CHAPTER 4

Testing with JUnit

"Any program feature without an automated test simply doesn't exist." [1]

Software bugs have enormous costs: time, money, frustrations, and even lives. How do we alleviate as much of these pains as possible? Creating and continuously executing test cases for our software is a practical and common approach to address software bugs before they make it past our local development environment.

The JUnit testing framework is now the de facto standard unit testing API for Java development. Ant integrates with JUnit to allow executing test suites as part of the build process, capturing their output, and generating rich color enhanced reports. In this chapter, we cover in more detail what testing can give us beyond knowing that our code is working well within some boundaries, then we cover the primary alternative to JUnit testing and why it is insufficient. The bulk remainder of the chapter, the largest part, is devoted to Ant's JUnit integration: how to use it, its limitations, and the techniques to make seamless integrated testing part of every build.

[1] *Extreme Programming Explained,* Kent Beck, page 57

4.1 REFACTORING

Assuming we accept the statement that all software systems must and will change over time, and also assuming that we all want our code to remain crisp, clean, and uncluttered of quick-and-dirty patches to accommodate the customer request du jour, how do we reconcile these conflicting requirements? Refactoring is the answer! Refactoring, as defined by Fowler, is the restructuring of software by applying a series of internal changes that do not affect its observable behavior (Fowler 1999).

Refactoring is one of the primary duties in agile methodologies such as eXtreme Programming. How can we facilitate constant refactoring of our code? Some of the key ways this can become easier is to have coding standards, simple design, a solid suite of tests, and a continuous integration process (Beck 1999). In an eXtreme Programming team, the names of the refactorings "replace type code with strategy" can become as commonplace as design patterns such as "the strategy pattern." Fowler's definitive *Refactoring* book provides a catalog of refactorings and when and how to apply them, just as the "Gang of Four" book (Gamma et al. 1995) is the definitive guide to design patterns.

We are not going to tell you how you should write your Java programs; instead, we refer you to some of the books in the Bibliography, such as *The Elements of Java Style* (Vermeulen et al. 2000) and Bloch's *Effective Java* (2001). These should be on the desk of every Java developer. We address Ant coding standards in appendix D. Just as good Java code should be simple, testable, and readable, your build file should be simple, testable, and follow coding standards; the XP methodology applies to build files and processes as much as to the Java source.

The remainder of this chapter is all about how to use Ant for testing. Continuous integration is a topic that will be touched upon in this chapter, but covered in more detail in chapter 16.

4.2 JAVA MAIN() TESTING

A common way that many Java developers exercise objects is to create a `main` method that instantiates an instance of the class, and performs a series of checks to ensure that the object is behaving as desired. For example, in our `HtmlDocument` class we define a `main` method as

```
public static void main(String args[]) throws Exception {
    HtmlDocument doc = new HtmlDocument(new File(args[0]));
    System.out.println("Title = " + doc.getTitle());
    System.out.println("Body = " + doc.getBodyText());
}
```

We are then able to run the program from the command-line, with the proper classpath set:

```
java org.example.antbook.ant.lucene.HtmlDocument
          test/org/example/antbook/ant/lucene/test.html
```

Using Ant as a Java program launcher, we can run it with the `<java>` task:

```
<java classname="org.example.antbook.ant.lucene.HtmlDocument">
  <arg value="test/org/example/antbook/ant/lucene/test.html"/>
  <classpath refid="test.classpath"/>
</java>
```

Writing `main` method checks is convenient because all Java IDEs provide the ability to compile and run the class in the current buffer, and certainly have their place for exercising an object's capability. There are, however, some issues with this approach that make it ineffective as a comprehensive test framework:

- There is no explicit concept of a test passing or failing. Typically, the program outputs messages simply with `System.out.println`; the user has to look at this and decide if it is correct.
- `main` has access to `protected` and `private` members and methods. While you may want to test the inner workings of a class may be desired, many tests are really about testing an object's interface to the outside world.
- There is no mechanism to collect results in a structured fashion.
- There is no replicability. After each test run, a person has to examine and interpret the results.

The JUnit framework addresses these issues, and more.

4.3 JUNIT PRIMER

JUnit is a member of the xUnit testing framework family and now the de facto standard testing framework for Java development. JUnit, originally created by Kent Beck and Erich Gamma, is an API that enables developers to easily create Java test cases. It provides a comprehensive assertion facility to verify expected versus actual results. For those interested in design patterns, JUnit is also a great case study because it is very pattern-dense. Figure 4.1 shows the UML model. The abstract `TestCase` class is of most interest to us.

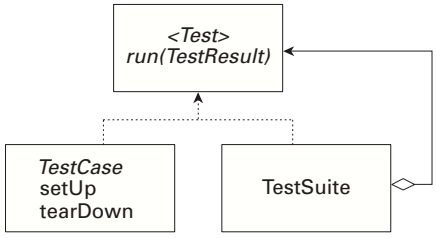

Figure 4.1
JUnit UML diagram depicting the composite pattern utilized by TestCase and TestSuite. A TestSuite contains a collection of tests, which could be either more TestSuites or TestCases, or even classes simply implementing the test interface.

4.3.1 Writing a test case

One of the primary XP tenets is that writing and running tests should be ***easy***. Writing a JUnit test case is intentionally designed to be as easy as possible. For a simple test case, you follow three simple steps:

1. Create a subclass of `junit.framework.TestCase`.
2. Provide a constructor, accepting a single String `name` parameter, which calls `super(name)`.
3. Implement one or more no-argument void methods prefixed by the word `test`.

An example is shown in the `SimpleTest` class code:

```
package org.example.antbook.junit;

import junit.framework.TestCase;

public class SimpleTest extends TestCase
{
    public SimpleTest (String name) {
        super(name);
    }

    public void testSomething() {
        assertTrue(4 == (2 * 2));
    }
}
```

4.3.2 Running a test case

`TestRunner` classes provided by JUnit are used to execute all tests prefixed by the word "test." The two most popular test runners are a text-based one, `junit.textui.TestRunner`, and an attractive Swing-based one, `junit.swingui.TestRunner`. From the command line, the result of running the text `TestRunner` is

```
java junit.textui.TestRunner org.example.antbook.junit.SimpleTest
.
Time: 0.01

OK (1 tests)
```

The dot character (`.`) indicates a test case being run, and in this example only one exists, `testSomething`. The Swing `TestRunner` displays success as green and failure as red, has a feature to reload classes dynamically so that it can remain open while code is recompiled, and will pick up the latest test case class each time. For this same test case, its display appears in figure 4.2.

