SAMPLE CHAPTER

IntelliJ IDEA IN ACTION

Duane K. Fields
Stephen Saunders
Eugene Belyaev

MANNING

IntelliJ IDEA in Action

by Duane K. Fields
Stephen Saunders
Eugene Belayev
with
Arron Bates

Sample Chapter 7

brief contents

7 Testing applications with JUnit

In this chapter...

- Essentials of automated unit testing with JUnit
- Creating a JUnit file template to quickly build your own tests
- Running JUnit tests in the IDE and interpreting the results

You could jokingly say that writing software doesn't create bugs, testing does. But if you ever intend to have an application used by anyone of consequence (a paying customer, for example), then the application must undergo some form of testing. Most applications also go through many changes and improvements. Testing becomes far more difficult the longer an application is in development, because you're not just testing new features—you have an obligation to maintain the old features, as well.

This area of continuing development is where automated unit testing comes into its own. Writing tests at the same time as the application code not only confirms the proper working of the application in the short term but also provides an automated means to confirm the application continues to work in the future.

JUnit (an open source project with excellent pedigree) has become a de facto standard in unit testing, and it's almost ubiquitous for unit testing in Java. IDEA has a very high degree of integration with JUnit. This integration provides the most productive means to write, manage and run unit tests without leaving the comfort of the IDE.

7.1 Testing applications with JUnit

Thorough, automated suites of unit tests are a cornerstone of the Extreme Programming movement and should be part of any large-scale development project. For the uninitiated, *unit tests* are used to programmatically verify the operation of code components, often down to the individual methods. By developing extensive unit tests for your code and running them often as part of the build and verification process, you can help reduce bugs and avoid regressions. IDEA encourages this practice through its integrated support for JUnit.

JUnit is a free unit-testing framework developed by superstars Kent Beck (of Extreme Programming fame) and Erich Gamma (master of design patterns). The package is built around a straightforward API that lets you create tests that verify your code's correct operation. You can learn the details of the framework and the JUnit API at http://www.junit.org. For the brave and impatient, we'll give a quick overview to get you started.

7.1.1 Understanding the JUnit philosophy

The benefit of using the JUnit framework is that it lets you write automated unit tests for your Java code so easily that you can't reasonably justify *not* doing so. The API is simple, readable, and—most important—painless.

Why you need automated unit tests

Automated unit testing is the easiest way we know to prevent regressions and to test those code nooks and crannies that are often overlooked during manual integration testing. As an added bonus, having code built on a solid foundation of unit tests can give you the confidence to explore design improvements, refactorings, new features, and other changes that you might otherwise deem too risky to pursue. If you can trust your unit tests, recertifying your code is as simple as rerunning your tests.

When to write unit tests

Everyone has their own opinion on testing, but most developers share a basic tenet: the earlier the better. Members of the Extreme Programming camp insist on having you write your tests first, before you begin coding the classes they're being designed to test. This approach has several advantages, such as giving you an idea of the type of operations that are required of the code and assuring the testability of your class. Tests are easy to write, but if you don't get in the habit of writing unit tests as you go, it's easy to fall behind. A common suggestion you'll hear is "Code a little, test a little."

Another good practice is to create a new unit test for each bug that is reported. Design the test to exploit the bug and reproduce its behavior. The unit test will of course fail until the bug is resolved, making it easy to know when you've fixed it. Just code until the test passes! This also ensures that the bug will never bite you again, as long as you continue running your unit tests.

When to run unit tests

You should run your unit tests frequently to stop bugs from creeping into your code. Ideally, every developer should run the full set of unit tests before each check-in (refer to chapter 8 for information on version control), to ensure that none of their changes have broken any of the existing code. You should also consider running unit tests on a nightly basis, perhaps as part of a nightly build process. Java build tools such as Ant fully integrate with JUnit, allowing you to easily automate not only the running of tests but the reporting of the results as well.

7.1.2 Exploring the JUnit API

JUnit tests are collections of methods designed to exercise all the possible operations your code is expected to perform. These testing methods are included in a Java class that is passed to a JUnit-aware application, which can then put them through their paces and report the results. A *test case* is a class that contains the testing code.

Test cases and test methods

All JUnit tests stem from the base class `junit.framework.TestCase`. To create your own test case, you extend this class. There are no required methods to implement, and there is no need to modify your existing code. What could be simpler? However, in order to test something, you must create at least one test method to exercise a portion of your code. Any test case can contain as many test methods as you require. Test methods should be fine grained, testing a single operation critical to the correct operation of your code. They should also be standalone and not require any special setup or be expected to run in any particular order. Not only does this make them easy to write, it also makes them easy to use. All that is required by the framework is that your test methods begin with the name `test` and adhere to the following method signature:

```
public void testSomething()
```

Test methods can also declare exceptions.

Beyond starting with the word `test`, it doesn't matter what you call your test, but you should probably name it something meaningful to help indicate its purpose—this name is used as a label when running and reporting on the results of the test. You're free to add other methods to the test class as needed to assist in testing.

How JUnit runs your tests

The reason for the naming restriction is that JUnit relies on Java's reflection mechanism to locate and run the individual test methods. When passed into an application that knows how to run JUnit tests (like IDEA), the test case's test methods can be determined on the fly. This means all you have to do to write new tests is create new methods and add them to a test case. No configuration files, no changes to your source code, no flaming hoops to jump through. You may notice that the test method doesn't return a boolean as you might expect. So how does a test pass? A test passes as long as it doesn't fail. And failing tests is the job of *assertions*.

