

3

The First Iteration

Ant can play an important role in testing. By integrating testing into the build and deployment process, it becomes easier to implement and enforce ease-of-use considerations and compliance with testing standards. Development teams will see a marked improvement in quality and can more easily stick to delivery schedules. Ant plays a key role in merging the testing process with the build process, to provide a seamless integration of the often-divergent processes.

IN THIS CHAPTER

- Benefits of Automated Testing
- Benefits of Integrating Testing into Ant
- Types of Automated Testing
- What Is Unit Testing?
- Test First Design
- JUnit
- Version-Control Systems
- CVS Access and Logins
- Basic Deployment

Story

Networks Byte Design, Inc., has embarked on a project to implement eMarket, their new e-commerce solution for sales and marketing. The core project team has been assembled and is developing the product using XP. So far, the eMarket project team has collected user stories written by the customer, decided on a high-level architecture, and implemented spikes to reduce the risks associated with the technical design decisions. Sandy, the project leader, schedules a *release planning* meeting between the development team and the customer to sort out the user stories in order to create a *release plan*.

At the meeting, the developers listen to the customer present each story. The development team divides into smaller groups and assigns a point value to each story to indicate its expected difficulty. During the meeting, some user stories are torn up and rewritten, some are discarded entirely, and a few new stories are added.

The development team returns the stories to the customer with their initial estimates and anticipated velocity. After seeing the estimates, the customer begins to define the priorities. The customer determines the order in which the stories should be implemented, based on the business value and time required to complete the story. The result of the meeting is that the first release plan is developed, which was created to get the stories with the highest business value completed first.

The customer has selected the stories that she wants to have completed in the first iteration, and puts them at the top of the stack of all the stories. The developers and customer go over the user stories, and the development team discusses the technical aspects of the stories and the tasks that need to be completed. Finally, the developers sign up for tasks for this iteration.

Sandy is concerned with the customer's uncertainty about requirements related to workflow and usability. She directs the team to focus on the user interface so that the customer can have an early look at the proposed design and workflow.

Michael, the lead developer, decides that for the first iteration, they will use dummy business objects that have their interfaces developed as needed, but will just return hard-coded data for the time being. The real implementation of the business objects will occur in later iterations.

Having completed the first release plan and iteration plan, the group begins work on the first iteration, which is to begin implementing the Web interface and mocking up the business objects. Each pair of developers will write a unit test that tests the functionality that they are about to implement. That is followed by the required class implementation. As new functionality is added to each class, the unit test is first modified before adding the new functionality to the class. The developers must run their unit tests each time they build their code. To do this efficiently, they need to modify the buildfile to handle unit testing. JUnit, a free open-source tool that integrates easily with Ant, is selected as the team's unit-testing tool.

The expense of fixing a bug also generally increases as you move along the development cycle. A bug caught during development is less expensive to fix than a bug caught during testing. The cost rises dramatically when a bug must be fixed in a product that has already shipped. The sooner bugs can be detected and fixed, the less money they cost to fix.

Testing is an important aspect of every development project. Unless the software can meet the minimal standards for usability, reliability, performance, requirements, and overall quality, it might never make it to a production system. All the work spent in planning, design, and coding will go to waste if the intended audience never uses the system.

With all the emphasis on testing, it would seem that the various forms of testing would be a major aspect of most software-development projects. Unfortunately, many projects start out with good intentions but rarely follow through with those plans. Most people agree on the importance that testing should play in the project, but testing often is conducted toward the end of the project, when the schedule for testing becomes severely compacted, sometimes to the point of becoming ineffective.

To attempt to combat this trend, newer development methodologies, such as XP, emphasize testing early and often in the process. Many of the techniques for testing in these methodologies are not new; they are simply changes in the frequency and schedule of testing procedures. Emphasis is placed on unit testing, in which developers test the smallest components of the system. These new concepts are not miracle cures or silver-bullet solutions, but they can help ensure that the software being developed has been designed and coded with testing in mind.

Benefits of Automated Testing

Automated testing brings a number of benefits. For one, the tests are repeatable. When a test is created, it can be run each time the testing process is launched. Automating testing reduces the fatigue of performing testing manually, which leads

to more consistent results. Also, because the tests are automated, they're easy to run, which means that they will be run more often. As new bugs are discovered and fixed, tests can be added to check for those bugs, to ensure that they aren't reintroduced. This increases the overall completeness of testing.

Automating the testing process can be as beneficial as automating the build process. The testing process is based on the concept of being repeatable, which requires an automated or, at the very least, well-documented process. Some of the benefits of automated testing are that it

- Is a repeatable process
- Uses a consistent approach
- Follows a documented process
- Frees up developer-hours for more profitable tasks
- Is expandable and flexible, with changes in code propagated to the testing procedure faster and more efficiently
- Negates the fatigue factor as development deadlines approach because automated tests will eliminate the stress and workload of manual testing on developers

Some drawbacks are worth mentioning, of course. Some features don't easily lend themselves to automated testing. For example, sometimes automation-testing software can be used to test complex GUI applications, but often these applications must be tested by hand.

Automated testing is not a panacea for all problems, but it can contribute to an efficient and effective software development process. Integrating a testing tool into Ant that wasn't designed to be executed from Ant can require additional work. This can be accomplished by extending Ant, using BSF scripting, or using the `<exec>` task to launch another tool. If the test will be run frequently, the effort is worth the benefits gained in ease of testing.

Because it's a repeatable process, automated testing achieves an important part of the testing process by making it possible to conduct regression testing, to retest the same scenario again. How many bugs reported by testing teams cannot be duplicated by the developers? How many bugs are fixed, yet the tests that are run to check the fixes are insufficient or different from the original tests? These are the types of issues that regression testing helps address, and this is why the benefit of repeatable tests is so high.

Consistency issues are easiest to observe in teams with multiple testers and developers, but even a single tester would rarely conduct the same tests the same way each time. Automating the process maintains consistency from one run of the test to the next, regardless of how much time passes between the two runs of the tests or who is executing the tests.