4.3.3 Asserting desired results

The mechanism by which JUnit determines the success or failure of a test is via assertion statements. An *assert* is simply a comparison between an expected value and an

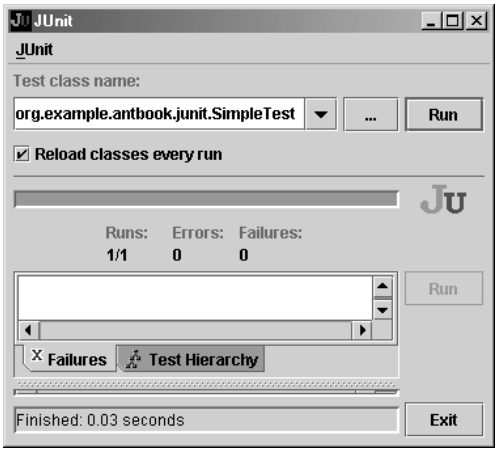

Figure 4.2
JUnit's Swing TestRunner

actual value. There are variants of the assert methods for each primitive datatype and for `java.lang.String` and `java.lang.Object`, each with the following signatures:

```
assertEquals(expected, actual)

assertEquals(String message, expected, actual)
```

The second signature for each datatype allows a message to be inserted into the results, which makes clear identification of which assertion failed. There are several other assertion methods:

- `assertEquals(expected, actual)`
  `assertEquals(String message, expected, actual)`
  This assertion states that the test `expected.equals(actual)` returns true, or both objects are null. The equality test for a `double` also lets you specify a range, to cope with floating point errors better. There are overloaded versions of this method for all Java's primitive types.
- `assertNull(Object object)`,
  `assertNull(String message, Object object)`
  This asserts that an object reference equals null.
- `assertNotNull(Object object)`,
  `assertNotNull(String message, Object)`
  This asserts that an object reference is not null.
- `assertSame(Object expected, Object actual)`,
  `assertSame(String message, Object expected, Object actual)`
  Asserts that the two objects are the same. This is a stricter condition than simple equality, as it compares the object identities using `expected == actual`.

- `assertTrue(boolean condition)`,
  `assertTrue(String message, boolean condition)`
  This assertion fails if the condition is false, printing a message string if supplied. The `assertTrue` methods were previously named simply `assert`, but JDK 1.4 introduces a new assert keyword. You may encounter source using the older method names and receive deprecation warnings during compilation.
- `fail()`,
  `fail(String message)`
  This forces a failure. This is useful to close off paths through the code that should not be reached.

JUnit uses the term *failure* for a test that fails expectedly, meaning that an assertion was not valid or a `fail` was encountered. The term *error* refers to an unexpected error (such as a `NullPointerException`). We will use the term *failure* typically to represent both conditions as they both carry the same show-stopping weight when encountered during a build.

4.3.4 TestCase lifecycle

The lifecycle of a `TestCase` used by the JUnit framework is as follows:

1. Execute `public void setUp()`.
2. Call a `test`-prefixed method.
3. Execute `public void tearDown()`.
4. Repeat these steps for each test method.

Any number of test methods can be added to a `TestCase`, all beginning with the prefix `test`. The goal is for each test to be small and simple, and tests will usually require instantiating objects. In order to create some objects and preconfigure their state prior to running each individual test method, override the empty `TestCase.setUp` method, and store state as member variables to your test case class. Use the `TestCase.tearDown` method to close any open connections or in some way reset state. Our `HtmlDocumentTest` takes advantage of `setUp` and `tearDown` (see later this chapter) so that all test methods will have implicit access to an `HtmlDocument`.

NOTE The `setUp` and `tearDown` methods are called before and after every test method is invoked, preventing one test from affecting the behavior of another. Tests should never make assumptions about the order in which they are called.

4.3.5 Writing a TestSuite

With JUnit's API, tests can be grouped into a *suite* by using the `TestSuite` class. Grouping tests may be a benefit to let you build several individual test cases for a particular subsystem and write an all-inclusive `TestSuite` that runs them all. A `TestSuite` also allows specific ordering of tests, which may be important—

although ideally the order of tests should not be relevant as each should be able to stand alone. Here is an example of a test suite:

```
public class AllTests extends TestSuite {
  static public Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(SimpleTest.class);
    return suite;
  }
}
```

You don't need to bother with test suites when running JUnit tests using Ant, because you can list a group of `TestCase` classes to run as a batch from the build file itself. (See section 4.6.2 for discussion of `<batchtest>`.) However, running a single `TestSuite` using the "running a single test case" trick in section 4.7.2 gives you flexibility in the grouping and granularity of test cases. Remember that a `TestCase` is a `Test`, and a `TestSuite` is also a `Test`, so the two can be used interchangeably in most instances.

4.3.6 Obtaining and installing JUnit

JUnit is just a download away at http://www.junit.org. After downloading the Zip or tar file, extract the junit.jar file. You must put junit.jar into ANT_HOME/lib so that Ant can find it. Because of Ant class loader issues, you must have junit.jar in the system classpath or ANT_HOME/lib; our recommendation is to keep your system classpath empty by placing such Ant dependencies in its lib directory.

Many IDEs can create JUnit test cases automatically from an existing Java class—refer to the documentation of your IDE for details. Be careful, however, not to let the habit of automatic test generation deter you from writing the tests first! We also encourage the exploration of the many great resources also found at the JUnit web site.

4.3.7 Extensions to JUnit

Because of its architecture, it is easy to build extensions on top of JUnit. There are many freely available extensions and companions for JUnit. Table 4.1 shows a few.

Table 4.1 A few notable companions to enhance the capabilities of JUnit testing

Name	Description
HttpUnit	A test framework that could be embedded in JUnit tests to perform automated web site testing.
JUnitPerf	JUnit test decorators to perform scalability and performance testing.
Mock Objects	Allows testing of code that accesses resources such as database connections and servlet containers without the need of the actual resources.
Cactus	In-container unit testing. Covered in detail in chapter 12.
DBUnit	Sets up databases in a known state for repeatable DB testing.

4.4 APPLYING UNIT TESTS TO OUR APPLICATION

This is the first place in our book where we delve into the application built to accompany this text. We could have written the book without a sample application and contrived the examples, but we felt that to have a common theme throughout the book would give you the benefit of seeing how all the pieces fit together.

