CHAPTER 4
JUnit

4.0 Introduction

Unit testing is at the heart of XP, and it is a central theme of this book. JUnit,* available from http://www.junit.org, is the de facto standard for Java unit testing. It is a simple framework for creating automated unit tests. JUnit test cases are Java classes that contain one or more unit test methods, and these tests are grouped into test suites. You can run tests individually, or you can run entire test suites.

Ant includes the junit task for running JUnit tests. We show how to run JUnit tests using Ant in Chapter 3.

Each JUnit test method should execute quickly. Speed is important because as more tests are written and integrated into the build process, it takes longer to run the entire test suite. Programmers do not want to be interrupted for long periods of time while tests run—so the longer the tests take to execute the greater the likelihood programmers will skip this critical phase.

You can also increase the likelihood that programmers will run the tests by making it extremely easy, preferably with a single command. The ability to run all tests with a single command or button click is nearly a requirement to claim that your project is doing XP. We showed how to run tests with Ant in the previous chapter, and many IDEs now make it possible to run tests by clicking on a menu item.

JUnit tests are pass/fail tests explicitly designed to run without human intervention. Because of this design, you can (and should) add your test suite to your continuous integration build process so the tests run automatically.

* We cover JUnit Version 3.8.1 in this chapter.
4.1 Getting Started

Problem
You want to write unit tests with JUnit.

Solution
Create a subclass of junit.framework.TestCase. Each unit test is represented by a testXXX() method within the TestCase subclass.

Discussion
Example 4-1 shows an extremely simple test case. A test case is a subclass of TestCase and contains a collection of unit tests. Instances of TestCase are sometimes referred to as test fixtures, although we prefer to say “test case” since that matches the class name. Each unit test is a public, no-argument method beginning with “test”. If you do not follow this naming convention, JUnit will not be able to locate your test methods automatically. Instead, you would have to write a suite() method and construct instances of your test case, passing the test method name to the constructor.

Example 4-1. Simple test case
package com.oreilly.javaxp.common;
import junit.framework.TestCase;
/**
 * Sample unit tests for the {link Person} class.
 */
public class TestPerson extends TestCase {
    /**
     * This constructor is only required in JUnit 3.7 and earlier.
     * @param testMethodName the name of the test method to execute.
     */
    public TestPerson(String testMethodName) {
        super(testMethodName);
    }

    /**
     * A unit test to verify the name is formatted correctly.
     */
    public void testGetFullName() {
        Person p = new Person("Aidan", "Burke");
        assertEquals("Aidan Burke", p.getFullName());
    }

    /**
     * A unit test to verify that nulls are handled properly.
     */
In JUnit 3.7 and earlier, the constructor is required and must have the signature shown in the TestPerson class. JUnit uses this constructor to create a new instance of the test case as it runs each of the unit test methods. The name argument matches the current unit test’s method name, allowing JUnit to use reflection to invoke the corresponding method. JUnit 3.8 removed the need for this constructor, so we will not include it in the remaining examples in this chapter.

The “test” methods are the actual unit tests. You must have at least one unit test in each test case or JUnit reports an error. Our TestPerson class has two unit tests, each of which checks different aspects of the Person class’s getFullName() method. Test methods should follow this signature:

```java
public void test<something>() [throws SomeException]
```

This naming convention allows JUnit to locate unit tests by reflection. Tests may throw any subclass of `java.lang.Throwable`. When this happens, JUnit catches the exception and reports a test error. It continues to execute any additional test methods.

Each unit test uses various `assertXXX()` methods to do the actual testing:

```java
assertEquals("Aidan Burke", p.getFullName());
```

This method confirms that its two arguments are equal. If the arguments are equal, the test passes. Otherwise, a test failure is reported and the remainder of the current test method is skipped. JUnit does proceed to execute other test methods, however. In the case of `Object` arguments (such as two Strings), the `.equals()` method is used for checking equality.

To compile TestPerson, include `junit.jar` in your classpath. The next recipe shows how to run the tests.

**See Also**

Recipe 4.2 shows how to run your tests. Recipe 4.3 explains the `assert()` methods. Recipe 4.4 describes how fine-grained your tests should be.

* You could adopt a different naming convention; however, JUnit would not automatically find your test methods. You would have to build your test suite manually by constructing instances of your test case, passing your method names to the constructor.
4.2 Running JUnit

Problem
You want to run your tests.

Solution
We have already demonstrated how to run JUnit using Ant, back in Chapter 3. In order to run tests from a script or in an IDE, include junit.jar in your classpath and then use the junit.textui.TestRunner class to run your tests in text mode. Use junit.swingui.TestRunner to run the tests in a Swing GUI.*

Discussion
JUnit can run tests in text or graphical mode. Text mode is faster, and is excellent for running tests as part of an automated build process. Graphical tests are more interesting to run, and can make it easier to analyze output from a large number of tests.

Text testing
Here’s an example session using the text-based TestRunner. The first line is typed at the prompt; the rest is output. The TestPerson class is the test case from the previous recipe.

```
java junit.textui.TestRunner com.oreilly.javaxp.junit.TestPerson
.F.F
Time: 0.02
There were 2 failures:
1) testGetFullName(com.oreilly.javaxp.junit.TestPerson)junit.framework.AssertionFailedError: expected:<Aidan Burke> but was:<AidanBurke>
at com.oreilly.javaxp.junit.TestPerson.testGetFullName(C:/cvsdata/java_xp_cookbook/examples/src/com/oreilly/javaxp/junit/TestPerson.java:24)
2) testNullsInName(com.oreilly.javaxp.junit.TestPerson)junit.framework.AssertionFailedError: expected:<? Burke> but was:<?Burke>
at com.oreilly.javaxp.junit.TestPerson.testNullsInName(C:/cvsdata/java_xp_cookbook/examples/src/com/oreilly/javaxp/junit/TestPerson.java:29)

FAILURES!!!
Tests run: 2, Failures: 2, Errors: 0
```

The first line of output shows a dot (.) as each test runs. Once you have dozens or hundreds of tests, the dots allow you to see that tests are progressing. JUnit also shows “F” for each failure:

```
.F.F
```

JUnit displays the cumulative time (in seconds), followed by a summary report of failures and errors. Both unit tests failed. The expected text didn’t match the existing text:

```
expected:<Aidan Burke> but was:<AidanBurke>
```

* Use junit.awtui.TestRunner for an older, AWT-based test runner.
Either our test is incorrect, or the Person class failed to insert a space between the first and last names. It’s the latter. The final line shows cumulative totals from the unit tests:

Tests run: 2, Failures: 2, Errors: 0

This indicates that a total of two tests ran, and both had failures. No tests had errors.

A test failure occurs when an assertXXX() statement fails. A test error occurs when a unit test throws an exception.

After fixing the Person class, we can run the tests again. We see the following output:

java junit.textui.TestRunner com.oreilly.javexp.junit.TestPerson

.. Time: 0.01
OK (2 tests)

**Graphical testing**

While text-mode testing is great for automated testing, it can be more interesting to watch your tests graphically, as in Figure 4-1. Here is the command to run the GUI:

java junit.swingui.TestRunner com.oreilly.javexp.junit.TestPerson

![JUnit Swing GUI](image)

**Figure 4-1. The JUnit Swing GUI**
The black-and-white figure does not illustrate the fact that the progress bar near the top of the screen is red, indicating one or more errors or failures. As the tests run, the progress bar fills from left to right.