Assertions

If you're familiar with JDK 1.4's `assert` keyword, be advised that it isn't used in the JUnit framework, but the concept is similar. Like the `assert` keyword, JUnit's assertion methods are designed to evaluate boolean pass/fail conditions. For example, a method may verify that a variable has a certain value, a list isn't empty, or that two objects that should be equivalent really are.

If any assertion fails, the test fails with a *failure*, an expected or tested-for condition; if any uncaught exception is thrown, the test fails with an *error,* an

unexpected or untested-for condition. That's all there is to it. You could get by with a simple boolean test, but to make things easy, there are convenience methods for testing all sorts of common conditions. There are also methods that include a message parameter, which is reported back to the user when a failure occurs. Here are some of the assertion methods you're likely to use:

- `assertEquals(String message, int expected, int actual)`
- `assertTrue(String message, boolean condition)`
- `assertNotNull(Object object)`

If you look at the JUnit API, you'll see that there are assertion methods that accept `Objects`, `Strings`, and all the primitives as well. There are also negative equivalents, equality checks, and other combinations that make things simple for you, the busy developer.

A simple unit test example

Listing 7.1 shows an example test case so you can see these methods used in context. It's designed to test a simple encryption class with two methods, `encrypt()` and `decrypt()`. Although it isn't an exhaustive or complicated test case, it illustrates the intent of the unit-testing framework.

Listing 7.1 An example of a simple unit test

```
import junit.framework.*;

public class CryptTest extends TestCase {

  public void testEncrypt() {
    String plainText = "convoy sails for England tonight";
    String crypted = CryptUtil.encryptString(plainText);
    assertFalse("Encrypted value should not be equal to the"+
      " original", crypted.equals(plainText));
  }

  public void testDecrypt() {
    String plainText = "my password is elephant";
    String crypted = CryptUtil.encryptString(plainText);
    String decrypt = CryptUtil.decryptString(crypted);
    assertEquals("Decrypted value should be equal to the"+
      " original", plainText, decrypt);
  }
}
```

Test fixtures

The previous example is simple, but not all tests are so straightforward. For example, if a series of tests requires some amount of setup, you can override the `setUp()` method of `TestCase`. This method is called immediately before each test method, followed by a call to `tearDown()`, which can be overridden to perform any necessary cleanup. This shared setup code is known as a *test fixture*.

The benefit of using test fixtures is best illustrated with an example. Say you're testing code responsible for managing a message board or discussion forum. You've determined that a series of tests is required to verify that you can manipulate postings appropriately, such as changing the subject or author. Each of these operations can be tested by a single test method, but they all must act on an existing message. In this case, a test fixture is the best way to prepare the system for the test methods. The `setUp()` method is responsible for creating a new posting, and the `tearDown()` method is used to delete the test message from the system following each test method.

Test suites

Related test cases are grouped into logical collections known as *test suites*. When running your unit tests, you can run not only individual test cases, but test suites as well. Generally, you'll want to have a single test suite that relates all your unit tests together so you can run them in a single operation. Additional hierarchy is possible, because test suites can include other test suites. By combining groups of related tests into suites, you can avoid running the entire test suite if you just want to test one area of the code. For example, you may group all the security-related tests separately from your database tests, allowing you to spot test certain areas of your program.

Test runners

An application that knows how to interpret and run JUnit test cases is a *test runner*. Although JUnit ships with its own simple test runner, as does Ant, IDEA includes an integrated test runner with many more features. We'll discuss this in detail in the next section.

7.2 Adding test cases to your project

IDEA doesn't provide the ability to create new JUnit test case classes explicitly, but it's easy to add this capability through IDEA's file templates feature. As discussed in chapter 12, file templates allow you to create a starting point for new

files created through the **Project** window. Using templates, you can start off with your basic class framework already defined, rather than an empty document.

7.2.1 Creating a test case from a file template

For our example template, we have followed the canonical JUnit test case structure and the optional constructor, as well as stubbing out the fixture methods. The complete text of our JUnit file template is shown in listing 7.2. Of course, you may prefer a slightly different layout, so feel free to customize it to your liking.

Listing 7.2 A file template for producing JUnit tests

```
#parse("File Header.java")
package ${PACKAGE_NAME};
import junit.framework.TestCase;

public class ${NAME} extends TestCase {
    public ${NAME}(String test) {
        super(test);
    }

    /**
     * The fixture set up called before every test method.
     */
    protected void setUp() throws Exception {
    }

    /**
     * The fixture clean up called after every test method.
     */
    protected void tearDown() throws Exception {
    }

    public void testSomething() throws Exception {
    }
}
```

As you can see, this template covers all the basic structure required by the JUnit API. All you have to do now is write your test methods, starting with adding some logic to the `testSomething()` stub, which as written doesn't accomplish a heck of a lot (but at least it always passes!).

7.2.2 Adding the JUnit library to your Classpath

IDEA ships with the latest version of the JUnit JAR file, but by default it isn't included in your project's Classpath. This causes IDEA to display import failure

messages in the editor, and your project won't compile. To correct this, you must use the Project settings and add either a local copy of junit.jar (or the version at $IDEA_HOME/lib/junit.jar) to your Classpath. You may want to visit the JUnit website and download the latest release, its source code, and its API reference in order to create a reusable library, as described in chapter 4.