The best kind of documentation is documentation that does not have to be written and yet is guaranteed to be correct. In a nutshell, that is a description of a self-documenting system. The goal is to create readable code. When the programmer clearly defines the testing goals for the test, someone who comes along later can easily understand the purpose of the test. This documentation does not have to be written; it is a beneficial side effect. The code is guaranteed to be correct because the tests have been executed under certain conditions and passed. Basically, tests should be as self-documenting as possible. Most developers don't like to comment the code that's going into the product. They're even less likely to comment the unit tests that accompany the product code because they know that the unit-test code won't go into the product. By developing readable code, the need to heavily comment the unit tests is greatly reduced.

By automating the testing process, the computer will usually execute the testing process in less time than it takes a tester to perform manually. Although the code will take the same time to execute, the prep time and interpretation of the results will be quicker with the automated process. Also, because this is a repeatable test, the automated approach becomes even more beneficial when the tests must be run multiple times. People get tired of conducting the same tests repeatedly, and will make mistakes, but the automated processes run consistently each time. Again, manual testing has its place; the advantage of automated testing is that it can easily catch many of the problems before manual testing even begins.

Benefits of Integrating Testing into Ant

Beyond the benefits of automating the testing process in general, integrating the testing process into the Ant build process has additional benefits:

- The integrated testing process is likely to be used.
- Integrated testing stresses the importance of testing early and often.
- Enforcement of testing standards is easier.
- Testing is consistent over every version of the software project.

By including the testing process inside the Ant build process, the likelihood that the process will be used increases enormously. If the process is as simple as performing a task that the developers do many times a day, many of the typical excuses for skipping the process are eliminated. If a process precludes the developers from executing it, the benefits of the process can never be realized.

Another benefit of placing the testing process inside Ant is that it enforces the idea that testing is part of the development process, not just something to be done by a testing group prior to shipping. Testing is conducted at many levels to test various aspects of the application. By including testing throughout the process, bugs can be found earlier, fewer mistakes are repeated, and problems are found closer to the time when they are introduced.

Finally, including testing in the Ant process provides for easier enforcement of the testing standards. Ant targets can be developed to ensure that all classes have a unit test created for them. We will look at techniques for accomplishing this in Chapter 9, “Company Reorganized—Working with a New Subteam,” and Chapter 10, “Additional Teams Adopt the XP Process.” The Ant build process can enforce the stipulation that tests must be run at certain points, such as before deployment. Ant also can enforce that all tests must pass before moving on to subsequent steps. By allowing Ant to enforce these standards, the development team is freed up to focus on developing production-quality code.

Types of Automated Testing

A variety of different types of automated testing can be conducted, and Ant can be used to integrate most, if not all, of them. These tests can be used to check the quality of every aspect of the project. Some of the types of testing that can be automated as part of the Ant build process include

- **Unit**—Verifies that certain parts of the code are working properly. It is not an overall system test.
- **Functional**—Used to test the usability of the system. This type of testing is also known as “black box testing” because the testing is conducted with no knowledge of how the code was implemented.
- **Acceptance**—Used to determine if all of the requirements have been met. These are the requirements defined by the customer in the form of user stories.
- **Regression**—Verifies that previously working parts of the system are still working after changes have been made.
- **Performance**—Verifies that the components of the systems meet the stated requirements for speed.
- **Load**—Stresses a system to test how it will behave under heavy use. A system that works well with one user may completely break down with several hundred or more users. Load testing helps to evaluate how well a system holds up under heavy use.

If your project is not using any form of automated testing currently, we highly recommend it. A great sense of accomplishment comes when a job has been done well. Testing allows success to be quantified and measured. By automating the test, you have the added benefit of being able to easily repeat the test.

In this chapter, we focus on *unit testing*. Unit testing is an important part of the XP process. Of all the types of automated testing, this is the one that should get the most use because developers should be running these tests at least daily, if not more frequently. Developers should also be writing new tests for all new code that they

write and all changes that they make. It has been our experience that this also is an area of testing that many teams overlook or do not adequately implement. An organization might have experience with integration or performance testing, but unit testing often falls by the wayside.

What Is Unit Testing?

A *unit* is defined as the smallest piece of functionality that can stand alone. In C++ and Java applications, a unit generally is seen as equivalent to a class. If the class has been well designed and deals with a distinct object, the one class equals one unit test is a good rule of thumb. Unit tests are one of the many concepts in Ant in which consistency in conventions is important and can reduce the effort required to automate the process.

As an example, let's consider a simple class with a method that adds an object to the end of a list. Without unit testing, you probably would have performed some basic testing to make sure that the object was added to the end of the list, not to the beginning or somewhere in the middle. Unit testing provides the confidence that the method will do the proper thing, no matter what the input is. For example, consider what the sample method would do if the inputs were as follows:

- The object is null.
- The list is empty.
- The list is null.
- The list has exactly one item.

These are the types of tests that often are not conducted. At some point, a section of code that calls the sample class might be modified so that the list can be empty. The application no longer operates correctly, but the bug is more difficult to locate because what appeared to be working code that has not been changed is now broken. Rightly so, the focus is placed on the new code that was written rather than the code where the bug actually exists. By ensuring that the code can properly deal with all types of inputs, this method can be used in a variety of circumstances. It will be unlikely that the code is hiding a dormant bug, which always seems to appear at the worst possible time.

Test First Design

Unit testing has become such an important part of the development process that it is a central focal point of XP. In XP, you actually write your unit tests first, and then write the code. This is known as Test First Design.

Test First Design is a required practice in XP. The premise of Test First Design is that quality software can be designed and implemented by using the following iterative process. First, test for a condition that we want our software to fulfill. Initially, this test will fail. Second, write the code necessary to make the test pass. Finally, the code

should be refactored periodically. Code can be refactored with confidence because unit tests are in place to detect any problems we may create, and to let us know when we have working code. This process is performed in a series of short iterations that will lead to an effective design. This also helps the developer to stay focused on implementing only what is needed.

Without the benefits that unit testing provides, development methodologies such as XP would not be possible. Even if your team is using a more traditional development methodology, unit testing can provide the same level of benefits and should be regarded as a critical part of all software development.

NOTE

In XP, unit tests are written first, and then the code they are designed to test follows. This is known as Test First Design. Writing tests first places the emphasis on how the class will be used, and therefore on coding only what is needed. This is also a good strategy for fixing bugs because it catches the presence of the bug immediately. Then write the code to pass the unit test. This way, you'll add to the overall suite of unit tests that are run, and you will catch the bug if it's ever reintroduced. You'll also ensure that your unit test works correctly.