The output is essentially the same as JUnit’s text UI; however, you can click on lines to see the message associated with each problem. This is a particular advantage of the graphical TestRunner when you have to sift through large numbers of problems.

Figure 4-2 shows the Test Hierarchy tab. This tab allows you to see which of the unit tests passed or failed, and allows you to re-run individual tests.

Figure 4-3 shows the output once all bugs are fixed and every test passes. You cannot tell, but the progress bar is now green.

**Reload classes every run**

On a final note, the JUnit GUI provides a checkbox allowing you to “Reload classes every run.” When checked, the JUnit ClassLoader reads the latest .class files for your tests each time they are run. This allows you to leave the GUI up while you recompile your source code. The new classes are loaded the next time you click the Run button.
See Also

Most Java IDEs are integrated with JUnit. Read your IDE documentation to learn how to run tests directly within the IDE. See Recipe 4.3 to learn how to provide more descriptive error messages. Chapter 3 shows how to run JUnit using Ant.

4.3 assertXXX() Methods

Problem

You want to use the various assertXXX() methods to test different conditions.

Solution

junit.framework.TestCase, the base class for all test cases, extends from junit.framework.Assert, which defines numerous overloaded assertXXX() methods. Your tests function by calling these methods.
Discussion

Table 4-1 summarizes the various `assertXXX()` methods that can be found in `junit.framework.Assert`. Although you could get by with using `assertTrue()` for nearly every test, using one of the more specific `assertXXX()` methods often makes your tests more understandable and provides good failure messages. This table is only a summary; each of the methods is overloaded as described shortly.

Table 4-1. Assert method summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert()</code></td>
<td>This method was deprecated in JUnit 3.7 because it interferes with the J2SE 1.4 <code>assert</code> keyword. You should use <code>assertTrue()</code> instead. This method was completely removed in JUnit 3.8.</td>
</tr>
<tr>
<td><code>assertEquals()</code></td>
<td>Compares two values for equality. The test passes if the values are equal.</td>
</tr>
<tr>
<td><code>assertTrue()</code></td>
<td>Evaluates a boolean expression. The test passes if the expression is false.</td>
</tr>
<tr>
<td><code>assertNotNull()</code></td>
<td>Compares an object reference to null. The test passes if the reference is not null.</td>
</tr>
<tr>
<td><code>assertNotSame()</code></td>
<td>Compares the memory address of two object references using the <code>==</code> operator. The test passes if both refer to different objects.</td>
</tr>
<tr>
<td><code>assertNull()</code></td>
<td>Compares an object reference to null. The test passes if the reference is null.</td>
</tr>
<tr>
<td><code>assertSame()</code></td>
<td>Compares the memory address of two object references using the <code>==</code> operator. The test passes if both refer to the same object.</td>
</tr>
<tr>
<td><code>assertTrue()</code></td>
<td>Evaluates a boolean expression. The test passes if the expression is true.</td>
</tr>
<tr>
<td><code>fail()</code></td>
<td>Causes the current test to fail. This is commonly used with exception handling, as shown in Recipe 4.12.</td>
</tr>
</tbody>
</table>

Optional first argument

All of the methods in Table 4-1 are overloaded to accept an optional `String` as the first argument. When specified, this argument provides a descriptive message should the test fail. Here is an example that shows two assert statements, one with the description and one without:

```java
assertEquals(employeeA, employeeB);
assertEquals("Employees should be equal after the clone() operation.",
            employeeA, employeeB);
```

The second version is preferable because it describes why the test failed, making it easier to fix problems down the road.

The message should describe what is asserted to be true, rather than what went wrong.
Equality comparison

The `assertSame()` method compares two object references, ensuring that they both refer to the same memory address. `assertSame()` uses the `==` operator for its comparison. The following two tests are functionally identical:

```java
assertSame("Expected the two parts to be identical.", part1, part2);
assertTrue("Expected the two parts to be identical.", part1 == part2);
```

While `assertSame()` compares memory addresses, `assertEquals()` compares contents. For objects, this means the `.equals()` method is used instead of `==`. `assertEquals()` has numerous overloaded implementations that compare objects to other objects, or primitives to other primitives. Regardless of what you are comparing, the expected value is always listed before the actual value you are testing against. Here are a few examples:

```java
// compare two objects (not using the description argument)
assertEquals(expectedBirthDate, son.getBirthDate());
assertEquals("Garrett", son.getMiddleName());

// compare primitives
assertEquals(50, race.getNumLaps());
assertEquals('a', car.getIdentifier());
assertEquals(expectedByte, customer.getCategory());
```

JUnit provides special `assertEquals()` methods comparing doubles and floats. These methods accept a delta argument, allowing for rounding off errors. Here is how you can validate that the temperature is 97.1 degrees, accurate to within 0.001 degrees:

```java
assertEquals("Temperature", expectedTemp, actualTemp, 0.001);
```

Additional examples

Here is how you check for a Boolean condition:

```java
assertTrue("Expected the temperature to be non-negative.", actualTemp >= 0);
```

Prior to JUnit 3.8, you had to adjust your tests slightly to check for false conditions:

```java
assertTrue("The car should not be running.", !car.isRunning());
```

JUnit 3.8 added the `assertFalse()` method, making the test more clear:

```java
assertFalse("The car should not be running.", car.isRunning());
```

Checking for `null` is easy:

```java
assertNull("Did not expect to find an employee.",
           database.getEmployee("someInvalidEmployeeId"));
```

You can also check for non-null values:

```java
assertNotNull("Expected to find an employee with id=" + id,
              database.getEmployee(id));
```

And finally, you can explicitly cause a test failure using the `fail()` method:

```java
fail("Unable to configure database.");
```
See Also
See the JavaDocs for `junit.framework.Assert`. The methods in `Assert` are all static, and can be called from other classes. See Recipe 6.2 for an example.

4.4 Unit Test Granularity

Problem
You want to know how fine-grained your unit tests should be.

Solution
Each unit test should check one specific piece of functionality. Do not combine multiple, unrelated tests into a single `testXXX()` method.

Discussion
Each test method can have as many `assertXXX()` calls as you like, but they can lead to problems:

```java
public void testGame() throws BadGameException {
    Game game = new Game();
    Ship fighter = game.createFighter("001");
    assertEquals("Fighter did not have the correct identifier", 
                  "001", fighter.getId());
    Ship fighter2 = game.createFighter("001");
    assertSame("createFighter with same id should return same object", 
               fighter, fighter2);
    assertFalse("A new game should not be started yet", game.isPlaying());
}
```

This is a bad design because each `assertXXX()` method is testing an unrelated piece of functionality. If the first `assertEquals()` fails, the remainder of the test is not executed. When this happens, you won’t know if the other tests are functional.

Example 4-2 shows a refactored test case that tests various aspects of our game independently. We will see how to remove the duplicated code in Recipe 4.5 when we talk about `setUp()` and `tearDown()`.