Without a doubt, one of the key points we want to emphasize is the importance of testing. Sure, this book is about Ant, yet Ant exists as a tool for assisting with the development of software and does not stand alone. To reiterate: "any program feature without an automated test simply doesn't exist." For developers to embrace testing as a routine, and even *enjoyable*, part of life, it must be easy. Ant facilitates this for us nicely with the ability to run JUnit test cases as an integral part of the build.

Why is the "Testing" chapter the right place to start seriously delving into our application? Because the tests were written first, our application did not exist until there was an automated test in place.

4.4.1 Writing the test first

At the lowest level of our application is the capability to index text files, including HTML files. The Jakarta Project's Lucene tool provides fantastic capabilities for indexing and searching for text. Indexing a document is simply a matter of instantiating an instance of `org.apache.lucene.document.Document` and adding *fields*. For text file indexing, our application loads the contents of the file into a field called *contents*. Our HTML document handling is a bit more involved as it parses the HTML and indexes the title (`<title>`) as a title field, and the body, excluding HTML tags, as a contents field. Our design calls for an abstraction of an HTML document, which we implement as an `HtmlDocument` class. One of our design decisions is that content will be indexed from filesystem files, so we will build our `HtmlDocument` class with constructor accepting a `java.io.File` as a parameter.

What benefit do we get from testing `HtmlDocument`? We want to know that JTidy, the HTML parser used, and the code wrapped around it is doing its job. Perhaps we want to upgrade to a newer version of JTidy, or perhaps we want to replace it entirely with another method of parsing HTML. Any of those potential scenarios make `HtmlDocument` an ideal candidate for a test case. Writing the test case first, we have

```
package org.example.antbook.ant.lucene;
import java.io.IOException;
import junit.framework.TestCase;

public class HtmlDocumentTest extends DocumentTestCase
{
    public HtmlDocumentTest (String name) {
        super(name);
    }

    HtmlDocument doc;
```

```
    public void setUp() throws IOException {
        doc = new HtmlDocument(getFile("test.html"));
    }

    public void testDoc() {
        assertEquals("Title", "Test Title", doc.getTitle());
        assertEquals("Body", "This is some test", doc.getBodyText());
    }

    public void tearDown() {
        doc = null;
    }
}
```

To make the compiler happy, we create a stub `HtmlDocument` adhering to the signatures defined by the test case. Take note that the test case is driving how we create our production class—this is an important distinction to make; test cases are not written after the code development, instead the production code is driven by the uses our test cases make of it. We start with a stub implementation:

```
package org.example.antbook.ant.lucene;
import java.io.File;

public class HtmlDocument {
    public HtmlDocument(File file) { }
    public String getTitle() { return null; }
    public String getBodyText() { return null; }
}
```

Running the unit test now will fail on `HtmlDocumentTest.testDoc()`, until we provide the implementation needed to successfully parse the HTML file into its component title and body. We are omitting the implementation details of how we do this, as this is beyond the scope of the testing chapter.

4.4.2 Dealing with external resources during testing

As you may have noticed, our test case extends from `DocumentTestCase` rather than JUnit's `TestCase` class. Since our application has the capability to index HTML files and text files, we will have an individual test case for each document type. Each document type class operates on a `java.io.File`, and obtaining the full path to a test file is functionality we consolidate at the parent class in the `getFile` method. Creating parent class `TestCase` extensions is a very common technique for wrapping common test case needs, and keeps the writing of test cases easy.

Our base `DocumentTestCase` class finds the desired file in the classpath and returns it as a `java.io.File`. It is worth a look at this simple code as this is a valuable technique for writing test cases:

```
package org.example.antbook.ant.lucene;
import java.io.File;
import java.io.IOException;
import junit.framework.TestCase;
```

```
public abstract class DocumentTestCase extends TestCase
{
    public DocumentTestCase(String name) {
        super(name);
    }

    protected File getFile(String filename) throws IOException {
        String fullname =
                     this.getClass().getResource(filename).getFile();
        File file = new File(fullname);
        return file;
    }
}
```

Before implementing the `HtmlDocument` code that will make our test case succeed, our build must be modified to include testing as part of its routine process. We will return to complete the test cases after adding testing to our Ant build process.

4.5 THE JUNIT TASK—<JUNIT>

One of Ant's many "optional"[2] tasks is the `<junit>` task. This task runs one or more JUnit tests, then collects and displays results in one or more formats. It also provides a way to fail or continue a build when a test fails.

In order to execute the test case that we have just written via Ant, we can declare the task with the name of the test and its classpath:

```
<junit>
  <classpath refid="test.classpath"/>
  <test name="org.example.antbook.ant.lucene.HtmlDocumentTest"/>
</junit>
```

And, oddly, the following is displayed:

```
    [junit] TEST org.example.antbook.ant.lucene.HtmlDocumentTest FAILED
BUILD SUCCESSFUL
```

There are two issues to note about these results: no details were provided about which test failed or why, and the build completed successfully despite the test failure. First let's get our directory structure and Ant build file refactored to accommodate further refinements easily, and we will return in section 4.6 to address these issues.

4.5.1 Structure directories to accommodate testing

A well-organized directory structure is a key factor in build file and project management simplicity and sanity. Test code should be separate from production code, under unique directory trees. This keeps the test code out of the production binary distributions, and lets you build the tests and source separately. You should use a package hierarchy as usual. You can either have a new package for your tests, or

[2] See chapter 10 for a discussion on Ant's task types

mimic the same package structure that the production classes use. This tactic makes it obvious which tests are associated with which classes, and gives the test package-level access privileges to the code being tested. There are, of course, situations where this recommendation should not be followed (verifying package scoping, for example), but typically mirroring package names works well.

NOTE A peer of one of the authors prefers a different and interesting technique for organizing test cases. Test cases are written as public nested static classes of the production code. The advantage is that it keeps the production and test code in very close proximity. In order to prohibit packaging and deploying test cases, he takes advantage of the `$` that is part of a nested class filename and excludes them. We mention this as an alternative, but do not use this technique ourselves.

During the build, Ant compiles production code to the build/classes directory. To separate test and production code, all test-generated artifacts go into `build/test`, with classes into `build/test/classes`. The other products of the testing process will be result data and reports generated from that data. Figure 4.3 shows the relevant structure of our project directory tree.

