses the concept of *mock objects*, which allow for the testing of objects without concern as to the objects the objects being tested call. A mock object is an object that returns predictable values to the object under test. For example, database functionality is always difficult to test due to the necessary setup involved in making the tests repeatable. If the objects to be tested call mock database objects, you can test these objects based on what they believe they are receiving rather than what they would actually receive.

Please do not take inclusion in this list as an endorsement as to the usefulness of the these frameworks. Their usefulness will vary depending on your needs and where the developers of the frameworks are with their projects. However, performing a quick search on SourceForge results in the following available testing frameworks and utilities: JFCUnit, StrutsTestCase, jWebUnit, JUnitEE, Pounder, Artima Suite Runner, Cricket Cage, JUnitEJB, JXUnit, GroboUtils, JUnitDoclet, NoUnit, AgileDox JUnit-addons, Quilt, JUB, Hansel, and JFunc. This is not a complete list, and I expect it to grow.

Cactus has emerged as one of the more popular server-side testing pieces for Java. It tests server-side components from the safety of a client-side position.

Using a Non-Plug-in-Based JUnit Extension

As of the writing of this chapter, a Cactus plug-in for Eclipse has not seen the light of day (well, it did and then it was pulled back into the closet kicking and screaming). However, all is not lost. You can still write Cactus tests with the existing JUnit Wizard. A nontrivial example will explain how to set up everything needed to make that happen.

Let's begin by installing Tomcat and Cactus. When those two items are ready to go, you will write a sample servlet and its test and then display the results of the test.

In order to run a servlet, you need a servlet engine. If you don't already have Tomcat, you can download it from `http://jakarta.apache.org/site/binindex.cgi` and install it in your favorite location (for example, `c:\tools\jakarta-tomcat-5.0.25`). From the same Web page, you can also download Cactus (scroll up to look for it and install Cactus into your favorite location—for example, `c:\tools\jakarta-cactus-13-1.6.1`). Your versions will probably be newer.

At this stage, you have completed the most difficult part of this example: downloading and installing the latest versions of Tomcat, and Cactus. Cactus needs no configuration, and you will configure Eclipse to run Tomcat from the Eclipse Launcher.

In a more complete J2EE environment, such as MyEclipse from Genuitec, there would be editors, wizards, and builders specific to servlets and JSPs. However, due to the addition of Tomcat support in Eclipse 3.0, you can develop basic J2EE applications by setting up the proper directory structure in your project and creating a valid `web.xml` file.

Create a servlet and deploy it to Tomcat using the following steps:

1. Create a new project by pressing Ctrl+N. When the New dialog opens, select Java Project and then click Next. Enter the project name **CactusTest** and click Next. In the Java Settings page on the Source tab, set the Default Output Folder to `CactusTest/WEB-INF/classes`. Click the Libraries tab, click Add External JARs, and navigate to `<Tomcat Install Directory>/common/lib`. Select `servlet-api.jar` and click Open. Click Finish to close the New dialog. The use of the `WEB-INF` directory gives you the Web application structure you will need when you deploy to Tomcat, and `servlet-api.jar` gives you the J2EE symbols needed to compile servlets.

2. Create a servlet class by pressing Ctrl+N. From within the New dialog, select Java, Class and then click Next. Enter `example` for Package, `HelloWorldServlet` for Name, and `javax.servlet.http.HttpServlet` for Superclass. Click Finish.

3. `HelloWorldServlet` will open in the Java editor. Right-click in the editor and click Source, Override/Implement Methods. When the Override/Implement Methods dialog opens, check `doGet()` and click OK. Delete the `TODO` line and `super.doGet(arg0, arg1)`. Add code to get the `PrintWriter` from the `HttpServletResponse` object and write a message. If you don't add code to produce some output, the servlet will fail to run:

```
protected void doGet(HttpServletRequest arg0, HttpServletResponse arg1)
    throws ServletException, IOException {
    PrintWriter out = arg1.getWriter();

    out.print("This is a message from the HelloWorldServlet!");
}
```

   Fix any compile errors by pressing Ctrl+Shift+O to add any missing imports. Save the file.

4. Create a `web.xml` file by pressing Ctrl+N. From the New dialog, select Simple, File and then click Next. Make sure the parent folder is `CactusTest/WEB-INF` and enter `web.xml` as the filename (see Figure 6.11). Click Finish.