Example 4-2. Refactored tests

```java
public void testCreateFighter() throws BadGameException {
    Game game = new Game();
    Ship fighter = game.createFighter("001");
    assertEquals("Fighter did not have the correct identifier", 
                  "001", fighter.getId());
    game.shutdown();
}
```
4.5 Set Up and Tear Down

Problem
You want to avoid duplicated code when several tests share the same initialization and cleanup code.

Solution
Use the setUp() and tearDown() methods. Both of these methods are part of the junit.framework.TestCase class.

Discussion
JUnit follows a very specific sequence of events when invoking tests. First, it constructs a new instance of the test case for each test method. Thus, if you have five test methods, JUnit constructs five instances of your test case. For this reason, instance

Example 4-2. Refactored tests  (continued)

```java
public void testSameFighters() throws BadGameException {
    Game game = new Game();
    Ship fighter = game.createFighter("001");
    Ship fighter2 = game.createFighter("001");
    assertEquals("createFighter with same id should return same object",
                fighter, fighter2);
    game.shutdown();
}

public void testGameInitialState() throws BadGameException {
    Game game = new Game();
    assertFalse("A new game should not be started yet", game.isPlaying());
    game.shutdown();
}
```

With this approach, one test failure will not cause the remaining assertXXX() statements to be skipped.

This issue raises the question: should a test method ever contain more than one assertXXX()? The answer is a definite yes! If you are testing a series of conditions in which subsequent tests will always fail when the first fails, you may as well combine all of the asserts in one test.

See Also
See Recipe 4.5 to learn how to create the Game object in the setUp() method and shutdown the game in the tearDown() method.
variables cannot be used to share state between test methods. After constructing all of the test case objects, JUnit follows these steps for each test method:

- Calls the test case’s setUp() method
- Calls the test method
- Calls the test case’s tearDown() method

This process repeats for each of the test methods in the test case. Example 4-3 shows how you can take advantage of setUp() and tearDown() to avoid duplicated code.

**Example 4-3. setUp() and tearDown()**

```java
package com.oreilly.java.xp.junit;

import com.oreilly.java.xp.common.BadGameException;
import com.oreilly.java.xp.common.Game;
import com.oreilly.java.xp.common.Ship;
import junit.framework.TestCase;

/**
 * Sample unit tests for the {@link Game} class.
 */
public class TestGame extends TestCase {
    private Game game;
    private Ship fighter;

    public void setUp() throws BadGameException {
        this.game = new Game();
        this.fighter = this.game.createFighter("001");
    }

    public void tearDown() {
        this.game.shutdown();
    }

    public void testCreateFighter() {
        assertEquals("Fighter did not have the correct identifier", "001", this.fighter.getId());
    }

    public void testSameFighters() {
        Ship fighter2 = this.game.createFighter("001");
        assertEquals("createFighter with same id should return same object", this.fighter, fighter2);
    }

    public void testGameInitialState() {
        assertTrue("A new game should not be started yet", !this.game.isPlaying());
    }
}
```
You can often ignore the `tearDown()` method because individual unit tests are not long-running processes, and objects are garbage-collected as soon as the JVM exits. `tearDown()` can be useful, however, if your tests do things like open database connections, show GUI frames, or consume other sorts of system resources that you would like to clean up immediately. If you are running a large suite of unit tests, setting references to `null` in your `tearDown()` methods may help the garbage collector reclaim memory as other tests run.

You may be wondering why you should write a `setUp()` method instead of simply initializing fields in a test case’s constructor. After all, since a new instance of the test case is created for each of its test methods, the constructor is always called before `setUp()`. In a vast majority of cases, you can use the constructor instead of `setUp()` without any side effects.

In cases where your test case is part of a deeper inheritance hierarchy, you may wish to postpone object initialization until instances of derived classes are fully constructed. This is a good technical reason why you might want to use `setUp()` instead of a constructor for initialization. Using `setUp()` and `tearDown()` is also good for documentation purposes, simply because it may make the code easier to read.

**See Also**

Recipe 4.6 shows how to set up data once for a whole series of tests.

### 4.6 One-Time Set Up and Tear Down

**Problem**

You want to run some setup code one time and then run several tests. You only want to run your cleanup code after all of the tests are finished.

**Solution**

Use the `junit.extensions.TestSetup` class.

**Discussion**

As outlined in Recipe 4.5, JUnit calls `setUp()` before each test, and `tearDown()` after each test. In some cases you might want to call a special setup method once before a series of tests, and then call a teardown method once after all tests are complete. The `junit.extensions.TestSetup` class supports this requirement. Example 4-4 shows how to use this technique.
TestSetup is a subclass of junit.extensions.TestDecorator, which is a base class for defining custom tests. The main reason for extending TestDecorator is to gain the ability to execute code before or after a test is run. The setUp() and tearDown() methods of TestSetup are called before and after whatever Test is passed to its constructor. In our example we pass a TestSuite to the TestSetup constructor:

```java
TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {
    protected void setUp() throws Exception {
        // do your one-time setup here!
    }

    protected void tearDown() throws Exception {
        // do your one-time tear down here!
    }
};
return setup;
}
```

This means that TestSetup’s setUp() method is called once before the entire suite, and tearDown() is called once afterwards. It is important to note that the setUp() and tearDown() methods within TestPerson are still executed before and after each individual unit test method within TestPerson.

**See Also**

Recipe 4.5 describes setUp() and tearDown().

---

* JUnit includes source code. Check out the code for TestSetup to learn how to create your own extension of TestDecorator.
4.7 Organizing Tests into Test Suites

Problem
You want to organize multiple tests into a suite of tests, all of which run at once.

Solution
JUnit does this automatically for each test case. You can construct an instance of
junit.framework.TestSuite to create your own suites manually.

Discussion
When you use the text or graphical test runner, JUnit looks for the following method
in your test case:

```java
public static Test suite() { ... }
```

If the method is not found, JUnit uses reflection to automatically locate all testXXX() methods
in your test case, adding them to a suite of tests. It then runs all tests in this
suite. You can duplicate the default suite() behavior as follows:

```java
public class TestGame extends TestCase {
    ...
    public static Test suite() {
        return new TestSuite(TestGame.class);
    }
}
```

By passing the TestGame.class object to the TestSuite constructor, you are telling
JUnit to locate all of the testXXX() methods in that class and add them to the suite.
This code does not do anything above and beyond what JUnit does automatically,
but there are more interesting ways to use the TestSuite class. For instance, you can
add individual tests to only run certain tests, or you can control the order in which
they are executed:

```java
public static Test suite() {
    TestSuite suite = new TestSuite();
    // To use this idiom, you must define the String constructor in your
    // TestGame class. Remember that JUnit 3.8 made that constructor optional.
    suite.addTest(new TestGame("testCreateFighter"));
    suite.addTest(new TestGame("testSameFighters"));
    return suite;
}
```

* The Ant junit task also looks for the suite() method.
Or, even better, you can compose multiple suites into other suites. You might recognize this as the Composite design pattern.* For example:

```java
public static Test suite() {
    TestSuite suite = new TestSuite(TestGame.class);
    suite.addTest(new TestSuite(TestPerson.class));
    return suite;
}
```

Now, when you run this test case, you will run all tests from both `TestGame` and `TestPerson`.