JUnit

Although many techniques can be used to implement unit testing, the most popular tool for Java development is JUnit. This tool provides a framework for developing unit tests, which fits very nicely into the Ant build and deployment process. You can find JUnit, along with a lot of documentation and add-on tools, at www.junit.org.

INSTALLATION OF JUNIT

To install JUnit, follow these steps:

1. Download JUnit from www.junit.org
 2. Add the `.junit.jar` file to your `CLASSPATH` or to the Ant `lib` directory. Because JUnit is an optional task, the `optional.jar` file that ships with the Ant distribution must be in the Ant `lib` directory as well.
-

Sample Class

Let's take a look at how to incorporate JUnit into your development process and seamlessly integrate it into the Ant buildfile. Listing 3.1 is a sample Java class for which you will create a JUnit unit test. This is a simple class with a constructor, along with setter and getter methods. This class is a simple domain object to store the information about a single sales item in the eMarket application.

LISTING 3.1 Sample Class for Use with JUnit Unit Test

```
/*.....  
File: salesItem  
.....*/  
  
package com.networksByteDesign.eMarket.inventory;  
  
public class salesItem  
{  
    /* =====  
    salesItem Constructor  
    ===== */  
  
    public salesItem(int id, String name)  
    {  
        mId        = id;  
        mName       = name;  
    }  
  
    /* =====  
    getId  
    ===== */  
  
    public int getId()  
    {  
        return mId;  
    }  
  
    /* =====  
    setId  
    ===== */  
  
    public void setId(int id)  
    {  
        if(id <= 0)  
        {  
            throw new IllegalArgumentException("Id must be a valid id #");  
        }  
  
        mId = id;  
    }  
}
```

LISTING 3.1 Continued

```
/* =====
   getName
   ===== */

public String getName()
{
    return mName;
}

/* =====
   setName
   ===== */

public void setName(String name)
{
    if(name == null || name.length() == 0)
    {
        throw new IllegalArgumentException("Name must be populated");
    }

    mName = name;
}

private int    mId    = 0;
private String mName = null;
}
```

Sample Unit Test

Let's create a unit test to demonstrate how JUnit hooks into Ant. Listing 3.2 is the JUnit test that was written to test the sample class shown in Listing 3.1. We have included tests for both the constructor and the setter method. The getter method is tested as part of the other two tests. This JUnit class also includes a `main()` method for running the unit test from the command line.

LISTING 3.2 JUnit Test for Sample Class in Listing 3.1

```
/*.....
File: salesItemTest
.....*/
```

LISTING 3.2 Continued

```

package com.networksByteDesign.eMarket.inventory;

// Internal libraries
import com.networksByteDesign.eMarket.inventory.salesItem;

// Third party libraries
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class salesItemTest extends TestCase
{
    ///////////////////////////////////////////////////////////////////
    // salesItemTest(String)
    ///////////////////////////////////////////////////////////////////

    /**
     * <p>
     * This is the constructor for the <code>salesItemTest</code>
     * class. It calls the super class and configures the instance.
     * </p>
     *
     * @param testName the name of the test to construct
     *
     * */

    public salesItemTest(String testName)
    {
        super(testName);
    }

    ///////////////////////////////////////////////////////////////////
    // main(String[])
    ///////////////////////////////////////////////////////////////////

    /**
     * <p>
     * This is the mainline for the <code>salesItemTest</code>
     * class. It runs the test suite that has been established.

```

LISTING 3.2 Continued

```
* </p>
*
* @param args any command line arguments to the test program
*
* */

public static void main (String[] args)
{
    junit.textui.TestRunner.run(suite());
}

////////////////////////////////////
// suite()
////////////////////////////////////

/**
 * <p>
 * This is the static method that defines the specific tests that
 * comprise the unittest.
 * </p>
 *
 * @return the test suite that has been established
 *
 * */

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

    // test constructor()
    suite.addTest(new salesItemTest("testConstructor"));
    suite.addTest(new salesItemTest("testSetter"));

    return suite;
}

////////////////////////////////////
// testConstructor()
////////////////////////////////////
```

LISTING 3.2 Continued

```
/**
 * <p>
 * Test for constructing a salesItem object
 * </p>
 *
 * */

public void testConstructor()
{
    int    id        = 123;
    String name     = "Router";

    // Does "happy path" work?
    salesItem test1 = new salesItem(id, name);
    assertEquals("Happy Path id test failed", id, test1.getId());
    assertEquals("Happy Path name test failed", name, test1.getName());

    // Is negative id handled?
    try
    {
        salesItem test2 = new salesItem(-123, name);
        fail("Expected exception was not thrown");
    }
    catch(IllegalArgumentException e) {}

    // Is zero id handled?
    try
    {
        salesItem test3 = new salesItem(0, name);
        fail("Expected exception was not thrown");
    }
    catch(IllegalArgumentException e) {}

    // Is empty string handled?
    try
    {
        salesItem test4 = new salesItem(id, "");
        fail("Expected exception was not thrown");
    }
    catch(IllegalArgumentException e) {}
}
```

LISTING 3.2 Continued

```
    // Is null string handled?
    try
    {
        salesItem test5 = new salesItem(id, null);
        fail("Expected exception was not thrown");
    }
    catch(IllegalArgumentException e) {}
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// testSetter()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/**
 * <p>
 * Test for setter for the salesItem object
 * </p>
 *
 * */

public void testSetter()
{
    int      id   = 123;
    String   name = "Router";
    salesItem test = new salesItem(456, "Another");

    // Does "happy path" work?
    test.setId(id);
    assertEquals("Happy Path id test failed", id, test.getId());

    test.setName(name);
    assertEquals("Happy Path name test failed", name, test.getName());

    // Is negative id handled?
    try
    {
        test.setId(-123);
        fail("Expected exception was not thrown");
    }
    catch(IllegalArgumentException e) {}
}
```