5. In the `web.xml` file, create a Web application entry that maps your `HelloWorldServlet` to the servlet name `helloworld`. This allows you to access the servlet using the name `helloworld`. Save the file:

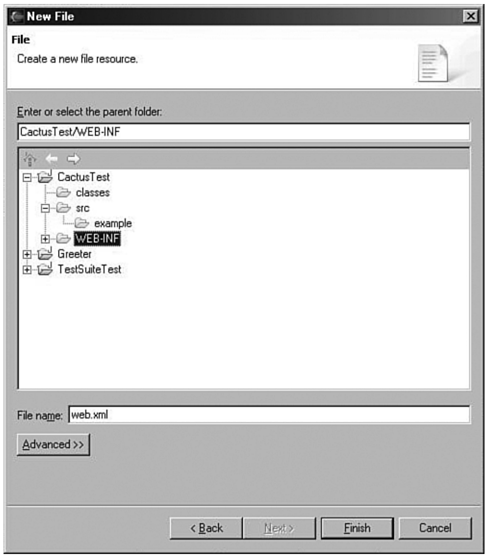
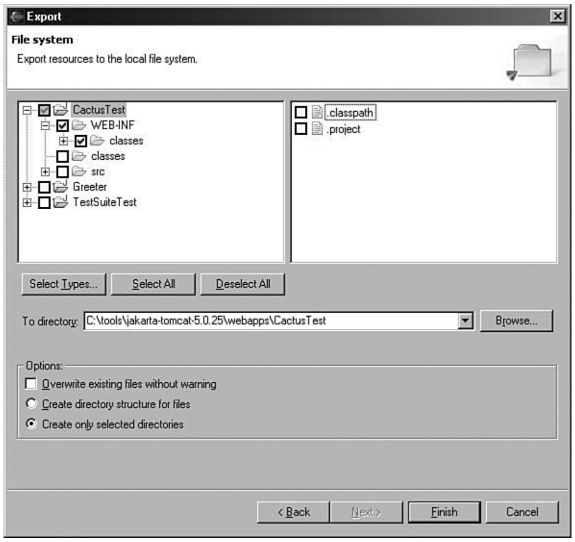


FIGURE 6.11 The New dialog preparing to create an empty `web.xml` file.

```
<!DOCTYPE web-app PUBLIC
  '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
  'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>
<web-app>
  <servlet>
    <servlet-name>helloworld</servlet-name>
    <servlet-class>example.HelloWorldServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>helloworld</servlet-name>
    <url-pattern>/helloworld</url-pattern>
  </servlet-mapping>
</web-app>
```



To directory: C:\tools\jakarta-tomcat-5.0.25\webapps\CactusTest

Options:
- [ ] Overwrite existing files without warning
- ( ) Create directory structure for files
- (●) Create only selected directories

FIGURE 6.12 The Export dialog, set up to copy the project files into the Tomcat webapps directory.

6. Deploy the CactusTest project to Tomcat. The simplest way to do that is to export CactusTest into the Tomcat `webapps` directory. In the Package Explorer, right-click the CactusTest project and select Export from the pop-up menu. When the Export dialog opens, select File System and click Next. On the File System page, select the CactusTest project in the window to the left and uncheck the `.classpath` and `.project` files in the window to the right. Click the plus sign to the left of the CactusTest node to reveal the `WEB-INF`, `classes`, and `src` directories. Check only the `WEB-INF` directory. This will export everything under `WEB-INF`, including `classes` and `web.xml`. The To Directory field needs to reference the Tomcat `webapps` directory. Click Browse and navigate to your Tomcat installation directory. When you find it, select it and click New Folder. Enter the folder name `CactusTest`, press Enter, and click OK to see the export directory target (see Figure 6.12). If the To directory does not display a path that includes `CactusTest`, click Browse, navigate to it, and click OK. Click Finish to deploy your project to Tomcat.