WARNING IDEA's bundled JUnit library is built atop the latest release of JUnit (version 3.8.1 at the time of this writing). If you include an older version of the JUnit APIs in your Classpath, IDEA will be unable to execute tests.

As we mentioned in chapter 4, IDEA modules have two distinct types of source paths that you can define, one for production sources and one for test sources. These have distinct output paths as well. If you choose separate output paths, you'll be able to package or deploy your application without having to include your test classes. When you're running unit tests, IDEA will automatically include your test case output path in your Classpath.

From a technical standpoint, it doesn't matter where your test sources and classes live. However, we recommend that you always place your tests in their own source tree, and make use of IDEA's tests paths.

7.3 Running test cases in IDEA

IDEA's integration of JUnit makes it easy to run test cases directly from the IDE and provides a number of convenience features. You can run any test or suite of tests with a single click. Any resulting failures can be corrected by jumping straight to the source of the problem, because IDEA provides links in its test reports back into your source tree. IDEA also lets you debug your tests by stepping through the execution one line at a time using the debugger.

7.3.1 Creating a Run/Debug configuration for your test

There are a number of different ways to run your test cases from within IDEA, but the most common method is through the **Run** menu. As we discussed in chapter 5, the **Run** menu and the **Run/Debug Configurations** dialog are used to execute applications within IDEA. They're also used to set up configurations for unit testing. Here you select test cases (or suites of test cases) to run via the **JUnit** tab. The **Run/Debug Configurations** dialog is shown in figure 7.1.

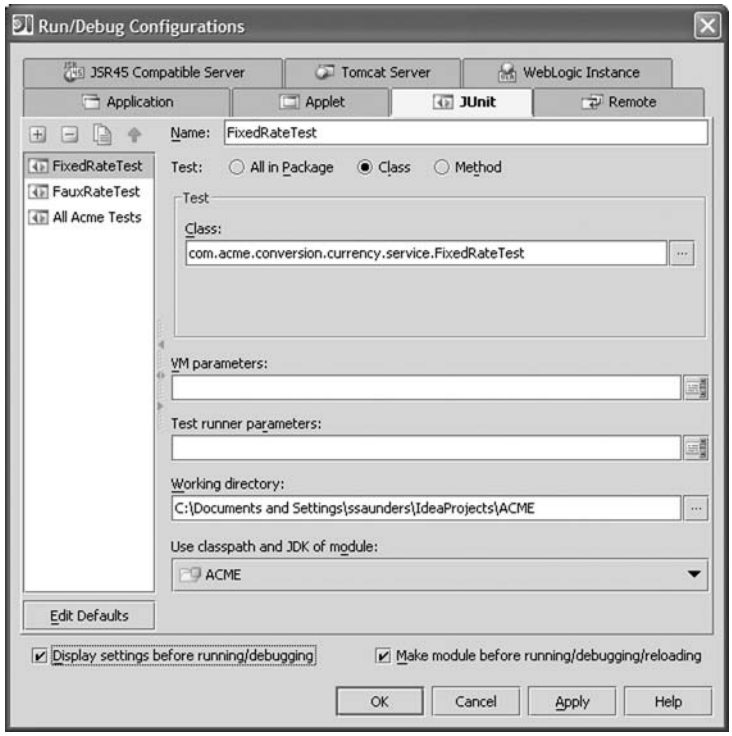

Figure 7.1 Use the Run/Debug Configurations panel to build one-click JUnit launch targets.

Selecting a test case to run

As with the other execution target types, the **JUnit** tab in the **Run/Debug Configurations** dialog lets you select a Java class that will become the target of this configuration. In this case, however, you are selecting a test case rather than an executable class. You can choose any test class in your Classpath—IDEA will even find classes for you and present an appropriately filtered list. Click the browse button next to the **Class** field to select the class by name, or navigate through your project tree. The list is filtered to include only JUnit tests—only classes that extend `junit.framework.TestCase` appear.

Select the test case you want to run from the class browser. You can name your test configuration whatever you wish; this is an arbitrary label for use in the **Run** menu. You also have the option to expand your selection to include all the test cases in a particular package or narrow it to a single method of your test. If you select the **Test Method** radio button option, an additional field appears, allowing

you to choose which method to run from the selected test case. To select more than a single test method from a test case without running all of them, you must create your own test suite, as described in the JUnit documentation.

Changing the test's working directory

The **Working directory** option in the **Run/Debug Configurations** dialog lets you change the base directory for all relative file paths. This is useful if your test cases must read or write data files as part of the testing process. Before executing your tests, the test runner changes to this directory. By default, this directory is the same directory your project file is in.

Passing parameters to the VM

The VM **parameters** field in the **Run/Debug Configurations** dialog is used to pass system properties or VM options such as the maximum heap size. Any arguments specified here are passed to the JVM executing your tests, just as they would be when directly running an application.

Passing parameters to the test runner

IDEA's test runner application ultimately relies on the test-running application included with the JUnit framework. As such, you can pass parameters to the underlying test runner by entering them in the appropriate configuration field. However, since the other configuration options give you an easier avenue of controlling how the tests are run, there's not much benefit to be gained through extra parameters; but this option is provided in the **Run/Debug Configurations** dialog for the sake of completeness. Refer to the JUnit documentation for a list of current options.