LISTING 3.2 Continued

```
// Is zero id handled?
try
{
    test.setId(0);
    fail("Expected exception was not thrown");
}
catch(IllegalArgumentException e) {}

// Is empty string handled?
try
{
    test.setName("");
    fail("Expected exception was not thrown");
}
catch(IllegalArgumentException e) {}

// Is null string handled?
try
{
    test.setName(null);
    fail("Expected exception was not thrown");
}
catch(IllegalArgumentException e) {}
}
}
```

The unit test has two test methods: `testConstructor()` and `testSetter()`. These two tests were added to the test suite in the `suite()` method. When each test method is called, JUnit will call a `setup()` method if there is one. The optional `setup()` method can be used to perform one-time activities needed for the test such as instantiating a certain class, or making a database connection. Next the `testConstructor()` method is called. Within this test method, several tests are run to verify correct behavior of the tested class under all different conditions. Correct behavior is checked with various `assertXXX()` methods, such as `assertTrue()`. The `assertXXX()` methods can be used to check expected values and fail the test if the expected values aren't received. In our example of a unit test, the setter methods should throw an `IllegalArgumentException` if the parameter is invalid. If the exception is not thrown, we call the `fail()` method to indicate that the test has not performed as expected and has failed. Finally, if we had implemented the optional `teardown()` method, it would be called after the test method completed. `teardown()` is used to clean up after a test, such as closing a database connection. The cycle

repeats for the next unit test. In the JUnit test class, the `suite()` method is used to add unit test methods to the suite of tests to be run. There are two ways to add tests. The first way is shown here:

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

    // test constructor()
    suite.addTest(new salesItemTest("testConstructor"));
    suite.addTest(new salesItemTest("testSetter"));

    return suite;
}
```

In this approach, each unit test is explicitly added in the `suite()` method. With this approach, each new unit test must be added as it is created. The other technique is to make use of the fact that JUnit uses Java reflection. If we name all of our unit test methods starting with “test*”, we can add all of the tests with one statement, as shown here:

```
public static Test suite()
{
    TestSuite suite = new TestSuite(salesItemTest.class);
    return suite;
}
```

The advantage to the second technique is that you don’t have to add each new unit-test method into the `suite()` as it is created. With the first approach, you have more control over which tests are run, in case you want to temporarily turn some of them off while debugging a problem.

Command-Line Unit Testing

Before we hook JUnit into Ant, let’s begin by running the unit test interactively at the command line. We first need to compile both the sample class and the unit test, using the `<compile>` target. In order to run the unit test at the command line, `.junit.jar` must be in the `CLASSPATH`. Listing 3.3 shows the output of running the unit test at the command line.

LISTING 3.3 Output of Running Command-Line JUnit Test

```
% java com.networksByteDesign.eMarket.inventory.salesItemTest
.F.
Time: 0.033
```

LISTING 3.3 Continued

There was 1 failure:

```
1) testConstructor(com.networksByteDesign.eMarket.inventory.salesItemTest)
   "Expected exception was not thrown"
```

FAILURES!!!

```
Tests run: 2, Failures: 1, Errors: 0
```

Simple JUnit Target

Of the two tests run, one failed. The constructor does not have the same level of checks as the setter, and it contains a bug if the object is constructed with a null or empty String. Before fixing this problem, let's hook the unit test into Ant, to see how failed tests are handled.

Listing 3.4 is a unittest target that simply calls the JUnit class shown in Listing 3.2. This is about as simple as a unit-testing target can be.

LISTING 3.4 Simple unittest Target

```
<?xml version="1.0" ?>
<project name="eMarket" default="compile" basedir=".">

  <property name="dirs.source" value="/usr/projects/eMarket/src" />
  <property name="dirs.backup" value="${user.home}/backup" />

  <!-- compile target -->
  <target name="compile" description="Compile all of the source code.">
    <javac srcdir="${dirs.source}" />
  </target>

  <!-- unittest target -->
  <target name="unittest"
    description="Run the unit tests for the source code.">
    <junit>
      <test
        name="com.networksByteDesign.eMarket.inventory.salesItemTest"
        />
    </junit>
  </target>
</project>
```

As you can see, this target simply calls the unit test just as we did from the command line. In Chapter 4, “The First Complete Build Process,” we will change this task to include sets of tests rather than listing each test individually. The output of this target appears in Listing 3.5.

LISTING 3.5 Output of Simple unittest Target with Broken Test

```
% ant unittest
Buildfile: build.xml

unittest:
    [junit] TEST
           com.networksByteDesign.eMarket.inventory.salesItemTest FAILED

BUILD SUCCESSFUL
Total time: 2 seconds
```

Although running the `unittest` target shows the test class that failed, the output does not tell which test within the class failed or contain other useful information. If we fix the class by having the constructor call the setter rather than setting the member variable directly, all the unit tests should pass. Listing 3.6 shows the output when all tests pass. As you can see, there is no output from the target.

LISTING 3.6 Output of Simple unittest Target with No Broken Tests

```
% ant unittest
Buildfile: build.xml

unittest:

BUILD SUCCESSFUL
Total time: 2 seconds
```

`haltonfailure/haltonerror`

Let's begin enhancing the `unittest` target by looking at what happens when you run multiple tests. JUnit has a concept of failures and errors. Failures are tests that do not pass, but in anticipated ways. For example, the sample unit test had a failure because we checked to make sure that a particular exception was thrown.

Errors are unanticipated problems. An exception that is thrown but not caught by the class or the test is a common occurrence of this. Errors are considered failures as well.

The JUnit task has two attributes for determining how Ant should behave if a failure or error occurs:

- `haltonfailure`
- `haltonerror`

If these attributes are set to `yes`, the build fails if a unit test experiences a failure or error.

Listing 3.7 shows a unit test run with `haltonfailure` set to `no`. Even though a test fails, the subsequent tests are still run.

LISTING 3.7 Output from Unit Test Run with `haltonfailure` Set to No

```
% ant unittest
Buildfile: build.xml

unittest:
    [junit] TEST com.networksByteDesign.eMarket.inventory.salesItemTest FAILED
    [junit] TEST com.networksByteDesign.eMarket.inventory.customerTest FAILED

BUILD SUCCESSFUL
Total time: 1 second
```

Listing 3.8 shows the same test run, but with `haltonfailure` set to `yes`. Notice that when the first test fails, the build ends. No further tests are run, and no opportunity exists to clean up after the test.

LISTING 3.8 Output from Unit Test Run with `haltonfailure` Set to Yes