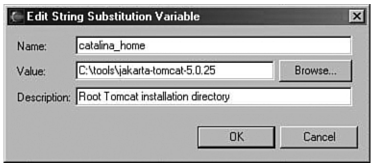
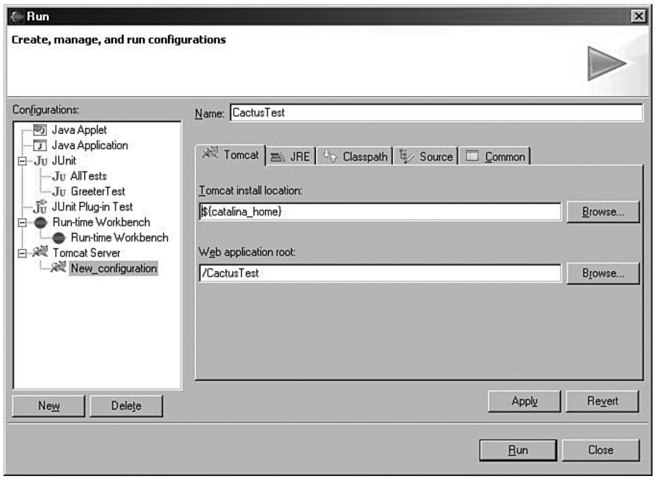

Edit String Substitution Variable
Name: catalina_home
Value: C:\tools\jakarta-tomcat-5.0.25
Description: Root Tomcat installation directory

FIGURE 6.13 The `catalina_home` variable set with the Tomcat home directory.

7. In order to start Tomcat from within Eclipse, you need to create a Tomcat launch configuration. Eclipse already defines a string-substitution variable called `catalina_home`, but it is not assigned a default value. From the main menu, select Window, Preferences and Run/Debug, String Substitution. The Variable column lists `catalina_home` with an empty Value. Select the `catalina_home` row and click Edit. When the Edit String Substitution Variable dialog opens, click Browse, navigate to the Tomcat installation directory, and click OK to return to the Edit String Substitution Variable dialog (see Figure 6.13). Click OK to close the Edit String Substitution Variable dialog and then click OK to close the Preferences dialog.

8. From the main menu, select Run, Run. When the Run dialog opens, select Tomcat Server from the Configurations list and then click New. For Name, enter `CactusTest`, and for Web Application Root, enter `/CactusTest` (see Figure 6.14). Click Run. When the Run dialog closes, the Console view will come forward and begin to output Tomcat logging statements. You are ready to check your servlet when the Console has a statement similar to this:

```
INFO: Server startup in 9494 ms
```


Run
Create, manage, and run configurations
Configurations:
- Java Applet
- Java Application
- JUnit
  - AllTests
  - GreeterTest
- JUnit Plug-in Test
- Run-time Workbench
  - Run-time Workbench
- Tomcat Server
  - New_configuration

Name: CactusTest
Tomcat install location: ${catalina_home}
Web application root: /CactusTest

FIGURE 6.14 The Run dialog with a new Tomcat Server configuration for the CactusTest Web application.

9. Open a browser and set the URL to `http://localhost:8080/CactusTest/helloworld`. You should see your message in the browser (see Figure 6.15).

http://localhost:8080/CactusTest/helloworld - Opera
File Edit View Navigation Bookmarks Mail Window Help
Back Forward Reload Home Hotlist Wand New Mail
BUY OPERA TODAY!
And make this banner go away.
OPERA software
Opera Opera Community OperaMail Super search Price Comparison search Amazon.com search
Weird ... BZFlag ... Keyhole gameju... Chapter... New R... java.ne... Products http://l... The Ja... http://l...
http://localhost:8080/CactusTest/helloworld
Go Google search 100%

FIGURE 6.15 The `HelloWorldServlet` output displayed in a browser.

At this point, you now have a working servlet that in practice may or may not have a JSP to which it is redirecting. Now let's take a look at what it would take to run a test on it.

The intention of this part of the chapter is not to explain the hows and whys of Cactus, but to show you how you can implement Cactus tests even without direct Cactus support within Eclipse. However, in order to appreciate what needs to be done, you will need a certain amount of Cactus background.

Cactus, as a server-side testing framework, has both a client-side and a server-side component. The server-side piece runs on the same server as the servlet, and the client-side component receives messages from the server-side piece via sockets. In order for that to work, you will need to update the Web app `web.xml` file and carry along certain JAR files that must be deployed on the server with the servlet and the servlet test code.