See Also
Recipe 4.11 provides suggestions for organizing test suites.

4.8 Running a Test Class Directly

Problem
You don’t want to invoke one of the JUnit test runners. Instead, you would like to run your test directly.

Solution
Add a `main()` method to your test case. Or, use Ant to run your tests as discussed in Chapter 3.

Discussion
Adding a `main()` method can make a class easier to run. Most IDEs allow you to click on a class and select some sort of “run” option from a popup menu, provided the class has a `main()` method. Here is a sample `main()` method for our `TestGame` class:

```java
public class TestGame extends TestCase {
    ...
    public static void main(String[] args) {
        junit.textui.TestRunner.run(new TestSuite(TestGame.class));
    }
}
```

When executed, this method runs the test suite in text mode. Output is sent to the console.

* See Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley).
See Also
Recipe 4.2 shows how to run unit tests. Chapter 3 shows how to run tests using Ant.

4.9 Repeating Tests

Problem
You want to run certain tests repeatedly.

Solution
Use the `junit.extensions.RepeatedTest` class.

Discussion
You may want to run certain tests repeatedly to measure performance or to diagnose intermittent problems. The `RepeatedTest` class makes this easy:

```java
public static Test suite() {
    // run the entire test suite ten times
    return new RepeatedTest(new TestSuite(TestGame.class), 10);
}
```

`RepeatedTest`'s first argument is another `Test` to run; the second argument is the number of iterations. Since `TestSuite` implements the `Test` interface, we can repeat the entire test as just shown. Here is how you can build a test suite where different tests are repeated differently:

```java
TestSuite suite = new TestSuite();
// repeat the testCreateFighter test 100 times
suite.addTest(new RepeatedTest(new TestGame("testCreateFighter"), 100));
// run testSameFighters once
suite.addTest(new TestGame("testSameFighters"));
// repeat the testGameInitialState test 20 times
suite.addTest(new RepeatedTest(new TestGame("testGameInitialState"), 20));
```

See Also
Recipe 4.13 shows more examples of `RepeatedTest`.

* Threading bugs are often intermittent.
4.10 Test Naming Conventions

Problem
You want to define a naming convention for your tests.

Solution
Prefix each test case classname with a consistent word, such as “Test” or “Unit-Test”. Put test cases in the same directory as the classes they are testing.

Discussion
Consistent naming conventions serve two purposes. First, they make your code more maintainable. Second, consistency facilitates automation. The tests in this chapter are prefixed with “Test”, resulting in names like TestGame, TestPerson, and TestAccount. These correspond to the Game, Person, and Account classes, respectively.

Writing one test case per class makes it very easy to glance at a directory and see which tests are available. Putting “Test” at the beginning of the filenames makes sorting easier, in our opinion, particularly when your IDE provides some sort of jump-to functionality. All tests will be grouped together and easy to identify. On the other hand, putting “Test” at the end of the filenames does make it easier to identify which classes do not have tests.

Another popular convention is to place all test classes in a parallel directory structure. This allows you to use the same Java package names for your tests, while keeping the source files separate. To be honest, we do not like this approach because you must look in two different directories to find files. Ant can easily exclude tests from your build even if they reside in the same source directory.

Finally, a consistent naming convention makes it easier to locate tests when using Ant for your builds. You might want to exclude all of your tests when you create a production build of your software. You can do this in Ant as follows:

```xml
<javac destdir="${dir.build}"/>
<src path="${dir.src}"/>
<!-- see how easy it is to exclude the tests from a build! -->
<exclude name="**/Test*.java"/>
</javac>
```

* IntelliJ IDEA (http://www.intellij.com) allows you to hit Ctrl-N and then begin typing a classname. If you follow the prefix convention, you immediately see a list of all tests as soon as you type Test.
4.11 Unit Test Organization

Problem
You want to organize all of your tests consistently.

Solution
Create a test case that runs all tests in the current package and subpackages. Duplicate this pattern for all packages in your application.

Discussion
Example 4-5 shows an example of the technique outlined in the solution just presented. It runs all of the test suites in the current package as well as delegating to each AllTests class in immediate subpackages.

Example 4-5. AllTests example
package com.oreilly.javap.junit;

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

Prefix or Postfix?

We prefer to prefix “Test” onto the beginning of our test classnames, while many other people prefer to postfix “Test” onto the end. The main argument for postfixing names is that Customer.java and CustomerTest.java appear next to each other when sorting files within a directory. This makes it easy to identify which classes have tests.

We use the prefix convention because it makes searching easier within our IDE. We recommend you use the approach you prefer consistently. Remember that Ant relies on a consistent filenames convention to locate tests.

See Also
Recipe 3.15 shows how to run tests using Ant based on naming conventions. Recipe 3.13 shows how to exclude test classes from a build.
Example 4.5. AllTests example (continued)

```java
/**
 * Runs all test suites in the current package and sub-packages.
 */
public class AllTests extends TestCase {

    /**
     * @return a suite containing all tests in this package
     *         and subpackages.
     */
    public static Test suite() {
        TestSuite suite = new TestSuite();

        // add tests from the current directory. This requires manual
        // updates, which is the main weakness of this technique
        suite.addTest(new TestSuite(TestGame.class));
        suite.addTest(new TestSuite(TestPerson.class));

        // add AllTests from any sub-packages
        suite.addTest(com.oreilly.javasp.junit.sub.AllTests.suite());
        // suite.addTest(...) // continue for other sub-packages

        return suite;
    }
}
```

This technique can be useful when using an IDE because you can select any AllTests class and run tests for a subset of your project. Assuming that you follow this pattern consistently, you can run the AllTests in your root directory to run every test in your application.

AllTests intentionally avoids the TestXXX naming convention outlined in Recipe 4.10. This prevents the AllTests from being executed when you tell Ant to find and run all TestXXX classes.

Human fallibility is the main weakness of this technique. If you are not diligent, you will forget to add some tests to one of the AllTests classes. This can be overcome by writing a utility to automatically generate the AllTests classes. Yet another technique is to do the same thing dynamically: to write a class that sifts through a directory/package looking for TestXXX classes and including them in the suite.

You might also want to consider whether the AllTests classes should run tests in subpackages, or just the current package. Here is a modification that allows you to choose the behavior you want based on a system property:

```java
public static Test suite() {
    TestSuite suite = new TestSuite();
```

* IntelliJ IDEA allows you to right-click on any directory and run all tests in that package, thus eliminating the need to manually create an AllTests class.
// add tests from the current directory
suite.addTest(new TestSuite(TestGame.class));
suite.addTest(new TestSuite(TestPerson.class));

// only test subdirectories if a system property is true
if ("true".equals(System.getProperty("test.subdirs"))) {
    // add AllTests from any sub-packages
    suite.addTest(com.oreilly.javaxp.junit.sub.AllTests.suite());
    // suite.addTest(...) // continue for other sub-packages
}

return suite;
}

See Also
Recipe 3.15 shows an example of Ant’s batchtest element.

4.12 Exception Handling

Problem
You want to test for exceptions.

Solution
Use a try/catch block to catch the expected exception. Call the fail() method if the exception does not occur.