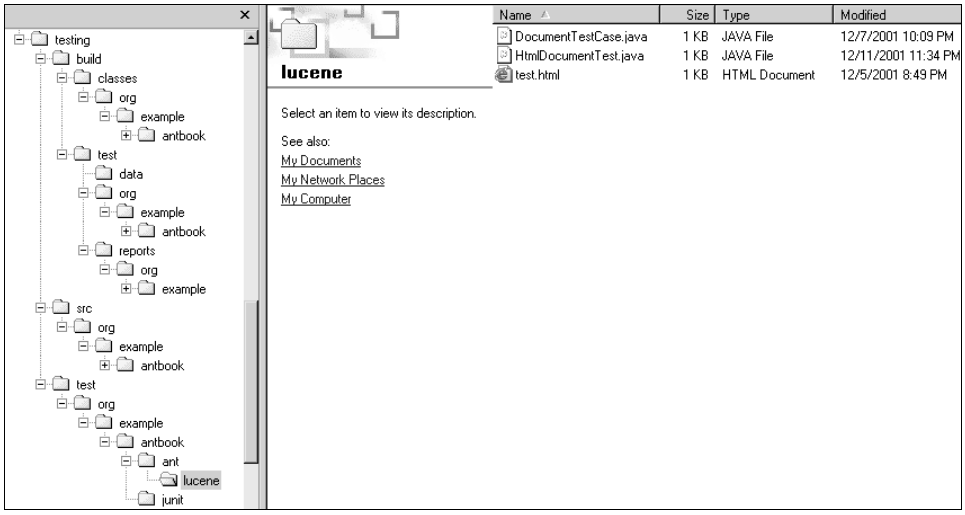

Figure 4.3 Our directory structure for unit test source code and corresponding compiled code and test results

4.5.2 Fitting JUnit into the build process

Adding testing into our build process is straightforward: simply add a few additional targets to initialize the testing directory structure, compile the test code, and then execute the tests and generate the reports. Figure 4.4 illustrates the target dependency graph of the build file.

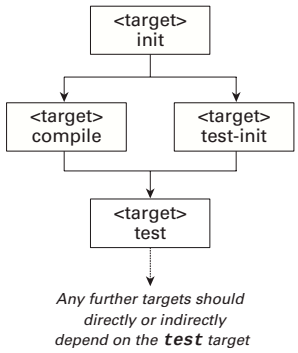

Figure 4.4 Refactoring our build process with unit testing targets

We use several build file properties and datatypes to make writing our test targets cleaner, to avoid hard-coded paths, and to allow flexible control of the testing process. First, we assign properties to the various directories used by our test targets:

```
<property name="test.dir" location="${build.dir}/test"/>
<property name="test.data.dir" location="${test.dir}/data"/>
<property name="test.reports.dir" location="${test.dir}/reports"/>
```

As we stated in chapter 3, when constructing subdirectories, like `test.data.dir` and `test.reports.dir`, of a root directory, you should define a property referring to the root directory and build the subdirectory paths from the root-referring property. If, for example, we had defined `test.data.dir` as `${build.dir}/test/data`, then it would not be possible to relocate the entire test output directory structure easily. With `test.dir` used to define the subdirectory paths, it is straightforward to override the `test.dir` property and move the entire tree. Another benefit could be to individually control where Ant places test reports (overriding `test.reports.dir`), so that we could place them in a directory served by a web server.

Compiling and running tests requires a different classpath than the classpath used in building our production compilation. We need JUnit's JAR file compilation and execution, and the test/classes directory for execution. We construct a single `<path>` that covers both situations:

```
<path id="test.classpath">
  <path refid="compile.classpath"/>
  <pathelement location="${junit.jar}"/>
  <pathelement location="${build.dir}/classes"/>
  <pathelement location="${build.dir}/test"/>
</path>
```

We originally defined the `compile.classpath` path in chapter 3; we reference it here because our test code is likely to have the same dependencies as our production code. The `test-compile` target utilizes `test.classpath` as well as `test.dir`:

```
<target name="test-compile" depends="compile,test-init">
  <javac destdir="${test.dir}"
         debug="${build.debug}"
         includeAntRuntime="true"
         srcdir="test">
    <classpath refid="test.classpath"/>
  </javac>

  <copy todir="${test.dir}">
    <fileset dir="test" excludes="**/*.java"/>
  </copy>
</target>
```

Note that in this particular example we are planning on building a custom Ant task so we set the `includeAntRuntime` attribute. Typically, you should set this attribute to `false`, to control your classpaths better. We follow the compilation with a `<copy>` task to bring over all non-.java resources into the testing classpath, which will allow our tests to access test data easily. Because of dependency checking, the `<copy>` task does not impact incremental build times until those files change.

4.6 TEST FAILURES ARE BUILD FAILURES

By default, failing test cases run with `<junit>` do not fail the build process. The authors believe that this behavior is somewhat backwards and the default should be to fail the build: you can set the `haltonfailure` attribute to `true` to achieve this result.[3] Developers must treat test failures in the same urgent regard as compilation errors, and give them the same show-stopping attention.

Adding both `haltonfailure="true"` and `printsummary="true"` to our `<junit>` element attributes, we now get the following output:

```
    [junit] Running org.example.antbook.ant.lucene.HtmlDocumentTest
    [junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.01 sec
BUILD FAILED
```

Our build has failed because our test case failed, exactly as desired. The summary output provides slightly more details: how many tests run, how many failed, and how many had errors. We still are in the dark about what caused the failure, but not for long.

4.6.1 Capturing test results

The JUnit task provides several options for collecting test result data by using *formatters*. One or more `<formatter>` tags can be nested either directly under `<junit>` or under the `<test>` (and `<batchtest>`, which we will explore shortly). Ant includes three types of formatters shown in table 4.2.

[3] The authors do not recommend `haltonfailure` to be enabled either. Read on for why.

Table 4.2 Ant JUnit task result formatter types.

<formatter> type	Description
brief	Provides details of test failures in text format.
plain	Provides details of test failures and statistics of each test run in text format.
xml	Provides an extensive amount of detail in XML format including Ant's properties at the time of testing, system out, and system error output of each test case.