Here's the short list of things that need to be done to write Cactus tests in Eclipse:

1. Copy a collection of JAR files needed by Cactus into `WEB-INF/lib`. This is for the Cactus test that runs on the Web server. These will be deployed along with the servlet to be tested.
2. Add a collection of JAR files to the classpath of the project. This way, you can run the Cactus test as a client so that it can connect to the server-side Cactus test.
3. Update `web.xml` to include the Cactus filter and servlet mappings.
4. Write a Cactus test either by extending `ServletTestCase` or by instantiating a `ServletTestCase` object within a JUnit test. If Tomcat is properly set up within Eclipse, you can run the Cactus test as a standalone JUnit test and it will communicate with the server-side code returning the results of the tests.

Cactus can be used to test servlets, JSPs, EJBs, and any server-side Java components. For this example, because you are going to write a servlet test, you will not look at some of the other test classes you could have extended. The CactusTest project will contain both the servlet to be tested and the test code.

Let's fill in the preceding steps. Before you create the test class, you need to import the following JAR files into the `WEB-INF/lib` directory for use by the Cactus server-side piece (all the JAR files are from the Cactus install directory):

- `aspectjrt-1.1.1.jar`
- `cactus-1.6.1.jar`
- `commons-logging-1.0.3.jar`
- `commons-httpclient-2.0.jar`
- `junit-3.8.1.jar`

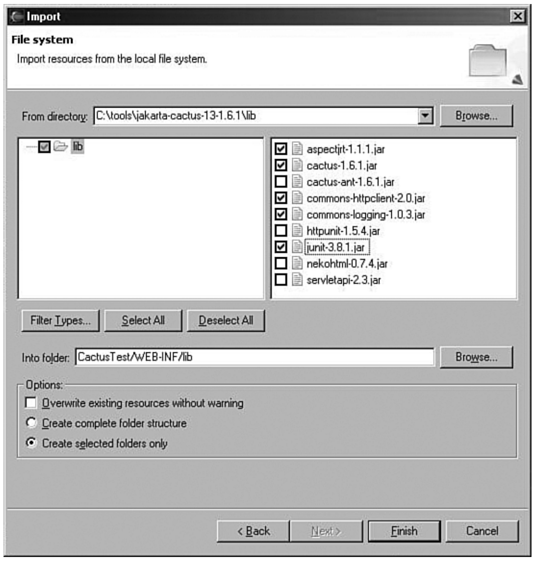
Add these to your `CactusTest project` by right-clicking `WEB-INF` and selecting Import from the pop-up menu. Because you want to copy the JAR files into the project, when the Import dialog opens, from the Select page choose File System as the import type and click Next. From the File System page of the Import dialog, click the top Browse button to navigate to the `lib` directory of the Cactus installation (for example, `C:\tools\jakarta-cactus-13-1.6.1\lib`) and select the `lib` node from the tree view in the window to the left. When you select the `lib` node, the available JAR files in that directory will be listed in the window to the right. Select the five JAR files listed previously (even if the version numbers are different—they should be higher [see Figure 6.16]). Change the Into Folder from

```
CactusTest/WEB-INF
```

to

```
CactusTest/WEB-INF/lib
```

and click Finish. Your WEB-INF directory now has a `lib` directory, and it contains the Cactus JARs. These JAR files will be used by your test when it runs on the server.



Import
File system
Import resources from the local file system.

From directory: C:\tools\jakarta-cactus-13-1.6.1\lib

☑ lib

- ☑ aspectjrt-1.1.1.jar
- ☑ cactus-1.6.1.jar
- ☐ cactus-ant-1.6.1.jar
- ☑ commons-httpclient-2.0.jar
- ☑ commons-logging-1.0.3.jar
- ☐ httpunit-1.5.4.jar
- ☑ junit-3.8.1.jar
- ☐ nekohtml-0.7.4.jar
- ☐ servletapi-2.3.jar

Into folder: CactusTest/WEB-INF/lib

Options:
- ☐ Overwrite existing resources without warning
- ○ Create complete folder structure
- ● Create selected folders only