Discussion
In the following example, the Person constructor should throw an IllegalArgumentException if both of its arguments are null. The test fails if it does not throw this exception.

    public void testPassNullsToConstructor() {
        try {
            Person p = new Person(null, null);
            fail("Expected IllegalArgumentException when both args are null");
        } catch (IllegalArgumentException expected) {
            // ignore this because it means the test passed!
        }
    }

Only use this technique when you are expecting an exception. For other error conditions, let the exception propagate to JUnit. It will catch the exception and report a test error. Here is something you do not want to do:

    // don't do this!
    public void testBadStyle() {
        // don't do this!
    }
try {
    SomeClass c = new SomeClass();
    c.doSomething();
    ...
} catch (IOException ioe) {
    fail("Caught an IOException");
} catch (NullPointerException npe) {
    fail("Caught a NullPointerException");
}

The main problem is that JUnit already catches unhandled errors, so you are doing unnecessary work. The extra try/catch code adds complexity to your tests, making them harder to maintain. The previous example is much simpler when written like this:

```
// must declare IOException because it is not a RuntimeException
public void testGoodStyle() throws IOException {
    SomeClass c = new SomeClass();
    c.doSomething();
    ...
}
```

### 4.13 Running Tests Concurrently

**Problem**

You want to run several tests concurrently using threads.

**Solution**

Use the `junit.extensions.ActiveTestSuite` class to build a suite of tests that run concurrently.

**Discussion**

The `ActiveTestSuite` class runs each of its tests in a separate thread. The suite does not finish until all of the test threads are complete. Example 4-6 shows how to run three different test methods in three different threads.

**Example 4-6. Running tests in different threads**

```java
public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new TestGame("testCreateFighter"));
    suite.addTest(new TestGame("testGameInitialState"));
    suite.addTest(new TestGame("testSameFighters"));
    return suite;
}
```
While you probably won’t use this technique often, running tests in threads can serve as a rudimentary stress tester. You might also use ActiveTestSuite to help identify threading problems in your code. By combining ActiveTestSuite with RepeatedTest, you can uncover threading problems that only show up intermittently.

Example 4-7 shows how you can combine repeated tests and other test suites into an ActiveTestSuite. Each of the repeated tests runs in a different thread; therefore, you end up with four threads. If you are experiencing occasional threading glitches, you might want to increase the number of iterations and run a similar test suite overnight.

Example 4-7. A more advanced test suite

```java
class TestGame {
    public boolean testCreateFighter()
    {...
    }
}

class TestPerson {
    public static Test suite() {
        TestSuite suite = new ActiveTestSuite();

        // run one test in a thread
        suite.addTest(new TestGame("testCreateFighter");

        // run this test 100 times in a second thread
        suite.addTest(new RepeatedTest(
            new TestGame("testGameInitialState"), 100));

        // run this test 200 times in a third thread
        suite.addTest(new RepeatedTest(
            new TestGame("testSameFighters"), 200));

        // run some other test suite in a fourth thread
        suite.addTest(TestPerson.suite());
        return suite;
    }
}
```

See Also
Recipe 4.9 explains the RepeatedTest class.

### 4.14 Testing Asynchronous Methods

**Problem**
You want to test asynchronous methods.

**Solution**
Use a mock listener to wait for the asynchronous method to complete.
Discussion

An asynchronous method executes in its own thread, notifying some listener when it is complete. Code that calls an asynchronous method does not block, meaning that you cannot write a test like this:

```java
public void testSomething() {
    someAsynchronousMethod();
    assertXXX(...);
}
```

The problem with this code lies in the fact that the `assertXXX()` is almost certainly executed before the thread started by `someAsynchronousMethod()` has a chance to do its work. We really need to do something like this:

1. Call an asynchronous method.
2. Wait until the method is complete.
3. Get the results.
   - If the method times out, fail.
   - Otherwise, check the results.

To illustrate, let’s look at a simple interface for searching. We assume that searching occurs in its own thread, notifying a `SearchModelListener` whenever the search is complete. Example 4-8 shows the API.

**Example 4-8. SearchModel interface**

```java
public interface SearchModel {
    void search(Object searchCriteria, SearchModelListener listener);
}
```

The `search()` method is asynchronous, notifying the `SearchModelListener` when it is complete. Example 4-9 shows the code for the `SearchModelListener` interface.

**Example 4-9. SearchModelListener interface**

```java
public interface SearchModelListener extends EventListener {
    void searchFinished(SearchModelEvent evt);
}
```

In order to test the search model, we must write a mock listener that waits for the search to complete. Once the mock listener receives its result, we can verify that the data is correct. Example 4-10 shows the code for a mock listener.

**Example 4-10. MockSearchModelListener class**

```java
class MockSearchModelListener implements SearchModelListener {
    private SearchModelEvent evt;

    public void searchFinished(SearchModelEvent evt) {
        this.evt = evt;
    }
}
```
The key to our mock listener is the synchronized block. This listener assumes that some other thread (our unit test) is waiting for the search to complete. By calling notifyAll(), the mock listener allows the unit test to “wake up” and continue. Example 4-11 shows the unit test, which ties everything together.

```java
Example 4-10. MockSearchModelListener class (continued)

synchronized (this) {
    notifyAll();
}

public SearchModelEvent getSearchModelEvent() {
    return this.evt;
}
```

The key to our mock listener is the synchronized block. This listener assumes that some other thread (our unit test) is waiting for the search to complete. By calling notifyAll(), the mock listener allows the unit test to “wake up” and continue. Example 4-11 shows the unit test, which ties everything together.

```java
Example 4-11. Asynchronous unit test

public void testAsynchronousSearch() throws InterruptedException {
    MockSearchModelListener mockListener = new MockSearchModelListener();
    SearchModel sm = new PersonSearchModel();
    // 1. Execute the search
    sm.search("eric", mockListener);

    // 2. Wait for the search to complete
    synchronized (mockListener) {
        mockListener.wait(2000);
    }

    // 3. Get the results
    SearchModelEvent evt = mockListener.getSearchModelEvent();

    // 3a) if the method times out, fail
    assertNotNull("Search timed out", evt);

    // 3b) otherwise, check the results
    List results = evt.getSearchResult();
    assertEquals("Number of results", 1, results.size());
    Person p = (Person) results.get(0);
    assertEquals("Result", "Eric", p.getFirstName());
}
```

The unit test first creates a mock listener, passing that listener to the search model. It then uses a synchronized block to wait until the listener calls notifyAll(). Calling wait(2000) indicates that the test will wait for at least two seconds before it stops waiting and continues. If this happens, the mock listener’s event object is null because it was never notified by the search model.

* notifyAll() can only be called within synchronized code.
Having a timeout period is critical; otherwise, your test will wait indefinitely if the asynchronous method fails and never notifies the caller.

Assuming the search completed within two seconds, the test goes on to check the results for correctness.

See Also
Mock objects are described in Chapter 6.

4.15 Writing a Base Class for Your Tests

Problem
You want to reuse the same behavior in all of your tests without duplicating code.

Solution
Define common behavior in a subclass of `junit.framework.TestCase` and extend from your class, rather than directly extending `TestCase`.