By default, `<formatter>` output is directed to files, but can be directed to Ant's console output instead. Updating our single test case run to include both the build failure upon test failure and detailed console output, we use this task declaration:

```
<junit printsummary="false" haltonfailure="true">
  <classpath refid="test.classpath"/>
  <formatter type="brief" usefile="false"/>
  <test name="org.example.antbook.ant.lucene.HtmlDocumentTest"/>
</junit>
```

This produces the following output:

```
    [junit] Testsuite: org.example.antbook.ant.lucene.HtmlDocumentTest
    [junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.01 sec
    [junit]
    [junit] Testcase: testDoc(org.example.antbook.ant.lucene

                    .HtmlDocumentTest):FAILED
    [junit] Title expected:<Test Title> but was:<null>
    [junit] junit.framework.AssertionFailedError:
            Title expected:<Test Title> but was:<null>
    [junit]     at org.example.antbook.ant.lucene
                  .HtmlDocumentTest.testDoc(HtmlDocumentTest.java:20)
    [junit]
    [junit]

BUILD FAILED
```

Now we're getting somewhere. Tests run as part of our regular build, test failures cause our build to fail, and we get enough information to see what is going on. By default, formatters write their output to files in the directory specified by the `<test>` or `<batchtest>` elements, but `usefile="false"` causes the formatters to write to the Ant console instead. It's worth noting that the stack trace shown is abbreviated by the formatter, showing only the most important pieces rather than line numbers tracing back into JUnit's classes. Also, we turned off the `printsummary` option as it duplicates and interferes with the output from the `brief` formatter.

XML formatter

Using the `brief` formatter directed to Ant's console is very useful, and definitely recommended to allow quick inspection of the results and details of any failures. The `<junit>` task allows more than one formatter, so you can direct results toward

several formatters at a time. Saving the results to XML files lets you process them in a number of ways. Our testing task now evolves to this:

```
<junit printsummary="false" haltonfailure="true">
  <classpath refid="test.classpath"/>
  <formatter type="brief" usefile="false"/>
  <formatter type="xml"/>
  <test todir="${test.data.dir}"
        name="org.example.antbook.ant.lucene.HtmlDocumentTest"/>
</junit>
```

The effect of this is to create an XML file for each test case run in the `${test.data.dir}` directory. In this example, the file name will be `TEST-org.example.antbook.ant.lucene.HtmlDocumentTest.xml`.

Viewing System.out and System.err output

While it is typically unnecessary to have test cases write to standard output or standard error, it might be helpful in troubleshooting. With no formatters specified and `printsummary` either `on` or `off`, the `<junit>` task swallows the output. A special value of `printsummary` lets you pass this output through back to Ant's output: `printsummary="withOutAndErr"`. The `plain`, `brief`, and `xml` formatters capture both output streams, so in our example `printsummary` is disabled because we use the `brief` formatter to output to the console instead.

With a `System.out.println("Hi from inside System.out.println")` inside a `testOutput` method of SimpleTest, our output is

```
test:
    [junit] Testsuite: org.example.antbook.junit.SimpleTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.09 sec
    [junit] ------------- Standard Output ---------------
    [junit] Hi from inside System.out.println
    [junit] ------------- ---------------- ---------------
    [junit]
    [junit] Testcase: testSomething took 0.01 sec
    [junit] Testcase: testOutput took 0 sec
[junitreport] Using Xalan version: 2.1.0
[junitreport] Transform time: 932ms

BUILD SUCCESSFUL
Total time: 2 seconds.
```

Note that it does not identify the test method, `testOutput` in this case, which generated the output.

4.6.2 Running multiple tests

So far, we've only run a single test case using the `<test>` tag. You can specify any number of `<test>` elements but that is still time consuming. Developers should not have to edit the build file when adding new test cases. Enter `<batchtest>`. You can nest filesets within `<batchtest>` to include all your test cases.

TIP Standardize the naming scheme of your test cases classes for easy fileset inclusions, while excluding helper classes or base test case classes. The normal convention-naming scheme calls for test cases, and only test cases, to end with the word "Test." For example, `HtmlDocumentTest` is our test case, and `DocumentTestCase` is the abstract base class. We use "TestCase" as the suffix for our abstract test cases.

The `<junit>` task has now morphed into

```
<junit printsummary="true" haltonfailure="true">
  <classpath refid="test.classpath"/>
  <formatter type="brief" usefile="false"/>
  <formatter type="xml"/>
  <batchtest todir="${test.data.dir}">
    <fileset dir="${test.dir}" includes="**/*Test.class"/>
  </batchtest>
</junit>
```

The `includes` clause ensures that only our concrete test cases are considered, and not our abstract `DocumentTestCase` class. Handing non-JUnit, or abstract, classes to `<junit>` results in an error.

4.6.3 Creating your own results formatter

The authors of the JUnit task framework wisely foresaw the need to provide extensibility for handling unit test results. The `<formatter>` element has an optional `classname` attribute, which you can specify instead of `type`. You must specify a fully qualified name of a class that implements the `org.apache.tools.ant.taskdefs.optional.junit.JUnitResultFormatter` interface. Given that the XML format is already provided, there is probably little need to write a custom formatter, but it is nice that the option is present. Examine the code of the existing formatters to learn how to develop your own.

4.7 GENERATING TEST RESULT REPORTS

With test results written to XML files, it's a straightforward exercise to generate HTML reports using XSLT. The `<junitreport>` task does exactly this, and even allows you to use your own XSL files if you need to. This task works by aggregating all of the individual XML files generated from `<test>`/`<batchtest>` into a single XML file and then running an XSL transformation on it. This aggregated file is named, by default, `TESTS-TestSuites.xml`.

Adding the reporting to our routine is simply a matter of placing the `<junitreport>` task immediately following the `<junit>` task:

```
<junitreport todir="${test.data.dir}">
  <fileset dir="${test.data.dir}">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames" todir="${test.reports.dir}"/>
</junitreport>
```

The `<fileset>` is necessary, and typically will be specified using an include pattern of `"TEST-*.xml"` since that is the default naming convention used by the XML formatter of `<junit>`. The `<report>` element instructs the transformation to use either `frames` or `noframes` Javadoc-like formatting, with the results written to the `todir` directory. Figure 4.5 shows the `frames` report of this example.

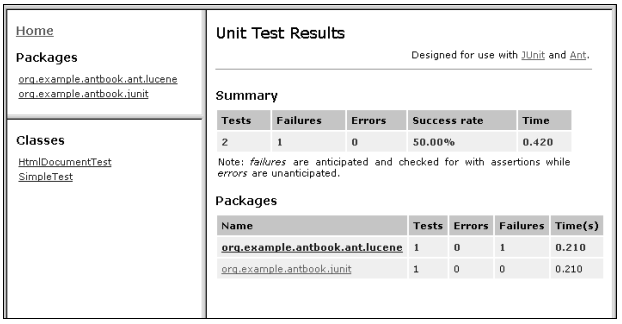

Figure 4.5
The main page, index.html, of the default frames <junitreport>. It summarizes the test statistics and hyperlinks to test case details.