```
% ant unittest
Buildfile: build.xml

unittest:

BUILD FAILED
file:/usr/projects/eMarket/build.xml:15:
    Test com.networksByteDesign.eMarket.inventory.salesItemTest failed

Total time: 1 second
```

Sometimes you want the build to fail, but not until you have cleaned up after the unit test. In this case, you can use the attributes `failureproperty` and `errorproperty`. If you supply a property name to these attributes, the property will be set automatically if

a failure or error occurs. Your Ant target can check these properties after cleaning up, to determine whether the build should be halted. Listing 3.9 shows an example of how `failureproperty` can be used.

LISTING 3.9 Example of Using `failureproperty` to Clean Up After a Failed Test

```
<?xml version="1.0" ?>
<project name="eMarket" default="compile" basedir=".">

  <property name="dirs.source" value="/usr/projects/eMarket/src" />
  <property name="dirs.backup" value="${user.home}/backup" />
  <property name="dirs.temp" value="/tmp" />

  <!-- compile target -->
  <target name="compile" description="Compile all of the source code.">
    <javac srcdir="${dirs.source}" />
  </target>

  <!-- unittest target -->
  <target name="unittest"
    description="Run the unit tests for the source code.">
    <junit haltonfailure="no" failureproperty="unittestFailed">
      <test
        name="com.networksByteDesign.eMarket.inventory.salesItemTest"
      />
      <test
        name="com.networksByteDesign.eMarket.inventory.customerTest"
      />
    </junit>

    <antcall target="cleanupUnittest" />
    <fail if="unittestFailed" message="One or more unit tests failed." />
  </target>

  <!-- cleanupUnittest target -->
  <target name="cleanupUnittest">
    <delete>
      <fileset dir="${dirs.temp}">
        <include name="*${ant.project.name}.test" />
      </fileset>
    </delete>
  </target>
</project>
```

The `unittest` target makes use of the `<antcall>` task. The `<antcall>` task is used to invoke another target within the same buildfile. Doing this creates another instance of a project. All of the properties in the current project will be passed to the new project unless the `inheritAll` attribute is set to `false`. The nested `<param>` element can also be used with `<antcall>` to pass new property values.

The `<fail>` task is used to inform the Ant build process that the build should fail and provide an appropriate message. In this case, we use the `if` attribute of the `<fail>` task to only cause the build to fail if the `unittestFailed` property is set. This allows us to cause the build to fail, but provide appropriate cleanup code prior to the failure.

`printsummary`

By default, Ant displays only the tests that fail or have an error. Although this is often desirable, some feedback on how things are progressing can be helpful when the unit testing process takes a long time. Listing 3.10 shows the output of the `unittest` target when multiple tests are run. As you can see, two classes are shown in the `unittest` section of the output.

LISTING 3.10 Build Output with Multiple Unit-Test Classes

```
% ant unittest
Buildfile: build.xml

unittest:
  [junit] TEST
           com.networksByteDesign.eMarket.inventory.salesItemTest FAILED
  [junit] TEST
           com.networksByteDesign.eMarket.inventory.customerTest FAILED

cleanupUnittest:
  [delete] Deleting 2 files from /tmp

BUILD FAILED
file:/usr/projects/eMarket/build.xml:26: One or more unit tests failed.

Total time: 2 seconds
```

To show the output from the tests being run, whether the tests fail or not, use the `printsummary` attribute. When this attribute is set to `yes`, all tests are summarized in the build output. Listing 3.11 shows the output using the same classes as in Listing 3.10. A third class is now visible. This class was being tested before, but because the tests passed, it was not visible. Setting the `printsummary` attribute shows all tests regardless of whether they pass or fail.

LISTING 3.11 Build Output with printsummary Set to Yes

```
% ant unittest
Buildfile: build.xml

unittest:
  [junit] Running com.networksByteDesign.eMarket.inventory.salesItemTest
  [junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.03 sec
  [junit] TEST
           com.networksByteDesign.eMarket.inventory.salesItemTest FAILED
  [junit] Running com.networksByteDesign.eMarket.inventory.customerTest
  [junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.015 sec
  [junit] TEST
           com.networksByteDesign.eMarket.inventory.customerTest FAILED
  [junit] Running com.networksByteDesign.eMarket.inventory.companyTest
  [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.169 sec

cleanupUnittest:

BUILD FAILED
file:/usr/projects/eMarket/build.xml:31: One or more unit tests failed.

Total time: 2 seconds
```

Using the `printsummary` attribute, you see not only all the classes, but also the number of tests run, how many failures and errors occurred, and the time elapsed for each test. You might want to experiment with both approaches and see which style works best for your team.

showoutput

If your classes make use of logging, whether with a logging tool such as `log4j` or a simple `System.out.println()`, that information can be displayed by running the unit tests. By setting the `showoutput` attribute, any information written to `stdout` and `stderr` is displayed in the unit test's output. Listing 3.12 shows the sample class with a logging statement in each setter method.

LISTING 3.12 Sample Class with Logging Statement

```
/*.....
   File: salesItem
   .....*/

package com.networksByteDesign.eMarket.inventory;
```

LISTING 3.12 Continued

```
public class salesItem
{
    /* =====
       salesItem Constructor
       ===== */

    public salesItem(int id, String name)
    {
        mId      = id;
        mName    = name;
    }

    /* =====
       getId
       ===== */

    public int getId()
    {
        return mId;
    }

    /* =====
       setId
       ===== */

    public void setId(int id)
    {
        System.out.println("ID = " + id);

        if(id <= 0)
        {
            throw new IllegalArgumentException("Id must be a valid id #");
        }

        mId = id;
    }

    /* =====
       getName
       ===== */
}
```

LISTING 3.12 Continued

```
public String getName()
{
    return mName;
}

/* =====
   setName
   ===== */

public void setName(String name)
{
    System.out.println("Name = " + name);

    if(name == null || name.length() == 0)
    {
        throw new IllegalArgumentException("Name must be populated");
    }

    mName = name;
}

private int    mId    = 0;
private String mName = null;
}
```

Listing 3.13 displays the output of the `unittest` target with `showoutput` and `printsummary` set to `yes`. This can be useful in debugging or when you create your nightly unit test process, later in this chapter.

LISTING 3.13 Output from `unittest` Target with `showoutput` Set to `Yes`