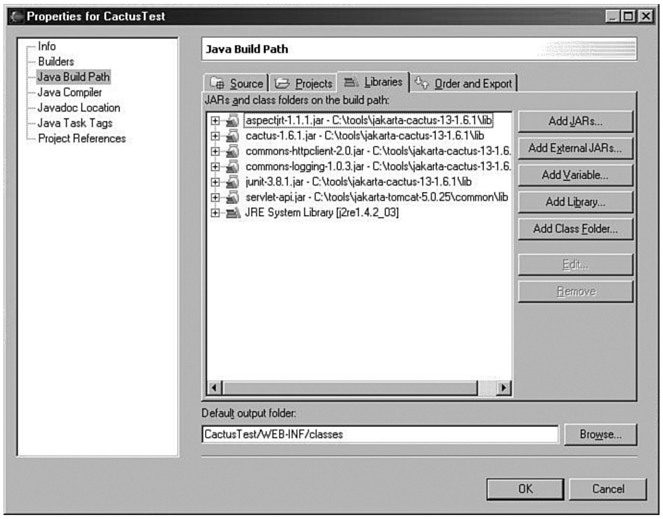
FIGURE 6.16 The Import dialog listing the JAR files found in the Cactus installation `lib` directory.

The next step is to add a collection of JAR files needed by Cactus onto the project classpath. Open the project's Properties dialog by right-clicking the CactusTest project name in the Package Explorer and selecting Properties from the pop-up menu. Select Java Build Path from the list to the left and the Libraries tab from the Java Build Path page to the right. Click Add External JARs and navigate to your Cactus installation `lib` directory (again, for example, `C:\tools\jakarta-cactus-13-1.6.1\lib`). Holding down the Ctrl key, select the following JAR files:

- `aspectjrt-1.1.1.jar`
- `cactus-1.6.1.jar`

- `commons-httpclient-2.0.jar`
- `commons-logging-1.0.3.jar`
- `junit-3.8.1.jar`

Click Open to select these five JAR files. They will be added to the Libraries tab's build path (see Figure 6.17). Click OK.



JARs and class folders on the build path:

```
aspectjrt-1.1.1.jar - C:\tools\jakarta-cactus-13-1.6.1\lib
cactus-1.6.1.jar - C:\tools\jakarta-cactus-13-1.6.1\lib
commons-httpclient-2.0.jar - C:\tools\jakarta-cactus-13-1.6.
commons-logging-1.0.3.jar - C:\tools\jakarta-cactus-13-1.6.
junit-3.8.1.jar - C:\tools\jakarta-cactus-13-1.6.1\lib
servlet-api.jar - C:\tools\jakarta-tomcat-5.0.25\common\lib
JRE System Library [j2re1.4.2_03]
```

Default output folder: CactusTest/WEB-INF/classes

FIGURE 6.17 The five JAR files needed from the Cactus installation by the client-side Cactus test.

The third step is to add the Cactus-specific filter and servlet information into the Web application `web.xml` file:

```
<!DOCTYPE web-app PUBLIC
  '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
  'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>
<web-app>
    <filter>
        <filter-name>FilterRedirector</filter-name>
        <filter-class>org.apache.cactus.server.FilterTestRedirector</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>FilterRedirector</filter-name>
        <url-pattern>/FilterRedirector</url-pattern>
    </filter-mapping>
    <servlet>
      <servlet-name>helloworld</servlet-name>
      <servlet-class>example.HelloWorldServlet</servlet-class>
    </servlet>

    <servlet>
        <servlet-name>ServletRedirector</servlet-name>
        <servlet-class>org.apache.cactus.server.ServletTestRedirector</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ServletRedirector</servlet-name>
        <url-pattern>/helloworld/ServletRedirector</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>helloworld</servlet-name>
       <url-pattern>/helloworld</url-pattern>
    </servlet-mapping>
</web-app>
```

This `web.xml` file is the complete file you will use. It contains both the `HelloWorldServlet` servlet mapping as well as the filter and servlet information needed to call Cactus (the shaded code). The only item of note is the way the Cactus `ServletRedirector` servlet is defined: Because our `HelloWorldServlet` is mapped to the name `helloworld`, the `ServletRedirector` must be defined relative to the servlet to be tested, so `/helloworld` is placed before `ServletRedirector` in the `<url-pattern>` tag of the servlet mapping.