Selecting the appropriate module

As you learned in chapter 4, IDEA manages most source and Classpath information at the module level. In the **Run/Debug Configurations** dialog, you'll therefore need to select the module to which the currently selected test case should run under if it exists in more than one module. This also determines the JDK used to execute the tests. An exception to this rule is when you're running all the tests in a given package.

Triggering a build automatically

The two checkbox options **Display settings before running/debugging** and **Make module before running/debugging/reloading** in the **Run/Debug Configurations** dialog behave exactly as they do for the other types of execution targets

discussed in chapter 5. If enabled, the first option displays the setup dialog each time you run the test, giving you the option of tweaking the settings before execution. The second option causes IDEA to force a rebuild before running the test. Enabling this option lets you edit your tests and then run them without having to explicitly request a new build. On the other hand, you can disable this option if you're rerunning tests without code changes. Remember that these options are global, and they affect all your Run/Debug configurations, not just your unit tests!

7.3.2 Running your unit test configuration

The basic use of Run/Debug configurations was covered in chapter 5. Unlike application targets, which execute via their `main()` method, JUnit configurations pass the selected test class or classes to a test runner for processing—no `main()` method is required. The test runner locates and executes the tests and reports the results.

Running a selected Run/Debug configuration

Once your test target is configured, select it from the Run/Debug drop-down on the toolbar in the **Run/Debug Configurations** dialog, or from the **Run** menu, and then click the **Run** icon on the toolbar or press **Shift-F10**. IDEA understands that running a JUnit test means launching the test runner, rather than executing the class directly. JUnit tests can be distinguished from other entries in the **Run** list by their icon.

TIP All tests run in the background, and you can execute multiple tests simultaneously if you wish. Each test gets its own tab in the **Run** tool window.

Defining temporary test targets

IDEA provides a convenient shortcut for creating and running test targets. When you select a package, class, or method in the project or structure windows, the right-click pop-up menu provides an option for running test cases for that selection, if there are any. If you've created a testing Run/Debug configuration, it's used. If not, a temporary target is created and added to the list of test configurations, and the test executes. Temporary targets are particularly handy when you're trying to debug a single test method. You can create a temporary target to run the method, using it until you correct the problem.

This shortcut works from the editor as well. Right-click inside the body of a test method from the editor, and select the **Run** option to run the test (or press **Ctrl+Shift+F10**). Clicking anywhere else inside the file allows you to execute the entire test case. See figure 7.2.

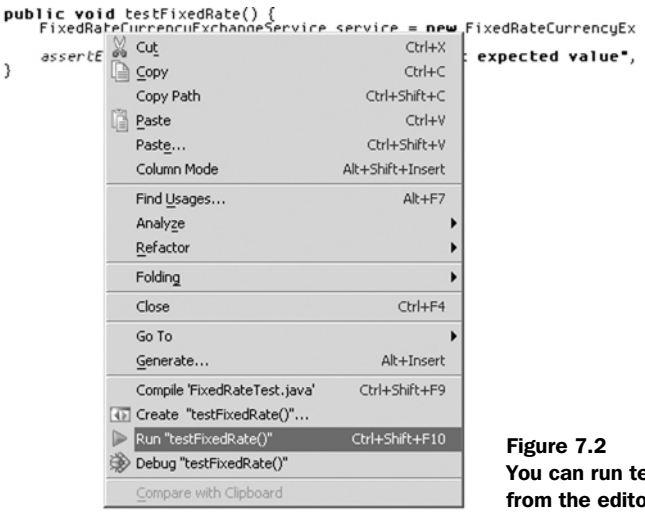

Figure 7.2
You can run temporary JUnit targets from the editor's context menu.

A Run/Debug configuration created in this manner is considered temporary, as described in chapter 5. It appears ghosted in the selection menu. To save it, select the corresponding **Save** option from the list under the **Run/Debug** drop-down. Even if you don't save it, you'll be able to tweak its behavior by selecting its entry in the **Run/Debug Configurations** dialog.

WARNING IDEA allows only a single temporary target to exist per project. If you create a second one, it will replace the first.

Debugging test cases

Not only does IDEA help you run your test cases, but it also helps you debug them. If you click the **Debug** icon instead of the **Run** icon in the toolbar in the **Run/Debug Configurations** dialog, your test cases execute in the debugger. This lets you set breakpoints and step through your code during execution, as described in chapter 6. You can set breakpoints in the test cases or in the application code the test cases call. Either way, this is a great technique to figure out why a test that ran fine last week suddenly blows up!

7.4 Working with IDEA's JUnit test runner

Each time you activate the **Run** or **Debug** command for your tests, IDEA invokes its test runner and opens a devoted tab in the **Run** tool window. IDEA's