Navigating to a specific test case displays results like figure 4.6.

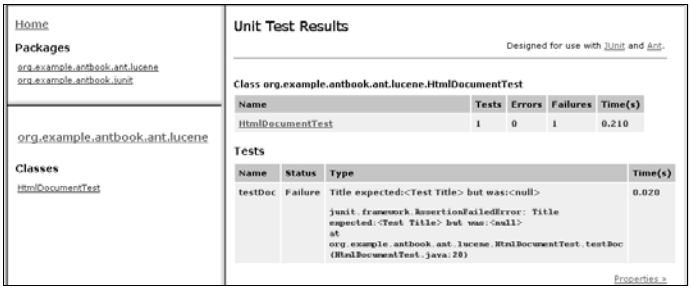

Figure 4.6
Test case results. The specific assertion that failed is clearly shown.

Clicking the Properties » hyperlink pops up a window displaying all of Ant's properties at the time the tests were run, which can be handy for troubleshooting failures caused by environmental or configuration issues.

NOTE There are a couple of issues with `<junit>` and `<junitreport>`. First, `<junit>` does not have any dependency checking logic; it always runs all tests. Second, `<junitreport>` simply aggregates XML files without any knowledge of whether the files it is using have any relation to the tests just run. A technique using `<uptodate>` takes care of ensuring tests only run if things have changed. Cleaning up the old test results before running tests gives you better reports.

Requirements of <junitreport>

The `<junitreport>` task requires an XSLT processor to do its thing. We recommend Xalan 2.x. You can obtain Xalan from http://xml.apache.org/xalan-j/. As with other dependencies, place xalan.jar into ANT_HOME/lib.

4.7.1 Generate reports and allow test failures to fail the build

We run into a dilemma with `<junitreport>` though. We've instructed `<junit>` to halt the build when a test fails. If the build fails, Ant won't create the reports. The last thing we want to do is have our build succeed when tests fail, but we must turn off `haltonfailure` in order for the reports to generate. As a solution, we make the `<junit>` task set specified properties upon a test failure or error, using the `failureProperty` and `errorProperty` attributes respectively.

Using the properties set by `<junit>`, we can generate the reports before we fail the build. Here is how this works:

```
<target name="test" depends="test-compile">
  <junit printsummary="false"
         errorProperty="test.failed"
         failureProperty="test.failed">    <-- haltonfailure has been removed
    <classpath refid="test.classpath"/>
    <formatter type="brief" usefile="false"/>
    <formatter type="xml"/>
    <batchtest todir="${test.data.dir}">
      <fileset dir="${test.dir}" includes="**/*Test.class"/>
    </batchtest>
  </junit>

  <junitreport todir="${test.data.dir}">
    <fileset dir="${test.data.dir}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames"
      todir="${test.reports.dir}"/>
  </junitreport>

  <fail message="Tests failed. Check log and/or reports."
        if="test.failed"/>    <-- Conditional <fail> task-based
</target>
```

NOTE *Remember that properties are immutable.* Use a unique previously undefined property name for `failureProperty` and `errorProperty`. (Both may be the same property name.) As for immutability, here is one of the holes in its rules. The value of these properties will be overwritten if an error or failure occurs with the value `true`. See chapter 3 for more information on properties.

Customizing the JUnit reports

If the default HTML generated by `<junitreport>` does not suit your needs, the output can be customized easily using different XSL files. The XSL files used by the task are embedded in Ant's `optional.jar`, and ship in the etc directory of the installation for customization use. To customize, either copy the existing `junit-frames.xsl` and `junit-noframes.xsl` files to another directory or create new

ones—you do need to use these exact file names. To use your custom XSL files, simply point the `styledir` attribute of the `<report>` element at them. Here we have a property `junit.style.dir` that is set to the directory where the XSL files exist:

```
<junitreport todir="${test.data.dir}">
  <fileset dir="${test.data.dir}">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames"
      styledir="${junit.style.dir}"
      todir="${test.reports.dir}"/>
</junitreport>
```

4.7.2 Run a single test case from the command-line

Once your project has a sufficiently large number of test cases, you may need to isolate a single test case to run when ironing out a particular issue. This feat can be accomplished using the `if`/`unless` clauses on `<test>` and `<batchtest>`. Our `<junit>` task evolves again:

```
<junit printsummary="false"
       errorProperty="test.failed"
       failureProperty="test.failed">
  <classpath refid="test.classpath"/>
  <formatter type="brief" usefile="false"/>
  <formatter type="xml"/>
  <test name="${testcase}" todir="${test.data.dir}" if="testcase"/>
  <batchtest todir="${test.data.dir}" unless="testcase">
    <fileset dir="${test.dir}" includes="**/*Test.class"/>
  </batchtest>
</junit>
```

By default, `testcase` will not be defined, the `<test>` will be ignored, and `<batchtest>` will execute all of the test cases. In order to run a single test case, run Ant using a command line like

```
ant test -Dtestcase=<fully qualified classname>
```

4.7.3 Initializing the test environment

There are a few steps typically required before running `<junit>`:

- Create the directories where the test cases will compile to, results data will be gathered, and reports will be generated.
- Place any external resources used by tests into the classpath.
- Clear out previously generated data files and reports.

Because of the nature of the `<junit>` task, old data files should be removed prior to running the tests. If a test case is renamed or removed, its results may still be present. The `<junit>` task simply generates results from the tests being run and does not concern itself with previously generated data files.