```
% ant unittest
Buildfile: build.xml

unittest:
[junit] Running com.networksByteDesign.eMarket.inventory.salesItemTest
[junit] ID = 123
[junit] Name = Router
[junit] ID = -123
[junit] ID = 0
[junit] Name =
[junit] Name = null
```

LISTING 3.13 Continued

```

[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.034 sec
[junit] TEST
      com.networksByteDesign.eMarket.inventory.salesItemTest FAILED
[junit] Running com.networksByteDesign.eMarket.inventory.customerTest
[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.009 sec
[junit] TEST
      com.networksByteDesign.eMarket.inventory.customerTest FAILED
[junit] Running com.networksByteDesign.eMarket.inventory.companyTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.186 sec

cleanupUnittest:

BUILD FAILED
file:/usr/projects/eMarket/build.xml:32: One or more unit tests failed.
Total time: 2 seconds

```

Formatter

The `<junit>`JUnit task provides formatter classes to facilitate the handling of the output from unit tests. These classes listen to all of the output and act as both filters and formatters in presenting the final output. A formatter can be added by using the nested `formatter` tag inside the `<junit>` task. Three basic formatters are provided by the `<junit>` task:

- Plain
- Brief
- XML

It's also possible to develop custom formatters. The following target shows how the formatter is set.

```

<!-- unittest target -->
<target name="unittest"
      description="Run the unit tests for the source code.">
  <mkdir dir="${dirs.test}"/>
  <junit haltonfailure="no"
        printsummary="yes"
        showoutput="yes">
    <formatter type="brief" usefile="true" />
    . . .
  </junit>
</target>

```

Plain Formatter

Plain is a flat-file text format that provides information about both the tests that failed and those that succeeded. If the output does not need to be parsed by another process and information on successful tests is desired, this is probably the formatter type to select. Listing 3.14 shows the output of a unit test run with the formatter set to plain.

LISTING 3.14 Sample JUnit Output File with JUnit Task Set to Plain Formatter

```
% ant unittest
Buildfile: build.xml

unittest:
[junit] Running com.networksByteDesign.eMarket.inventory.salesItemTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.025 sec
[junit] Testsuite: com.networksByteDesign.eMarket.inventory.salesItemTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.025 sec

[junit] Testcase: testConstructor took 0.017 sec
[junit] Testcase: testSetter took 0 sec
[junit] Running com.networksByteDesign.eMarket.inventory.customerTest
[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.113 sec
[junit] Testsuite: com.networksByteDesign.eMarket.inventory.customerTest
[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.113 sec

[junit] Testcase: testConstructor took 0.003 sec
[junit]     FAILED
[junit] Expected exception was not thrown
[junit] junit.framework.AssertionFailedError:
      Expected exception was not thrown
[junit]     at com.networksByteDesign.eMarket.inventory.customerTest.
          testConstructor(Unknown Source)
[junit]     at sun.reflect.NativeMethodAccessorImpl.
          invoke0(Native Method)
[junit]     at sun.reflect.NativeMethodAccessorImpl.
          invoke(NativeMethodAccessorImpl.java:39)
[junit]     at sun.reflect.DelegatingMethodAccessorImpl.
          invoke(DelegatingMethodAccessorImpl.java:25)

[junit] Testcase: testConstructorTestcase: testSetter took 0 sec
[junit] TEST com.networksByteDesign.eMarket.inventory.customerTest FAILED
[junit] Running com.networksByteDesign.eMarket.inventory.companyTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.033 sec
```

LISTING 3.14 Continued

```
[junit] Testsuite: com.networksByteDesign.eMarket.inventory.companyTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.033 sec

[junit] Testcase: testConstructor took 0.003 sec
[junit] Testcase: testSetter took 0.001 sec

cleanupUnittest:

BUILD FAILED
file:build.xml:80: One or more unit tests failed.

Total time: 2 seconds
```

Brief Formatter

Brief is the same as the plain formatter, except that detailed information on successful tests is filtered out. If the output does not need to be parsed by another process and detailed information only on failed tests is desired, this is probably the formatter type to select. Listing 3.15 shows the output of a unit test run with the formatter set to *brief*.

LISTING 3.15 Sample JUnit Output File with JUnit Task Set to Brief Formatter

```
% ant unittest
Buildfile: build.xml

unittest:
[junit] Testsuite: com.networksByteDesign.eMarket.inventory.salesItemTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.005 sec

[junit] Testsuite: com.networksByteDesign.eMarket.inventory.customerTest
[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.007 sec

[junit] Testcase: testConstructor(com.networksByteDesign.eMarket.
                    inventory.customerTest): FAILED
[junit] Expected exception was not thrown
[junit] junit.framework.AssertionFailedError: Expected exception was not
                    thrown
[junit]     at com.networksByteDesign.eMarket.inventory.customerTest.
                    testConstructor(Unknown Source)
[junit]     at sun.reflect.NativeMethodAccessorImpl.
                    invoke0(Native Method)
```

LISTING 3.15 Continued

```

[junit]    at sun.reflect.NativeMethodAccessorImpl.
           invoke(NativeMethodAccessorImpl.java:39)
[junit]    at sun.reflect.DelegatingMethodAccessorImpl.
           invoke(DelegatingMethodAccessorImpl.java:25)

[junit] TEST com.networksByteDesign.eMarket.inventory.customerTest FAILED
[junit] Testsuite: com.networksByteDesign.eMarket.inventory.companyTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.001 sec

cleanupUnittest:

BUILD FAILED
file:build.xml:82: One or more unit tests failed.