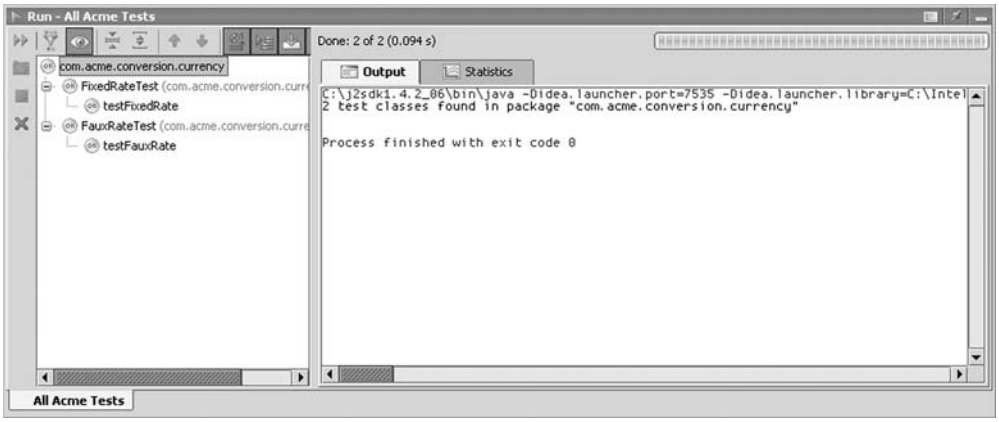

Figure 7.3 IDEA's JUnit test runner gives you complete access to test results and statistics.

test running can show you much more than just which tests passed and which failed. Through IDEA's JUnit test runner, you can also view any output or error messages your test cases produced, as well as how long they took to run and how much memory they used.

7.4.1 Exploring the JUnit tool window

When you run unit tests in IDEA, the test runner interface appears in a tab of the **Run** window, which pops up automatically when you begin the testing session. A typical example is shown in figure 7.3. Note that running applications and unit tests share the same tool window, but each target appears in its own tab.

The test runner toolbar

The **JUnit** tool window has a number of toolbar options. We'll discuss each of these in turn, and for your reference, they're shown in table 7.1.

Table 7.1 The JUnit toolbar affords you complete control over your JUnit execution.

Icon	Shortcut	Function
	Ctrl+F5	Rerun Test
		Hide Passed Tests
		Track Running Tests

continued on next page

Table 7.1 The JUnit toolbar affords you complete control over your JUnit execution. *(continued)*

Icon	Shortcut	Function
	Ctrl+Numpad(+)/Ctrl+Numpad(-)	Collapse All/Expand All
	Ctrl+Alt+Down, Ctrl+Alt+Up	Previous, Next Failed Test
		Select First Failed Test When Execution Finished
		Scroll to Stacktrace
		Auto Scroll to Source
		Open Source at Exception
	Ctrl+Break	Dump Threads
	Ctrl+F2	Stop
	Ctrl+Shift+F4	Close

The test tree structure

The left pane of the test runner window includes a tree structure that represents all the tests present in the current test configuration. The *root*—the topmost element of the tree—represents the entry point you selected to run; this may be a package, a test suite, or test case. If you're running an individual method, an implicit test suite is created for you. If you've nested suites of tests together, then additional levels of hierarchy are present. Because you're free to create suites of suites and so on, there is no limit to the levels of hierarchy you can create. In all cases, the innermost elements (the *leaf nodes*) are the individual tests, which come from your test methods. Each test is represented by an icon, which represents its current state. The meanings of these icons are summarized in table 7.2.

Table 7.2 The test tree keeps track of the state of all your tests.

Icon	Description
	Test Error
	Test Failed
	Test in Progress (animated)
	Test Passed
	Test Paused
	Test Terminated
	Test Not Run

Navigating through the test tree

As with other tool windows, the **Collapse All** and **Expand All** icons control the appearance of the test tree. Expanding all the entries lets you get to all the tests, whereas collapsing them limits the list to your top-level test cases or suites. These options are unavailable if only a single test case is involved, because there is only one level of hierarchy to deal with. Otherwise, you can use the tree controls to expand and contract individual nodes of the tree as desired.

You can achieve a similar navigation through the keyboard via the left and right arrow keys. Pressing the left arrow collapses the current node, and the right arrow expands it. You can visit each node in the tree by continually pressing the right arrow key until you've traversed all the entries. The up and down arrows similarly allow you to move between the individual tests, test cases, and test suites.

7.4.2 Monitoring testing progress

Once you begin testing, a message at the top of the **JUnit** window appears, showing the total number of tests being run in this session. After the tests have been run, the message indicates the number of failures (if any) and the total amount of time elapsed. All tests are run sequentially, one after the other. Tests are never run in parallel, ensuring that your tests don't interfere with each other. Keep in mind that the order in which tests are run is never guaranteed.

Tracking completion with the testing progress bar

The testing progress bar at the top of the **JUnit** window shows the percentage of tests that have been executed so far. This bar updates continuously through the testing process, as each test is completed, and represents the relative percentage of completion. The color of the bar indicates the current pass/fail status of your testing session. The bar's segments appear green if all your tests have completed successfully so far or red if any errors or failures were encountered.

This bar tracks progress across all the tests being run in this session, not just those that belong to the current test case or test suite. Also note that the test progress bar indicates the relative number of tests remaining, but not necessarily the amount of time left, because each test will take a different amount of time to complete.

Watching the currently running test

If you enable the **Track Running Test** option in the toolbar in the **JUnit** window, the test runner selects each test case as it's running, allowing you to monitor its output or runtime statistics as they're generated. When one test completes, the test runner automatically selects the next one.

7.4.3 Managing the testing session

IDEA lets you manage the currently running test session, just as it does when running other applications. You can stop or rerun tests if necessary. If you take no action, the test runner runs along happily on its own until all the tests have finished running.

Aborting a test in progress

You can end your testing session at any time by clicking the **Stop** button in the **JUnit** window (or pressing **Ctrl+F2**). Doing so shuts down the VM immediately, terminating any tests that are currently in progress. The icons of the tests tell you which tests completed, which were terminated, and which were never run, as shown in table 7.2. Note, however, that clicking **Stop** stops the entire testing session, not just the currently running test.