Discussion
JUnit does not require that your tests directly extend `TestCase`. Instead, you can introduce new `TestCase` extensions for common behavior. You might want to ensure that some common initialization code is always executed before each of your tests. In that case, you might write something like this:

```java
public abstract class MyAbstractTestCase extends TestCase {
    public MyAbstractTestCase() {
        initializeApplicationProperties();
    }

    public MyAbstractTestCase(String testName) {
        super(testName);
        initializeApplicationProperties();
    }

    // initialize some custom application framework. Leave this method
    // protected so subclasses can customize.
    protected void initializeApplicationProperties() {
        MyFramework.initialize("common/myappconfig.properties");
    }
}
```
Tests in your application can now extend `MyAbstractTestCase` and your framework initialization code will always be executed before the tests run.

Providing convenience methods is another reason why you might want to extend `TestCase`. We show this in the next recipe when we define a method to retrieve a Swing `JFrame` for graphical testing.

### 4.16 Testing Swing Code

**Problem**

You want to write unit tests for Swing portions of your application.

**Solution**

Keep application logic separate from GUI layout, thus minimizing the need to test graphical code directly. Also, design your user interface in terms of discrete components that are testable without complex setup and configuration.

**Discussion**

Graphical code presents many testing challenges. For instance, many Swing functions only work when the components are visible on screen. In these cases, your tests have to create dummy frames and show the components before the tests can succeed. In other cases, Swing schedules events on the AWT event queue rather than updating component states immediately. We show how to tackle this issue in the next recipe.

Ideally, you should strive to minimize the need to test Swing code in the first place. Application logic, such as computing the monthly payment amount for a loan, should not be intertwined with the `JTable` that displays the payment history. Instead, you might want to define three separate classes:

- **Loan**
  A utility class that keeps track of payments, interest rates, and other attributes. This class can be tested independently of Swing.

- **LoanPaymentTableModel**
  A Swing table model for a history of loan payments. Because table models are nongraphical, you can test them just like any other Java class.

- **JTable**
  Displays the `LoanPaymentTableModel`. Because `JTable` is provided with Swing, you don’t have to test it.

There are more complex scenarios where you cannot avoid Swing testing. Let’s suppose you need a panel to display information about a person and would like to test
it. The Person class is easily testable on its own, and probably contains methods to retrieve a name, address, SSN, and other key pieces of information. But the PersonEditorPanel is graphical and a little more challenging to test. You might start with the code shown in Example 4-12.

**Example 4-12. First draft of PersonEditorPanel.java**

```java
public class PersonEditorPanel extends JPanel {
    private JTextField firstNameField = new JTextField(20);
    private JTextField lastNameField = new JTextField(20);
    // @todo – add more fields later
    private Person person;

    public PersonEditorPanel() {
        layoutGui();
        updateDataDisplay();
    }

    public void setPerson(Person p) {
        this.person = p;
        updateDataDisplay();
    }

    public Person getPerson() {
        // @todo – update the person with new information from the fields
        return this.person;
    }

    private void layoutGui() {
        // @todo – define the layout
    }

    private void updateDataDisplay() {
        // @todo – ensure the fields are properly enabled, also set data on the fields.
    }
}
```

Our PersonEditorPanel does not function yet, but it is far enough along to begin writing unit tests. Before delving into the actual tests, let’s look at a base class for Swing tests. Example 4-13 shows a class that provides access to a JFrame for testing purposes. Our unit test for PersonEditorPanel will extend from SwingTestCase.

**Example 4-13. SwingTestCase.java**

```java
package com.oreillyjavaxp.junit;
import junit.framework.TestCase;
import javax.swing.JFrame;
import java.lang.reflect.InvocationTargetException;
```
SwingTestCase provides access to a JFrame and takes care of disposing the frame in its tearDown() method. As you write more Swing tests, you can place additional functionality in SwingTestCase.

Example 4-14 shows the first few tests for PersonEditorPanel. In these tests, we check to see if the fields in the panel are enabled and disabled properly.

Example 4-14. The first PersonEditorPanel tests

```java
public class TestPersonEditorPanel extends SwingTestCase {
    private PersonEditorPanel emptyPanel;
    private PersonEditorPanel tannerPanel;
    private Person tanner;

    protected void setUp() throws Exception {
        // create a panel without a Person
        this.emptyPanel = new PersonEditorPanel();

        // create a panel with a Person
        this.tanner = new Person("Tanner", "Burke");
        this.tannerPanel = new PersonEditorPanel();
        this.tannerPanel.setPerson(this.tanner);
    }

    public void testTextFieldsAreInitiallyDisabled() {
        assertTrue("First name field should be disabled", 
                !this.emptyPanel.getFirstNameField().isEnabled());
        assertTrue("Last name field should be disabled", 
                !this.emptyPanel.getLastNameField().isEnabled());
    }

    public void testEnabledStateAfterSettingPerson() {
        assertTrue("First name field should be enabled", 
                this.tannerPanel.getFirstNameField().isEnabled());
    }
}
```
You might notice that our tests have to get to the first and last name fields, so we need to introduce the `getFirstNameField()` and `getLastNameField()` methods in our panel:

```java
JTextField getFirstNameField() {
    return this.firstNameField;
}

JTextField getLastNameField() {
    return this.lastNameField;
}
```

These methods are package-scope because we only need them for testing purposes. When you first run the unit tests, they will fail because we did not write any logic to enable and disable the fields. This method can be added to `PersonEditorPanel` in order to make the tests pass:

```java
private void updateEnabledStates() {
    this.firstNameField.setEnabled(person != null);
    this.lastNameField.setEnabled(person != null);
}
```

Once you get these tests working, you can test for the actual values of the two fields:

```java
public void testFirstName() {
    assertEquals("First name", ",
        this.emptyPanel.getFirstNameField().getText());
    assertEquals("First name", this.tanner.getFirstName(),
        this.tannerPanel.getFirstNameField().getText());
}

public void testLastName() {
    assertEquals("Last name", ",
        this.emptyPanel.getLastNameField().getText());
    assertEquals("Last name", this.tanner.getLastName(),
        this.tannerPanel.getLastNameField().getText());
}
```

These will also fail until you add some more logic to `PersonEditorPanel` to set data on the two text fields:

```java
private void updateDataDisplay() {
    if (this.person == null) {
        this.firstNameField.setText(" ");
        this.lastNameField.setText(" ");
    } else {
        this.firstNameField.setText(this.person.getFirstName());
        this.lastNameField.setText(this.person.getLastName());
    }
    updateEnabledStates();
}
```
When complete, your tests should confirm that you can create an empty panel, set a person object on it, and retrieve person object after it has been edited. You should also write tests for unusual conditions, such as a null person reference or null data within the person. This is a data-oriented test, ensuring that the panel properly displays and updates its data. We did not try to verify the graphical positioning of the actual components, nor have we tried to test user interaction with the GUI.

**See Also**

Recipe 4.18 discusses problems with `java.awt.Robot`. Chapter 11 provides some references to Swing-specific testing tools. Recipe 11.5 discusses some pros and cons of making methods package-scope for the sole purpose of testing them.

### 4.17 Avoiding Swing Threading Problems

**Problem**

You want to test Swing functions that dispatch to the AWT event queue, such as focus traversal.