Our `test-init` target is defined as:

```
<target name="test-init">
  <mkdir dir="${test.dir}"/>

  <delete dir="${test.data.dir}"/>
  <delete dir="${test.reports.dir}"/>
  <mkdir dir="${test.data.dir}"/>
  <mkdir dir="${test.reports.dir}"/>
</target>
```

4.7.4 Other test issues

Forking

The `<junit>` task, by default, runs within Ant's JVM. There could be VM conflicts, such as static variables remaining defined, so the attribute `fork="true"` can be added to run in a separate JVM. The `fork` attribute applies to the `<junit>` level affecting all test cases, and it also applies to `<test>` and `<batchtest>`, overriding the fork setting of `<junit>`. Forking unit tests can enable the following (among others):

- Use a different JVM than the one used to run Ant (`jvm` attribute)
- Set timeout limitations to prevent tests from running too long (`timeout` attribute)
- Resolve conflicts with different versions of classes loaded by Ant than needed by test cases
- Test different instantiations of a singleton or other situations where an object may remain in memory and adversely affect clean testing

Forking tests into a separate JVM presents some issues as well, because the classes needed by the formatters and the test cases themselves must be in the classpath. The nested classpath will likely need to be adjusted to account for this:

```
<classpath>
  <path refid="test.classpath"/>
  <pathelement path="${java.class.path}"/>
</classpath>
```

The JVM provided property `java.class.path` is handy to make sure the spawned process includes the same classpath used by the original Ant JVM.

Configuring test cases dynamically

Test cases ideally are stateless and can work without any external information, but this is not always realistic. Tests may require the creation of temporary files or some external information in order to configure themselves properly. For example, the test case for our custom Ant task, `IndexTask`, requires a directory of documents to index and a location to place the generated index. The details of this task and its test case are not covered here, but how those parameters are passed to our test case is relevant.

The nested <sysproperty> element of <junit> provides a system property to the executing test cases, the equivalent of a -D argument to a Java command-line program:

```
<junit printsummary="false"
       errorProperty="test.failed"
       failureProperty="test.failed">
  <classpath refid="test.classpath"/>
  <sysproperty key="docs.dir" value="${test.dir}/org"/>
  <sysproperty key="index.dir" value="${test.dir}/index"/>
  <formatter type="xml"/>
  <formatter type="brief" usefile="false"/>
  <test name="${testcase}" if="testcase"/>
  <batchtest todir="${test.data.dir}" unless="testcase">
    <fileset dir="${test.dir}" includes="**/*Test.class"/>
  </batchtest>
</junit>
```

The docs.dir property refers to the org subdirectory so that only the non-.java files copied from our source tree to our build tree during test-init are seen by IndexTask. Remember that our test reports are also generated under test.dir, and having those in the mix during testing adds unknowns to our test case. Our IndexTaskTest obtains these values using System.getProperty:

```
private String docsDir = System.getProperty("docs.dir");
private String indexDir = System.getProperty("index.dir");
```

Testing database-related code and other dynamic information

When crafting test cases, it is important to design tests that verify expected results against actual results. Code that pulls information from a database or other dynamic sources can be troublesome because the expected results vary depending on the state of things outside our test cases' control. Using mock objects is one way to test database-dependent code. Refactoring is useful to isolate external dependencies to their own layer so that you can test business logic independently of database access, for example.

Ant's <sql> task can preconfigure a database with known test data prior to running unit tests. The DBUnit framework (http://dbunit.sourceforge.net/) is also a handy way to ensure known database state for test cases.

4.8 SHORT-CIRCUITING TESTS

The ultimate build goal is to have unit tests run as often as possible. Yet running tests takes time—time that developers need to spend developing. The <junit> task performs no dependency checking; it runs all specified tests each time the task is encountered. A common practice is to have a distribution target that does not depend on the testing target. This enables quick distribution builds and maintains a separate target that performs tests. There is certainly merit to this approach, but here is an alternative.

In order for us to have run our tests and have build speed too, we need to perform our own dependency checking. First, we must determine the situations where we can skip tests. If all of the following conditions are true, then we can consider skipping the tests:

- Production code is up-to-date.
- Test code is up-to-date.
- Data files used during testing are up-to-date.
- Test results are up-to-date with the test case classes.

Unfortunately, these checks are not enough. If tests failed in one build, the next build would skip the tests since all the code, results, and data files would be up-to-date; a flag will be set if a previous build's tests fail, allowing that to be taken into consideration for the next build. In addition, since we employ the single-test case technique shown in section 4.7.2, we will force this test to run if specifically requested.

Using `<uptodate>`, clever use of mappers, and conditional targets, we will achieve the desired results. Listing 4.1 shows the extensive `<condition>` we use to accomplish these up-to-date checks.

Listing 4.1 Conditions to ensure unit tests are only run when needed

```
<condition property="tests.uptodate">
  <and>
    <uptodate>                                                          (1)
      <srcfiles dir="${src.dir}" includes="**/*.java"/>
      <mapper type="glob"
              from="*.java"
              to="${build.classes.dir}/*.class" />
    </uptodate>

    <uptodate>                                                          (2)
      <srcfiles dir="${test.src.dir}" includes="**/*.java"/>
      <mapper type="glob"
              from="*.java"
              to="${test.classes.dir}/*.class" />
    </uptodate>

    <uptodate>                                                          (3)
      <srcfiles dir="${test.src.dir}" excludes="**/*.java"/>
      <mapper type="glob"
              from="*"
              to="${test.classes.dir}/*" />
    </uptodate>

    <not>                                                               (4)
      <available file="${test.last.failed.file}"/>
    </not>

    <not>                                                               (5)
      <isset property="testcase"/>
    </not>
```

```
      <uptodate>
        <srcfiles dir="${test.src.dir}" includes="**/*.java"/>
        <mapper type="package"[4]
                from="*Test.java"
                to="${test.data.dir}/TEST-*Test.xml"/>
      </uptodate>
    </and>
  </condition>
```

❻

Let's step back and explain what is going on in this `<condition>` in detail.

❶ Has production code changed? This expression evaluates to true if production class files in `${build.classes.dir}` have later dates than the corresponding .java files in `${src.dir}`.

❷ Has test code changed? This expression is equivalent to the first, except that it's comparing that our test classes are newer than the test .java files.

❸ Has test data changed? Our tests rely on HTML files to parse and index. We maintain these files alongside our testing code and copy them to the test classpath. This expression ensures that the data files in our classpath are current with respect to the corresponding files in our test source tree.

❹ Did last build fail? We use a temporary marker file to flag if tests ran but failed. If the tests succeed, the marker file is removed. This technique is shown next.

❺ Single test case run? If the user is running the build with the `testcase` property set we want to always run the test target even if everything is up to date. The conditions on `<test>` and `<batchtest>` in our "test" target ensure that we only run the one test case requested.

❻ Test results current? The final check compares the test cases to their corresponding XML data files generated by the "xml" `<formatter>`.