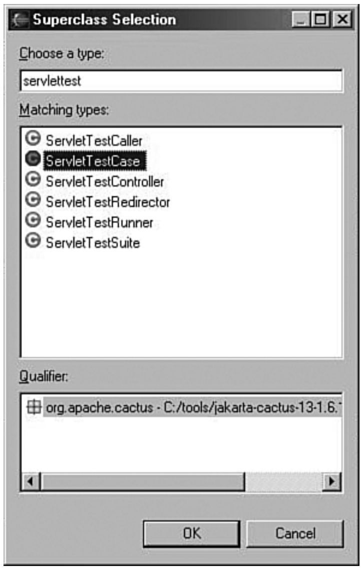


FIGURE 6.18 The Superclass Selection dialog displaying the Cactus `ServletTestClass`.

After all that setup, it is time to create a Cactus test. Select `HelloWorldServlet` in the Package Explorer view. Create a class that extends `ServletTestCase` by pressing Ctrl+N to open the New dialog and selecting Java, JUnit, JUnit Test Case. Click Next. In the JUnit Test Case page, all the standard information should be filled in for you. The one change you need to make is to change the superclass to `ServletTestCase`. Click Browse, enter `ServletTest` in the topmost text field of the Superclass Selection dialog, and select `ServletTestCase` (see Figure 6.18). Click OK to close the Superclass Selection dialog. Click Finish on the New JUnit Test Case page to close the dialog and accept the class information. The `HelloWorldServletTest` opens in the Java editor and is quite empty.

A standard JUnit class has a setUp() method, a tearDown() method, and one or more test methods that start with the word *test*. Cactus can use the setUp() and tearDown() methods, but it also adds the capability to add custom setup and teardown methods per test. The custom setup and teardown methods are called begin<name of method being tested>() and end<name of method being tested>(). The test method for the servlet is going to test the doGet() method, so you could create three methods: beginDoGet(), endDoGet(), and testDoGet(). For this example, you will only create the last two. To check whether the call to doGet() streamed information to the caller, you are going to check the servlet output in endDoGet().

For the purposes of this exercise, you are only going to look at calling doGet(), but Cactus allows you to call any method defined in the servlet. Two objects created by Cactus to predictably test servlets are WebRequest and WebResponse. WebRequest represents an HTTP request message to the servlet, and WebResponse represents the HTTP response from the servlet. When endDoGet() is called, its single argument is going to be a WebResponse object. You will use it to find out what the servlet printed out, as shown here:

```
package example;
import java.io.IOException;

import javax.servlet.ServletException;

import org.apache.cactus.ServletTestCase;
import org.apache.cactus.WebResponse;


public class HelloWorldServletTest extends ServletTestCase {

    public void testDoGet() {
        HelloWorldServlet servlet = new HelloWorldServlet();

        try {
            servlet.doGet(request, response);
        } catch (ServletException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void endDoGet(WebResponse response) {
        String actual = response.getText();
        String expected = "This is a message from the HelloWorldServlet!";
        assertNotNull(actual);
        assertEquals(expected, actual);
    }
}
```

The `testDoGet()` and `endDoGet()` methods are quite small. To test a servlet, Cactus asks that you instantiate your servlet in the test method as well as check its response in the end method. The code in `testDoGet()` does the following:

- Instantiates the servlet.
- Calls `doGet()` and passes in predefined request and response objects. The `try/catch` code is there as protection in case the method call throws an exception.

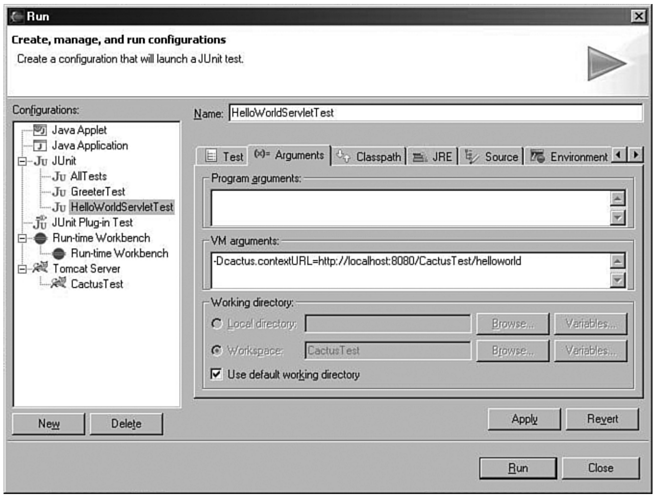
As you'll recall, the `doGet()` method of `HelloWorldServlet` gets the `PrintWriter` object and then prints a greeting. The `testDoGet()` method makes the call to `doGet()`, and the test method `endDoGet()` gets the text from the response object and compares what it finds to what it was expecting.