Running your tests again

You can easily rerun your testing session without having to leave the **JUnit** window. If you click the **Rerun** icon (**Ctrl+F5**), all the current tests are run again using the same settings as in the initial run. If enabled in your test configuration setup, your project is recompiled as needed before the tests are run. The new test results replace the old results unless you've locked down the current tab by selecting the **Pin Tab** option on the **Tab** context menu.

Dumping thread information to the output window

As in the debugger, while a test is running, you can ask the VM to give you some insight into thread activity. Clicking the **Dump Threads** icon in the **JUnit** window outputs the state of all your threads to the JUnit output stream. This operation doesn't affect your running tests and doesn't stop the testing process. You can view the results in the **Output** tab (along with the rest of your test output) by selecting the topmost test from the test tree.

7.4.4 Analyzing test results

When it's all said and done, you'll want to review the results of your tests and, if there were any failures, find out why they happened. When your tests have been run, a message at the top of the window summarizes the number of failures, if any, as well as the number of tests run.

Difference between errors and failures

As far as JUnit is concerned, encountering a runtime error or throwing an uncaught exception inside a test is enough to fail a test. IDEA, however, differentiates between these types of errors and a test method failing an assertion check. As shown in table 7.2, different icons are assigned to differentiate these two distinct conditions and highlight the errors. Errors include runtime exceptions, declared exceptions, and other problems: for instance, if a test case's class can't be found or can't execute for some reason. In figure 7.4, you see a number of failed test cases. In the **Output** window, it's easy to spot the problem—the rate returned by the fixed rate service wasn't the rate that was expected.

This distinction between errors and failures can be important when you're designing test cases. To take advantage of it, we suggest that you design your test cases such that exceptions are thrown only when a problem that prevents running of the test is encountered, and that a legitimate failure is the result of unsuccessful assertion. This strategy will allow you to more easily spot tests that can't run properly due to configuration problems or other issues rather than a true bug in your code.

Hiding the results of successful tests

Enabling the **Hide Passed Tests** option on the toolbar in the **JUnit** window alters the test tree to show only the tests that failed or encountered errors, thus allowing you to concentrate on the problems. This button is a toggle; you can toggle the

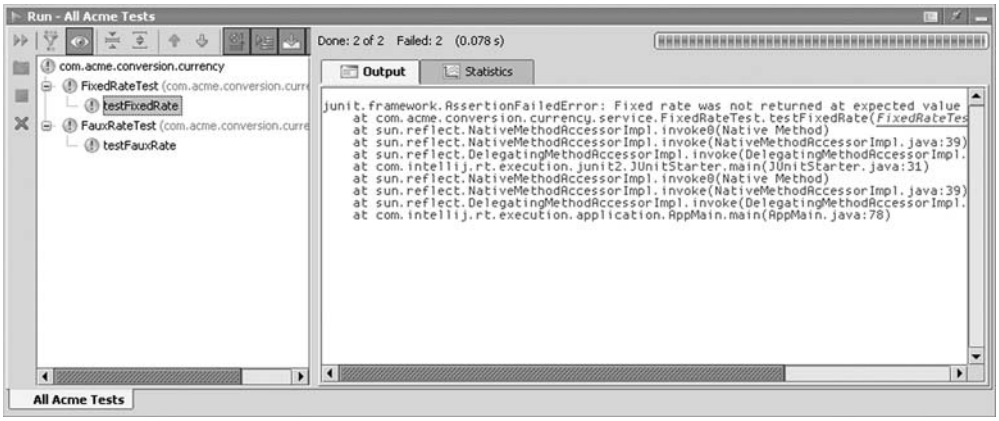

Figure 7.4 The JUnit tool window shows you the output from tests encountering both failures and errors as you select them. Failures are distinguished from errors by their icons in the test tree.

view of passed test cases on or off at any time. If a test case or suite contained no failing tests, it's hidden as well.

Navigating through the failed tests

Doubtless you'll encounter failed tests. Using the **Previous** and **Next Failure** arrows (or **Ctrl+Alt+Up** and **Ctrl+Alt+Down**, respectively) you can easily navigate through these failures. Each time you move forward or backward through the list, the next failed test is selected in the test tree. To automatically select the first failure upon completion of your tests, enable the **Select First Failed Test** option from the **JUnit** window's toolbar. You may also determine your starting point by clicking any test in the tree.

Reviewing test cases

In order to learn why a test failed, you need to understand the steps it was taking by reviewing the source code behind it. Double-clicking any test case in the tree (failed or not) takes you straight to that test's source code in the editor window. By backtracking into the source code behind the test as you review the test output messages, you should be able to spot the problem.

TIP Don't overlook the possibility that your bug might be in the test itself, rather than in the code you're testing!

Similarly, you can right-click a test in the tree and select either the **Jump to Source (F4)** or **Show Source (Ctrl+Enter)** option to review the test's source code. The difference between the two options is that **Jump to Source** moves your focus to the editor window, while **Show Source** leaves you in the **JUnit** window. You can also use the **Autoscroll to Source** toolbar button to keep the source window matching the current selection in the **JUnit** tool window.