**Solution**

Write a utility method to wait until pending AWT event queue messages are processed.

**Discussion**

Suppose you want to test the focus traversal order of the `PersonEditorPanel` introduced in the previous recipe. You want to ensure that focus travels from component to component in the correct order as the user hits the tab key. To do this, you write the following test:

```java
public void testTabOrder() {
    JTextField firstNameField = this.tannerPanel.getFirstNameField();

    firstNameField.requestFocusInWindow();
    firstNameField.requestFocusInWindow();

    // simulate the user hitting tab
    firstNameField.transferFocus();
    firstNameField.transferFocus();

    // ensure that the last name field now has focus
    JTextField lastNameField = this.tannerPanel.getLastNameField();
    assertTrue("Expected last name field to have focus",
               !lastNameField.hasFocus());
}
```
As written, this test fails. First and foremost, the components must be visible on screen before they can obtain focus. So you try modifying your setUp() method as follows:

```java
protected void setUp() throws Exception {
    this.emptyPanel = new PersonEditorPanel();
    this.tanner = new Person("Tanner", "Burke");
    this.tannerPanel = new PersonEditorPanel();
    this.tannerPanel.setPerson(this.tanner);
    getTestFrame().getContentPane().add(this.tannerPanel,
            BorderLayout.CENTER);
    getTestFrame().pack();
    getTestFrame().show();
}
```

This takes advantage of the JFrame provided by our base class, SwingTestCase. When you run your test again, it still fails! The initial focus never made it to the first name field. Here is a partial solution:

```java
public void testTabOrder() {
    JTextField firstNameField = this.tannerPanel.getFirstNameField();
    // make sure the first name field has focus
    while (!firstNameField.hasFocus()) {
        getTestFrame().toFront();
        firstNameField.requestFocusInWindow();
    }
    // simulate the user hitting tab
    firstNameField.transferFocus();
    // ensure that the last name field now has focus
    JTextField lastNameField = this.tannerPanel.getLastNameField();
    assertTrue("Expected last name field to have focus",
            lastNameField.hasFocus());
}
```

This approach keeps trying until the first name field eventually gains focus. It also brings the test frame to the front of other windows because, during testing, we found that the frame sometimes gets buried if the user clicks on any other window while the test is running. We discovered this by repeating our tests and clicking on other applications while the tests ran:

```java
public static Test suite() {
    return new RepeatedTest(
            new TestSuite(TestPersonEditorPanel.class), 1000);
}
```

We still have one more problem. When the test runs repeatedly, you will notice that the test fails intermittently. This is because the transferFocus() method does not occur immediately. Instead, the request to transfer focus is scheduled on the AWT
event queue. In order to pass consistently, the test must wait until the event has a chance to be processed by the queue. Example 4-15 lists the final version of our test.

**Example 4-15. Final tab order test**

```java
public void testTabOrder() {
    JTextField firstNameField = this.tannerPanel.getFirstNameField();

    // make sure the first name field has focus
    while (!firstNameField.hasFocus()) {  
        getTestFrame().toFront();
        firstNameField.requestFocusInWindow();
    }

    // simulate the user hitting tab
    firstNameField.transferFocus();

    // wait until the transferFocus() method is processed
    waitForSwing();

    // ensure that the last name field now has focus
    JTextField lastNameField = this.tannerPanel.getLastNameField();
    assertTrue("Expected last name field to have focus",  
               lastNameField.hasFocus());
}
```

The `waitForSwing()` method is a new feature of our base class, `SwingTestCase`, that blocks until pending AWT events like `transferFocus()` are processed:

```java
public void waitForSwing()  
    if (!SwingUtilities.isEventDispatchThread()) {  
        try {  
            SwingUtilities.invokeAndWait(new Runnable() {  
                public void run() {  
                }  
            });  
        } catch (InterruptedException e) {  
            } catch (InvocationTargetException e) {  
                }
    }
```

Now, our test runs almost 100% of the time. When repeating the test thousands of times, you can make the test fail every once in a while by randomly clicking on other applications. This is because you take away focus just before the test asks the last name field if it has focus. This sort of problem is unavoidable, and illustrates one of the reasons why you should minimize your dependency on Swing tests.

The most valuable lesson of this recipe is the technique of repeating your graphical tests many thousands of times until all of the quirky Swing threading issues are resolved. Once your tests run as consistently as possible, remove the repeated test.
Also, while your tests are running, avoid clicking on other running applications so you don’t interfere with focus events.

See Also
Chapter 11 provides some references to Swing-specific testing tools.

4.18 Testing with the Robot

Problem
You want to simulate the user clicking on the mouse or typing with the keyboard using `java.awt.Robot`.

Solution
We do not recommend this technique.

Discussion
`java.awt.Robot` allows Java applications to take command of native system input events, such as moving the mouse pointer or simulating keystrokes. At first glance, this seems to be a great way to test your GUIs. Your tests can do exactly what the user might do and then verify that your components are displaying the correct information.

We have found that this approach is dangerous.

- Robot tests are very fragile, breaking any time the GUI layout changes.
- If the user moves the mouse while the tests are running, the Robot continues clicking, sometimes on the wrong application.*
- Since the tests run so quickly, it can be impossible to stop the tests once the Robot gets confused and starts clicking on other apps and typing characters that show up in other windows.

If you really feel that you could use some Robot tests, consider naming them differently than other tests. You might have a collection of `RobotTest*.java` tests. You can then run them independently of other tests, if you are extremely careful to avoid touching the mouse while the tests run.

* One programmer reported that the Robot sent a partially completed email because it clicked on the send button of the mail client instead of a button in the application being tested.
See Also
Chapter 11 provides some references to Swing-specific testing tools.

4.19 Testing Database Logic

Problem
You want to test database logic using JUnit.

Solution
Write scripts to generate a stable testing database and test against that data.

Discussion
Testing against a database is challenging in many organizations because you have to define predictable data. The only truly reliable approach is to create the test data in a private database automatically for each set of tests. When the tests are finished, you should destroy the test data. If you create the test database manually, you run the risk of corruption over time as people modify data that your tests assume is present.

For very large databases, you may have to settle for either creating clean test data daily or weekly, or loading a subset of the database with well-known testing records.

We recommend that you follow the technique outlined in Recipe 4.6 to perform one-time setup and tear down before and after a group of tests. You can create the test data in the one-time setup, and remove it in the one-time tear down. Once you have control of the data, you can test against that data:

```java
public void testDeleteEmployee() throws SQLException {
    EmployeeDAO dao = new EmployeeDAO();
    assertNotNull("Employee 'ericBurke' should be present",
                dao.getEmployee("ericBurke"));
    dao.deleteEmployee("ericBurke");
    assertNull("Employee 'ericBurke' should not be present",
                dao.getEmployee("ericBurke"));
}
```

* This is a political battle in many companies, because the database administrators might not give programmers the permission to create new tables or perform other functions necessary to create test data.
Another challenge is the fact that early tests might modify data in ways that interfere with later tests. In these cases, you can either write functions to clean up data after the earlier tests run, or you can build your test suites manually. By building the suites manually, you can control the order in which the tests run:

```java
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestEmployeeDB("testCreateEmployee"));
    suite.addTest(new TestEmployeeDB("testUpdateEmployee"));
    suite.addTest(new TestEmployeeDB("testDeleteEmployee"));
    return suite;
}
```

Database-specific copy, backup, and restore mechanisms are sometimes tremendously faster than reinitializing the database with a series of SQL statements. For example, if your database is MS SQL Server, you can copy over a known testing database .mdf/.ldf file to get your database to a known state very quickly.