Your Web application needs to be deployed to Tomcat. Right-click the CactusTest project and select Export from the pop-up menu. Select File System from the Select page and click Next. Select and open the CactusTest node and uncheck `classes` and `src` in the list to the left and uncheck `.classpath` and `.project` in the list to the right. Make sure To Directory is set to the Tomcat `webapps/CactusTest` directory. Click Finish. When the Question dialog opens asking if you want to overwrite existing files, click Yes To All.

There is only one thing left to do before you run the test: You need to create a launch configuration to give the Cactus framework the URL to use in calling the servlet. Select `HelloWorldServletTest` in the Package Explorer and select Run, Run from the main menu. In the Run dialog, select any of the JUnit tests or the JUnit node and click New. A new configuration page appears with as much information as the system could glean from the file you selected before opening the dialog. The only thing left for you to configure is the URL for Cactus to use in testing the servlet. Even though you could pass Cactus the URL to use by setting up a `cactus.properties` file, you will instead put the URL in an environment variable that the JVM will place in the runtime environment. Click the Arguments tab and enter the following in the VM Arguments area (see Figure 6.19):

```
-Dcactus.contextURL=http://localhost:8080/CactusTest/helloworld
```

Click Apply and then click Close to shut down the Run dialog. You're done! Time to run the test.



Create, manage, and run configurations

Create a configuration that will launch a JUnit test.

Configurations:

Name: HelloWorldServletTest

Program arguments:

VM arguments:

-Dcactus.contextURL=http://localhost:8080/CactusTest/helloworld

Working directory:

Local directory:

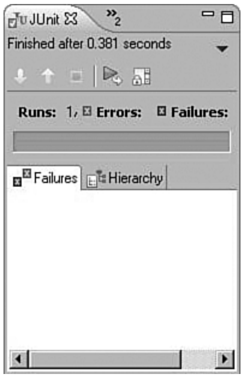
Workspace: CactusTest

☑ Use default working directory

FIGURE 6.19 The Run dialog with the environment variable entry for `cactus.contextURL`.

If you have not stopped Tomcat since you last started it, go to the Console view and click the red square button located on its toolbar (it is the first button on the left). The Console view title will display a message similar to the following:

```
Console (<terminated> CactusTest [TomcatServer] ...
```



Finished after 0.381 seconds

Runs: 1/ Errors: Failures:

FIGURE 6.20 The JUnit TestRunner displaying the green bar of success for the Cactus servlet test.

Restart Tomcat from within Eclipse by opening the Launch Configuration dialog (select Run, Run and then select Tomcat Server, CactusTest). Click Run. The Console view should have fresh output from the Tomcat server. When the Console states that the server has started, you are ready to run the Cactus test using JUnit.

Check that the servlet is available by opening a browser and going to `http://localhost:8080/CactusTest/helloworld`. You should see the same page as before with your welcome message displayed.

Select `HelloWorldServletTest` in the Package Explorer. From the main menu select Run, Run As, JUnit Test. The Launcher will use the launch configuration you just created and will start the JUnit TestRunner with `HelloWorldServletTest` as its only test class. If all goes well, the TestRunner GUI will open and display the green bar of success (see Figure 6.20). Click the Hierarchy tab to see that `testDoGet()` was run and that it succeeded.

If you already had the TestRunner GUI open, it may not come to the front if the test passes, so click the JUnit tab to see it.

The test that just ran went through the following steps, assuming Tomcat is running and the servlet is available (this is a very abbreviated version of the actual process):

- The client code, `HelloWorldServletTest`, ran and performed standard JUnit initialization.
- The Cactus framework looked for the `beginDoGet()` method. Because you did not write `beginDoGet()`, this did not do anything.
- Cactus opened a connection to the `ServletRedirector` running within Tomcat. The `ServletRedirector` was deployed with the `HelloWorldServlet` and will take care of running the server-side component of the test.
- `ServletRedirector` reinstantiated `HelloWorldServletTest` on the server side within Tomcat and called `testDoGet()`, which created an instance of the `HelloWorldServlet`. Because `HelloWorldServletTest` was running within the servlet engine, it safely created `HelloWorldServlet` and was able to call any and all methods available in the `HelloWorldServlet` API.
- When `testDoGet()` completed, `ServletRedirector` returned to the client side, which promptly called `endDoGet()`. The `endDoGet()` method received the `WebResponse` object, which contained the text output by `HelloWorldServlet`.