Total time: 2 seconds

```

XML Formatter

The XML format provides the most information and should be used whenever the output will be parsed by another process, such as an XSLT to generate an HTML report. However, because some constructs are illegal in XML, the output from your unit tests may be filtered to prevent the inclusion of information that would invalidate the XML file. Listing 3.16 shows the output of a unit test run with the formatter set to `xml`. The XML formatter can be used to supply the test results in XML to the `<junitreport>` task, which provides HTML reports and is discussed in Chapter 5, “Creating the Automated Nightly Build.”

LISTING 3.16 Sample JUnit Output File with JUnit Task Set to XML Formatter

```

% ant unittest
Buildfile: build.xml

unittest:
[junit] <?xml version="1.0" encoding="UTF-8" ?>
[junit] <testsuite errors="0" failures="0"
           name="com.networksByteDesign.eMarket.inventory.salesItemTest"
           tests="2" time="0.169">
[junit]   <properties>
[junit]     <property name="dirs.temp" value="/tmp"></property>
[junit]     <property name="java.vm.version"
           value="1.4.1_01-12"></property>

```

LISTING 3.16 Continued

```

[junit]     <property name="java.io.tmpdir" value="/tmp"></property>
[junit]     <property name="os.name" value="Mac OS X"></property>
[junit]     <property name="ant.home"
                value="/usr/software/ant/"></property>
...

[junit] </properties>
[junit] <testcase name="testConstructor" time="0.0050"></testcase>
[junit] <testcase name="testSetter" time="0.0"></testcase>
[junit] <system-out><![CDATA[]]></system-out>
[junit] <system-err><![CDATA[]]></system-err>
[junit] </testsuite>
[junit] <?xml version="1.0" encoding="UTF-8" ?>
[junit] <testsuite errors="0" failures="1"
                name="com.networksByteDesign.eMarket.inventory.customerTest"
                tests="2" time="0.027">
[junit]   <properties>
[junit]     <property name="dirs.temp" value="/tmp"></property>
[junit]     <property name="java.vm.version"
                value="1.4.1_01-12"></property>
[junit]     <property name="java.io.tmpdir" value="/tmp"></property>
[junit]     <property name="os.name" value="Mac OS X"></property>
[junit]     <property name="ant.home"
                value="/usr/software/ant/"></property>
...

[junit] </properties>
[junit] <testcase name="testConstructor" time="0.0070">
[junit]   <failure message="Expected exception was not thrown"
                type="junit.framework.AssertionFailedError">
                junit.framework.AssertionFailedError:
                Expected exception was not thrown
[junit]   at com.networksByteDesign.eMarket.inventory.customerTest.
                testConstructor(Unknown Source)
[junit]   at sun.reflect.NativeMethodAccessorImpl.
                invoke0(Native Method)
[junit]   at sun.reflect.NativeMethodAccessorImpl.
                invoke(NativeMethodAccessorImpl.java:39)
[junit]   at sun.reflect.DelegatingMethodAccessorImpl.
                invoke(DelegatingMethodAccessorImpl.java:25)

```

LISTING 3.16 Continued

```
[junit] </failure>
[junit] <system-out><![CDATA[]]></system-out>
[junit] <system-err><![CDATA[]]></system-err>

...

[junit] </testsuite>
```

```
cleanupUnitTest:
```

```
BUILD FAILED
file:build.xml:82: One or more unit tests failed.
```

```
Total time: 4 seconds
```

Direct Output with the usefile Attribute

Another attribute of the formatter tag is `usefile`. Normally, all information published by a formatter is sent to a file. Especially in the case of the XML formatter, where later processing is planned to take place, this file-based approach is usually the best way. However, if the intent is to provide feedback directly to the user or to add the information to a build log, setting `usefile = "no"` will send the formatter information to the screen instead.

The files created by the `<formatter>` are named `TEST-<the name of the class>.txt` for plain and brief and `TEST-<the name of the class>.xml` for xml. The filename for the `companyTest` class would be `TEST-com.networksByteDesign.eMarket.inventory.companyTest.txt` or `TEST-com.networksByteDesign.eMarket.inventory.companyTest.xml`.

Alternative TestRunners

JUnit uses a class called `TestRunner` to execute the individual tests and display the results. JUnit provides alternative `TestRunner` classes to meet the needs of various users. All of the tests we have been running use the `textui TestRunner` as can be seen in Listing 3.17.

LISTING 3.17 The Output from Running a Unit Test Using the `textui TestRunner`

```
% java junit.textui.TestRunner
    com.networksByteDesign.eMarket.inventory.salesItemTest
.F.
Time: 0.006
```

LISTING 3.17 Continued

There was 1 failure:

```
1) testConstructor(com.networksByteDesign.eMarket.inventory.salesItemTest)
   "Expected exception was not thrown"
```

FAILURES!!!