**See Also**

Recipe 4.6 shows how to implement oneTimeSetUp() and oneTimeTearDown().

### 4.20 Repeatedly Testing the Same Method

**Problem**

You want to test a method with a wide range of input data. You are not sure if you should write a different test for each combination of input data, or one huge test that checks every possible combination.

**Solution**

Write a `suite()` method that iterates through all of your input data, creating a unique instance of your test case for each unique input. The data is passed to the test cases through the constructor, which stores the data in instance fields so it is available to the test methods.

**Discussion**

You often want to test some piece of functionality with many different combinations of input data. Your first impulse might be to write a different test method for each possible combination of data; however, this is tedious and results in a lot of mundane coding. A second option is to write a single, big test method that checks every possible combination of input data. For example:
public void testSomething() {
    Foo foo = new Foo();
    // test every possible combination of input data
    assertTrue(foo.doSomething(false, false, false);
    assertFalse(foo.doSomething(false, false, true);
    assertFalse(foo.doSomething(false, true, false);
    assertTrue(foo.doSomething(false, true, true);
    ...etc
}

This approach suffers from a fatal flaw. The problem is that the test stops executing as soon as the first assertion fails, so you won’t see all of the errors at once. Ideally, you want to easily set up a large number of test cases and run them all as independent tests. One failure should not prevent the remaining tests from running.

To illustrate this technique, Example 4-16 contains a utility for determining the background color of a component. The getFieldBackground() method calculates a different background color based on numerous parameters—for example, whether a record is available and whether the current data is valid.

Example 4-16. UtilComponent

public class UtilComponent {
    public static final int ADD_MODE = 1;
    public static final int RECORD_AVAILABLE_MODE = 2;
    public static final int NO_RECORD_AVAILABLE_MODE = 3;

    public static final int KEY_FIELD = 100;
    public static final int REQUIRED_FIELD = 101;
    public static final int READONLY_FIELD = 102;
    public static final int NORMAL_FIELD = 103;

    public static final Color FIELD_NORMAL_BACKGROUND = Color.WHITE;
    public static final Color FIELD_ERROR_BACKGROUND = Color.PINK;
    public static final Color FIELD_REQUIRED_BACKGROUND = Color.CYAN;
    public static final Color FIELD_DISABLED_BACKGROUND = Color.GRAY;

    public static Color getFieldBackground(
        int screenMode,
        int fieldType,
        boolean valid,
        boolean requiredConditionMet) {
        if (fieldType == READONLY_FIELD
            || screenMode == NO_RECORD_AVAILABLE_MODE
            || (fieldType == KEY_FIELD && screenMode != ADD_MODE)) {
            return FIELD_DISABLED_BACKGROUND;
        } else if (!valid) {
            return FIELD_ERROR_BACKGROUND;
        } else if ((fieldType == KEY_FIELD || fieldType == REQUIRED_FIELD) && !requiredConditionMet) {
            ...
There are 48 possible combinations of inputs to the getFieldBackground() method, and 4 possible return values. The test case defines a helper class that encapsulates one combination of inputs along with an expected result. It then builds an array of 48 instances of this class, 1 per combination of input data. Example 4-17 shows this portion of our test.

Example 4-17. Defining the test data

```java
public class TestUtilComponent extends TestCase {

    private int testNumber;

    static class TestData {
        int screenMode;
        int fieldType;
        boolean valid;
        boolean requiredConditionMet;
        Color expectedColor;

        public TestData(int screenMode, int fieldType, boolean valid, boolean requiredConditionMet, Color expectedColor) {
            this.screenMode = screenMode;
            this.fieldType = fieldType;
            this.valid = valid;
            this.requiredConditionMet = requiredConditionMet;
            this.expectedColor = expectedColor;
        }
    }

    public void testUtilComponent() {
        // Test data setup...
    }

    // Test cases...
}
```
Example 4-17. Defining the test data (continued)

private static final TestData[] TESTS = new TestData[] {
    new TestData(UtilComponent.ADD_MODE, 0,
        UtilComponent.KEY_FIELD, false, false, UtilComponent.FIELD_FLAGS_ERROR_BACKGROUND),
    new TestData(UtilComponent.ADD_MODE, 1,
        UtilComponent.KEY_FIELD, false, true, UtilComponent.FIELD_FLAGS_ERROR_BACKGROUND),
    new TestData(UtilComponent.ADD_MODE, 2,
        UtilComponent.KEY_FIELD, true, false, UtilComponent.FIELD_FLAGS_REQUIRED_BACKGROUND),
    ...continue defining TestData for every possible input

The test extends from the normal JUnit TestCase base class, and defines a single private field called testNumber. This field keeps track of which instance of TestData to test. Remember that for each unit test, a new instance of TestUtilComponent is created. Thus, each instance has its own copy of the testNumber field, which contains an index into the TESTS array.

The TESTS array contains every possible combination of TestData. As you can see, we include comments containing the index in the array:

    new TestData(UtilComponent.ADD_MODE, 0)

This index allows us to track down which test cases are not working when we encounter failures. Example 4-18 shows the remainder of our test case, illustrating how the tests are executed.

Example 4-18. Remainder of TestUtilComponent

    public TestUtilComponent(String testMethodName, int testNumber) {
        super(testMethodName);
        this.testNumber = testNumber;
    }

    public void testFieldBackgroundColor() {
        TestData td = TESTS[this.testNumber];

        Color actualColor = UtilComponent.getFieldBackground(td.screenMode,
            td.fieldType, td.valid, td.requiredConditionMet);

        assertEquals("Test number " + this.testNumber + ": ",
            UtilComponent.colorToString(td.expectedColor),
            UtilComponent.colorToString(actualColor));
    }
Our constructor does not follow the usual JUnit pattern. In addition to the test method name, we accept the test number. This is assigned to the testNumber field, and indicates which data to test.

The testFieldBackgroundColor() method is our actual unit test. It uses the correct TestData object to run UtilComponent.getFieldBackground(), using assertEquals() to check the color. We also use UtilComponent to convert the color to a text string before doing the comparison. Although this is not required, it results in much more readable error messages when the test fails.

The final portion of our test is the suite() method. JUnit uses reflection to search for this method. If found, JUnit runs the tests returned from suite() rather than using reflection to locate testXXX() methods. In our case, we loop through our array of test data, creating a new instance of TestUtilComponent for each entry. Each test instance has a different test number, and is added to the TestSuite. This is how we create 48 different tests from our array of test data.

Although we have a hardcoded array of test data, there are other instances where you want to make your tests more customizable. In those cases, you should use the same technique outlined in this recipe. Instead of hardcoding the test data, however, you can put your test data into an XML file. Your suite() method would parse the XML file and then create a TestSuite containing the test data defined in your XML.

See Also

Recipe 4.5 explains how JUnit normally instantiates and runs test cases.