Rerunning failed tests

You also have the option of running an individual test again by right-clicking its entry in the test tree and selecting the appropriate run option. Suites and test cases can also be run this way and will execute all the tests below them as well. If you run a test in this manner, it clears the current set of results unless you've locked down the results tab by right-clicking it and selecting the **Pin Tab** option.

Unfortunately, you're limited to running individual tests, test cases, or suites. It isn't possible to run ad hoc collections of tests (for example, all the failing tests) without manually defining a test suite using the JUnit API.

Examining test error messages and output

In the **Output** tab of the **JUnit** window, you can examine the runtime output generated by each test. This includes any output sent to the standard output stream as well as standard error (which is displayed in red text). If any exceptions were thrown, or if a test failed an assertion, this output is displayed here as well. This tab is also active while the tests are running, allowing you to monitor your tests' progress. If you have a stack trace in your test's output, any reachable source code reference is hot-linked to your editor, allowing you to quickly view or edit the source code. In the case of an assertion failure where a message was specified in the test, this message is shown along with a stack trace.

IDEA conveniently isolates each test's output. Clicking an individual test in the tree shows only the output stemming from the selected test. As you might expect, selecting any node in the test hierarchy includes the output of all the tests below it.

Tracking the time and memory usage of each test

The **Statistics** tab in the **JUnit** window shows how much time it took to run each test case and how much memory was consumed during its run. You can view test statistics summarized up to the suite or test case level by selecting the appropriate level of hierarchy in the test case tree. Here's what the columns mean:

- **Time elapsed**—The number of seconds it took to run this test.
- **Usage Delta**—The amount of memory apparently consumed during this test.
- **Usage Before**—The amount of memory in use at the start of the test.
- **Usage After**—The amount of memory in use after the test has been completed.
- **Results**—A summary of test results. For individual tests, this column shows a pass or fail; but for suites and test cases, it shows the number of passed and failed tests.

Take these statistics with a grain of salt—the timing and memory usage data are collected only to give you an approximate gauge of test case performance. Many things can affect the accuracy of this data; for example, if the garbage collector runs during a test case execution, the amount of memory shown in the statistics is wrong. Nevertheless, the **Statistics** view is a good way to keep an eye on the general state of things, as shown in figure 7.5. For example, if your unit tests now take twice as long, you may want to investigate. Perhaps there are just more tests than before, but a recent change may have drastically affected system performance.

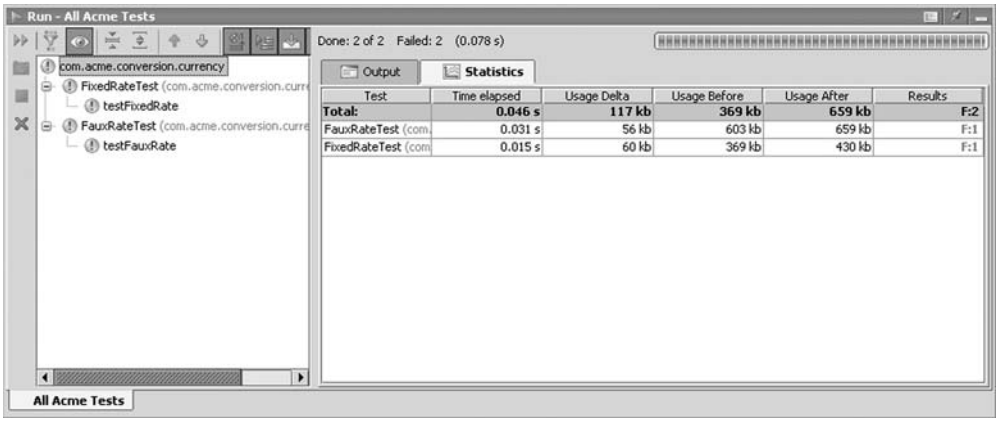

Figure 7.5 The Statistics tabs can show you how much memory was consumed by each test as well as how long it took to run each one.

Running tests with other test runners

The test runner built into IDEA is powerful and flexible. However, if you're nostalgic, you can use the classic test runners included with the JUnit framework. (Note, however, that you'll lose many of IDEA's JUnit integration features.) To do so, add a `main()` method, as shown in the following code, and run the class as a normal Run/Debug configuration target instead of a JUnit configuration:

```
public static void main(String[] args) {
    TestRunner testRunner = TestRunner();
    testRunner.run(StringUtilsTest.class);
}
```

Two test runners are included with the standard JUnit distribution: one in the `junit.swingui` package that creates a graphical interface, and another, text-only version in `junit.textui`. Import the appropriate version into your code. You can also use this technique to run other, third-party test runners if you wish.

The text-only version runs your JUnit tests in a standard IDEA **Run** window. A simple test summary report is shown in figure 7.6, along with a list of failures and their accompanying stack traces. Any errors are hot-linked back to the source of the failure inside your test case. In this example, you can see that a failure occurred inside the `testFixedRate()` method. Any output produced by the test cases is also displayed in this window, as are your assert failure messages.

The graphical test runner, shown in figure 7.7, is a little fancier than the text version, providing an alternate view of your test execution results. As shown, you

Figure 7.6 The textual test runner included with JUnit is simple, fast, and efficient.

can browse the list of test executions and failures as well as review summary information. All this information and more is available through IDEA's test runner, however, with better integration with the editor.