All this is independent of Eclipse. The most interesting thing about the Cactus test is that you are able to run the server side and client side within one environment.

So what did it really take to complete these steps? You copied a number of JAR files into `WEB-INF/lib` so that the server-side code would work. You also updated the `web.xml` file to support the Cactus `ServletRedirector`, and you put a number of JAR files in the project's classpath so the client-side component would work. You wrote a test that extended a Cactus base class, you deployed the code to Tomcat, and then you ran your test code as a JUnit test.

This example may have seemed like it needed a lot of setup. In the normal course of your day-to-day development, you will probably use a more comprehensive plug-in to make the development of your J2EE application simpler. Plug-ins such as Lomboz and MyEclipse not only control the configuration, starting, and stopping of application servers, but they also create support files like `web.xml`.

Perhaps by the time this book comes out the Cactus Eclipse plug-in will be released again, and the preceding steps will be reduced a bit. Go to `http://jakarta.apache.org/cactus/integration/eclipse/index.html` to check on the plug-in's progress. Better yet, join in and help to get it ready.

Ant Support for JUnit

Ant already has a task tag called `junit` that will take care of running your tests in a batch mode. In addition, Ant also has a task called `junitreport` that will take care of generating a report with useful information about the test run.

Many JUnit (or JUnit-like) extensions are available for you to choose from. The amount of support varies, depending on the current popularity of the tool and whether or not it behaves well with existing frameworks. Take a stroll through JUnit.org and SourceForge.net to look for help in your testing efforts.

To quote Mae West, "Too much of a good thing can be wonderful!"

SHOP TALK

Test/Integrate Daily

Something we will not discuss at any length is the fact that even though we develop our software using tools such as Eclipse, we will rarely deploy the files created by the IDE. Build tools, such as Ant, should be used to extract, build, and deploy the final production systems. In many cases, what is missing is the additional step of checking the integrity of the system about to be deployed.

In order to prove the integrity of a system, it is necessary to run all the tests written by the various developers in one giant testing marathon. Of course, if the first time you run all the tests needed by the system is before deployment, you were in trouble long before you got out of bed. The concept of continuous integration has been gaining favor as something that should be done to any nominally nontrivial system as a way of checking the system every day. No, you did not read that wrong. Every day. Multiple times a day.

And if you have inherited a system already in production with no supporting tests, remember that you can add tests as bugs are reported. Rather than spend all your time dreaming up tests to check the code, just write tests that prove a reported bug exists and that it has been fixed. If you write a test every time a bug is reported, not only will the number of bugs be reduced, they will not reappear.

Anyway, many books and articles are available to you to begin looking up information about continuous integration and the tools that support it, such as CruiseControl. Appendix E, "Recommended Resources," lists just a few Web sites to begin your journey to more predictable software.

In Brief

The Eclipse support for JUnit is quite extensive and convenient. The JUnit TestRunner is a full-fledged SWT GUI and is well-integrated with the Java Development Tooling plug-in. As an accepted standard, JUnit goes a long way in encouraging the use of tests in day-to-day development and continuous integration. JUnit's small API makes it easy to learn and easy to use.

In this chapter you looked at the following JUnit capabilities:

- Implementing a JUnit test involves writing a Java class and selecting it before opening the New dialog to select the JUnit Test Case. The results of JUnit tests are viewed through the TestRunner GUI.
- The creation and execution of JUnit tests are supported from within Eclipse.
- The JDT debugger let's you set breakpoints and debug JUnit tests.
- JUnit test suites can be created using the JUnit Wizard. You can re-create the test suite through the wizard as well as modify the test suite by hand.
- There are many JUnit extension frameworks. Non-plug-in JUnit extensions, such as Cactus, can be included in your projects in two ways: by inheriting from a base class such as `ServletTestCase` or by instantiating objects of type `ServletTestCase` in your JUnit test.
- Tomcat and Cactus can be downloaded from the Jakarta site for use by Eclipse in the development of Web applications.
- Cactus programs to test servlets can be implemented and run from within Eclipse using the built-in support for Tomcat and the JUnit launcher support.