Our test target, incorporating the last build test failure flag, is now

```
<property name="test.last.failed.file"
          location="${build.dir}/.lasttestsfailed"/>

<target name="test" depends="test-compile"
        unless="tests.uptodate">

  <junit printsummary="false"
         errorProperty="test.failed"
         failureProperty="test.failed"
         fork="${junit.fork}">
    <!-- . . . -->
  </junit>
```

[4] The `package` mapper was conceived and implemented by Erik while writing this chapter.

```
  <junitreport todir="${test.data.dir}">
    <!-- . . . -->
  </junitreport>

  <echo message="last build failed tests"
        file="${test.last.failed.file}"/>
  <fail if="test.failed">
    Unit tests failed.  Check log or reports for details
  </fail>

  <!-- Remove test failed file, as these tests succeeded -->
  <delete file="${test.last.failed.file}"/>
</target>
```

The marker file `${build.dir}/.lasttestsfailed` is created using `<echo>`'s file creation capability and then removed if it makes it past the `<fail>`, indicating that all tests succeeded.

While the use of this long `<condition>` may seem extreme, it accomplishes an important goal: tests integrated directly in the dependency graph won't run if everything is up-to-date.

Even with such an elaborate up-to-date check to avoid running unit tests, some conditions are still not considered. What if the build file itself is modified, perhaps adjusting the unit test parameters? What if an external resource, such as a database, changes? As you can see, it's a complex problem and one that is best solved by deciding which factors are important to your builds. Such complexity also reinforces the importance of doing regular clean builds to ensure that you're always building and testing fully against the most current source code.

This type of up-to-date checking technique is useful in multiple component/build-file environments. In a single build-file environment, if the build is being run then chances are that something in that environment has changed and unit tests should be run. Our build files should be crafted so that they play nicely as subcomponent builds in a larger system though, and this is where the savings become apparent. A master build file delegates builds of subcomponents to subcomponent-specific build files. If every subcomponent build runs unit tests even when everything is up-to-date, then our build time increases dramatically. The `<condition>` example shown here is an example of the likely dependencies and solutions available, but we concede that it is not simple, foolproof, or necessary. Your mileage is likely to vary.

4.8.1 Dealing with large number of tests

This technique goes a long way in improving build efficiency and making it even more pleasant to keep tests running as part of every build. In larger systems, the number of unit tests is substantial, and even the slightest change to a single unit test will still cause the entire batch to be run. While it is a great feeling to know there are a large number of unit tests keeping the system running cleanly, it can also be a build burden. Tests must run quickly if developers are to run them every build. There is no single solution for this situation, but here are some techniques that can be utilized:

- You can use conditional patternset includes and excludes. Ant properties can be used to turn off tests that are not directly relevant to a developer's work.
- Developers could construct their own JUnit `TestSuite` (perhaps exercising each particular subsystem), compiling just the test cases of interest and use the single test case method.

4.9 BEST PRACTICES

This chapter has shown that writing test cases is important. Ant makes unit testing simple by running them, capturing the results, and failing a build if a test fails. Ant's datatypes and properties allow the classpath to be tightly controlled, directory mappings to be overridden, and test cases to be easily isolated and run individually. This leaves one hard problem: designing realistic tests.

We recommend the following practices:

- Test *everything* that could possibly break. This is an XP maxim and it holds.
- A well-written test is hard to pass. If all your tests pass the first time, you are probably not testing vigorously enough.
- Add a new test case for every bug you find.
- When a test case fails, track down the problem by writing more tests, before going to the debugger. The more tests you have, the better.
- Test invalid parameters to every method, rather than just valid data. Robust software needs to recognize and handle invalid data, and the tests that pass using incorrect data are often the most informative.
- Clear previous test results before running new tests; delete and recreate the test results and reports directories.
- Set `haltonfailure="false"` on `<junit>` to allow reporting or other steps to occur before the build fails. Capture the failure/error status in a single Ant property using `errorProperty` and `failureProperty`.
- Pick a unique naming convention for test cases: *Test.java. Then you can use `<batchtest>` with Ant's pattern matching facility to run only the files that match the naming convention. This helps you avoid attempting to run helper or base classes.
- Separate test code from production code. Give them each their own unique directory tree with the same package naming structure. This lets tests live in the same package as the objects they test, while still keeping them separate during a build.
- Capture results using the XML formatter: `<formatter type="xml"/>`.
- Use `<junitreport>`, which generates fantastic color enhanced reports to quickly access detailed failure information.
- Fail the build if an error or failure occurred: `<fail if="test.failed"/>`.

- Use informative names for tests. It is better to know that `testDocumentLoad` failed, rather than `test17` failed, especially when the test suddenly breaks four months after someone in the team wrote it.
- Try to test only one thing per test method. If `testDocumentLoad` fails and this test method contains only one possible point of failure, it is easier to track down the bug than to try and find out which one line out of twenty the failure occurred on.
- Utilize the testing up-to-date technique shown in section 4.8. Design builds to work as subcomponents, and be sensitive to build inefficiencies doing unnecessary work.

Writing test cases changes how we implement the code we're trying to test, perhaps by refactoring our methods to be more easily isolated. This often leads to developing software that plays well with other modules because it is designed to work with the test case. This is effective particularly with database and container dependencies because it forces us to decouple core business logic from that of a database, a web container, or other frameworks. Writing test cases may actually improve the design of our production code. In particular, if you cannot write a test case for a class, you have a serious problem, as it means you have written untestable code.

Hope is not lost if you are attempting to add testing to a large system that was built without unit tests in place. Do not attempt to retrofit test cases for the existing code in one big go. Before adding new code, write tests to validate the current behavior and verify that the new code does not break this behavior. When a bug is found, write a test case to identify it clearly, then fix the bug and watch the test pass. While some testing is better than no testing, a critical mass of tests needs to be in place to truly realize such XP benefits as fearless and confident refactoring. Keep at it and the tests will accumulate allowing the project to realize these and other benefits.

4.10 SUMMARY

Unit testing makes the world a better place because it gives us the knowledge of a change's impact and the confidence to refactor without fear of breaking code unknowingly. Here are some key points to keep in mind:

- JUnit is Java's de facto testing framework; it integrates tightly with Ant.
- `<junit>` runs tests cases, captures results, and can set a property if tests fail.
- Information can be passed from Ant to test cases via `<sysproperty>`.
- `<junitreport>` generates HTML test results reports, and allows for customization of the reports generated via XSLT.

Now that you've gotten Ant fundamentals down for compiling, using datatypes and properties, and testing, we move to executing Java and native programs from within Ant.