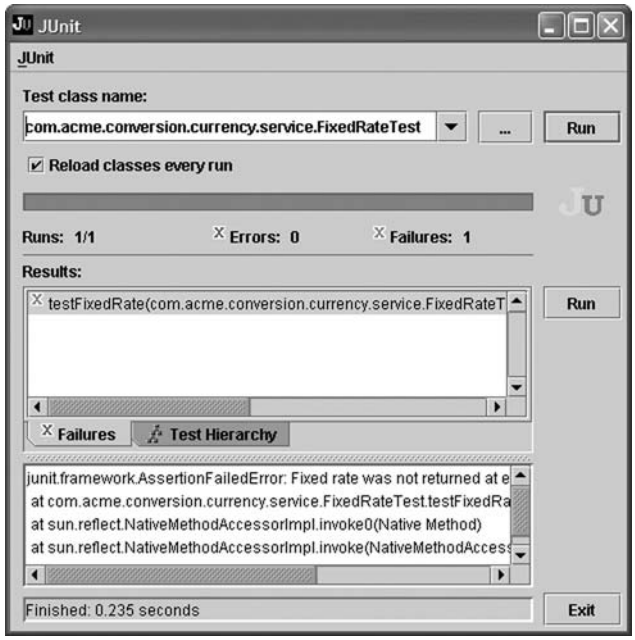

Figure 7.7
The graphical test runner included with the JUnit framework offers only basic options and no direct integration with IDEA.

Running unit tests through Ant

If your project calls JUnit through Ant, you should be able to run the same unit tests with IDEA. However, it's also possible to run your test cases through Ant directly, because it includes its own JUnit test runner (part of Ant's optional package, but included with IDEA).

When you execute test cases through Ant, the results are displayed in the Ant output window along with other build output. Any source code references present in the test output are hot-linked to the editor, but you won't get the GUI or other features of the IDEA test runner when running this way. Refer to the Ant documentation for details on creating JUnit targets in Ant.

7.5 Improving the quality of the ACME project

In the last chapter, you used IDEA's debugging features to identify a flaw in the ACME project. By implementing unit tests (and running them regularly), you can ensure that bugs like those don't creep in. Listing 7.3 shows a sample JUnit test, just for illustration, that can be added to your module and used to make sure the fixed-rate currency exchange service always returns the expected rate. Try adding it and running it with IDEA using the steps explained in this chapter.

Listing 7.3 JUnit test case that ensures `FixedRateCurrencyExchangeService` returns the correct rate

```
package com.acme.conversion.currency.service;

import junit.framework.TestCase;

public class FixedRateTest extends TestCase {
    /**
     * To test the fixed rate, we need to do the following:
     * 1) Get an instance of the FixedRateCurrencyExchangeService
     * 2) From it, request its rate
     * 3) Compare that rate with the expected value, which is 1.5
     */
    public void testFixedRate() {
        FixedRateCurrencyExchangeService service =
            new FixedRateCurrencyExchangeService();

        assertEquals("Fixed rate was not returned at " +
            "expected value",
            1.5d, service.getRate(), 0.01d);
      }
}
```

7.6 Summary

IDEA has very close and natural support for JUnit, a technology for automated unit testing that's quickly and pervasively being adopted by the industry. Automated unit testing is a common-sense practice to ensure software quality, and it's a cornerstone practice that underpins the agile software development movement and the goal of continuous integration. Continuous integration invites many changes early and often. Unit testing with IDEA and its JUnit support provides the easiest means for you to write, manage, and run these unit tests, and to interpret the results of their output so that bugs are identified and fixed. As an application moves forward, this is the easiest way to keep abreast of continual changes.

JAVA

IntelliJ IDEA IN ACTION

COVERS IDEA v5

Duane K. Fields ▪ Stephen Saunders ▪ Eugene Belyaev

If you work with IDEA, you know its unique power and have already seen a jump in your productivity. But because IDEA is a rich system you, like many others, are probably using just a small subset of its features. You can overcome this syndrome and see your productivity take another leap forward—all you need is this book.

For new users, this book is a logically organized and clearly expressed introduction to a big subject. For veterans, it is also an invaluable guide to the expert techniques they need to know to draw a lot more power out of this incredible tool. You get a broad overview and deep understanding of the features in IDEA.

The book takes you through a sample project—from using the editor for entering and editing code, to building, running and debugging, and testing your application. The journey then continues into the far corners of the system. Along the way, the authors carefully explain IDEA's features and show you fun tricks and productivity-enhancing techniques that are the result of their combined years of experience.

What's Inside

- INTELLIGENT EDITING OF Java • JSP • XML • HTML • custom file types
- HOW TO add plugins • customize and extend IDEA • fine tune your code • inspect and analyze • refactor • develop Swing and J2EE applications

Duane K. Fields is a software developer and manager. He co-authored Manning's best-selling *Web Development with JavaServer Pages*. Stephen Saunders is a software engineer with experience in knowledge management, financial services, and data management. Eugene Belyaev is the cofounder, president, and chief technology officer of JetBrains, the company that created IDEA.

"An absolute winner! A must for every IDEA developer's shelf!"

—Mark Monster
Software Developer, BT

"… straightforward and very clear."

—Mark Woon
Software Developer
Stanford University

"… an entertaining read with excellent examples. Even a seasoned IDEA user will find it helpful."

—Sean Garagan
Technical Director
Versata, Inc.

www.manning.com/fields3

ISBN 1-932394-44-3

MANNING $44.95 US/$60.95 Canada