```
Tests run: 2, Failures: 1, Errors: 0
```

Ant offers an AWT TestRunner that provides a graphical user interface to the test results. This can be used by calling `java junit.awtui.TestRunner com.networksByteDesign.eMarket.inventory.salesItemTest` and can be seen in Figure 3.1.

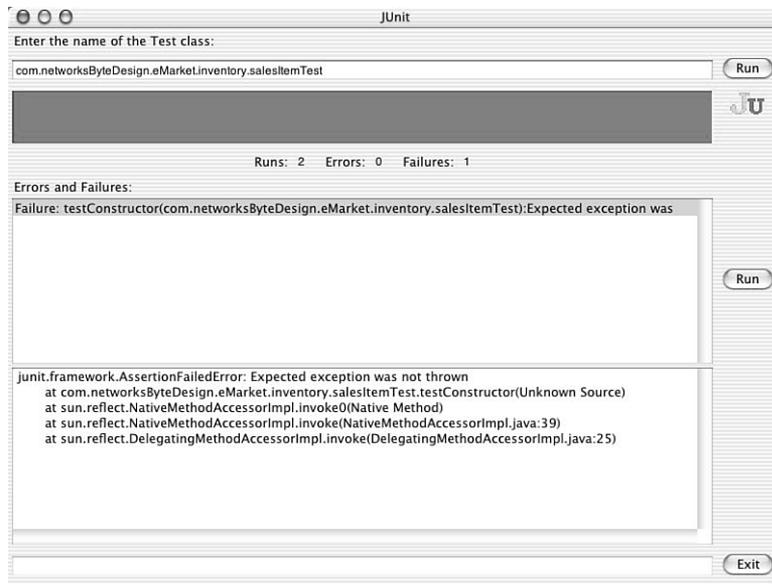


FIGURE 3.1 The AWT TestRunner provides a graphical user interface for the test results.

Ant also offers a Swing version of the TestRunner, which provides a more modern graphical interface. This can be used by calling `java junit.swingui.TestRunner com.networksByteDesign.eMarket.inventory.salesItemTest` and can be seen in Figure 3.2.

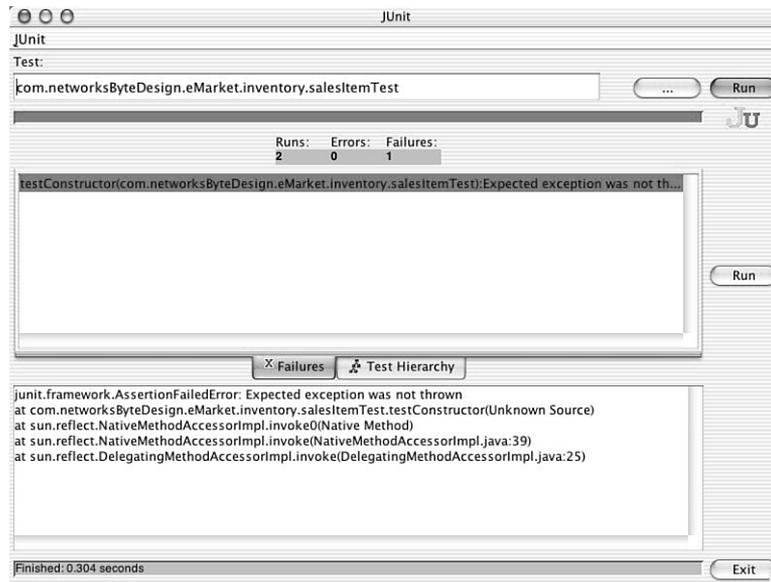


FIGURE 3.2 The Swing version of the TestRunner provides a more modern graphical interface.

Forking Unit Tests

It's possible for unit tests to have undesirable side effects on the build process. For example, if someone put a condition in their code that called `System.exit()`, this could cause the entire build process to exit. We would rather have just the unit test fail, but the entire build process continues to completion. Ant provides a way to insulate the build process from unintended side effects caused by unit testing, by allowing unit tests to be forked into their own JVM. There are several reasons why you might want to fork the JUnit tests into a separate process:

- As just discussed, forking will isolate a unit test from the build process. If the unit test caused a condition that prompted the process to exit, this would prevent the build process from exiting because the unit test is executing as a separate process.
- Forking will allow the JVM to take advantage of a multiprocessor platform.
- Forking unit tests into separate processes can also insulate them from picking up side effects that might occur in the build process. For example, if a unit test modifies the state of a singleton class, another unit test running in the same JVM might execute differently than if the singleton's state had never been altered.

Version-Control Systems

Story

On Wednesday of the second week of the project, the development server finally became available. Until now, the developers had been doing all work on their local desktop machines, and transferring files between one another for compiling and testing. This includes the initial spike tests as well as development of production code. One of the developers on the team is John Reynolds, who is a Java developer and has expertise in CVS. Michael asks John to set up their CVS repository, and to pair with Jeff so that Jeff can learn more about setting up CVS. Jeff in turn will be able to teach John more about using Ant. The two will work together to define the necessary targets in the buildfile to handle their CVS needs. Because Jeff has completed the unit testing targets for the buildfile, he begins adding some targets that they will definitely need: perform a CVS update and perform a clean build.

After the CVS repository is set up, the developers are able to add and commit their code to the repository, and begin controlling their code in CVS. Michael sets the requirement that code committed to CVS must compile and pass its unit tests.

No matter what the size of the development team is, version control plays an important role in the process. At one level, you can think of version control as a part of an automated backup process. Beyond that basic level, version control enables development team members to work in conjunction with each other, regress to earlier versions, and treat a group of files as a single unit. We believe that it's important at the outset to set up a version-control practice. Version control is one of those areas that isn't as interesting as architecture or design, and it can easily be neglected. This is exactly why we think it's important to set up such a system initially; that way, it isn't postponed or neglected entirely.

CVS Access and Logins

Ant supports several version-control systems, either natively or through the use of optional tasks. The version-control system called *Concurrent Versions System* (CVS) is used by Ant's developers to manage the Ant source code and is the version-control system that is the most integrated with Ant. CVS is widely used in both open-source and commercial software development, and it is available on most platforms. Although we focus on CVS here, the concepts demonstrated in this book can be applied to almost any revision-control system.

CVS Login

CVS has a repository that contains the revision history of each of the files. The location of this repository typically is referred to as `CVSRROOT`. Usually, an environment variable named `CVSRROOT` is set, and CVS commands implicitly make use of it to determine which repository to act on. The value of `CVSRROOT` also can be passed as a parameter to CVS commands.

Most CVS tasks in Ant behave in the same manner as their CVS command-line counterparts. If the `CVSR00T` environment variable is set, the Ant tasks pick it up automatically. `CVSR00T` also can be set as an attribute on the CVS tasks. The one exception is the `<cvslogin>` task, which requires `CVSR00T` to be set as an attribute because it ignores the environment variable.

From a user's viewpoint, logging into CVS is not a step that takes place every time the developer accesses CVS. When a user logs in to CVS, a local password file, typically named `.cvspass`, is created. This file is used to authenticate the user on subsequent calls to CVS. So, the CVS login command doesn't actually log in the user, as most systems do, but it creates the password file in preparation for later CVS commands. Therefore, entering an invalid password when logging in does not do anything. The result is seen only when the first CVS command is run. Listing 3.18 shows the `cvsLogin` target.

LISTING 3.18 `cvsLogin` Target

```
<!-- cvsLogin target -->
  <target name="cvsLogin">
    <input message="Please enter CVS password:"
          addproperty="cvsPassword" />

    <condition property="noCVSPassword">
      <equals arg1="" arg2="{cvsPassword}" />
    </condition>

    <fail if="noCVSPassword">You did not enter your CVS password.</fail>

    <cvspass cvsroot=":local:/usr/local/cvsArchive"
            password="{cvsPassword}" />
  </target>
```

This target begins by incorporating the concepts from the backup tasks, where the `<input>` task was used to obtain the `cvsPassword` from the user. If no password is entered, the task fails with an appropriate error message.

Next we call the `cvspass` task to set the `cvsroot` and `password` attributes. The other optional attribute is `passfile`, which defaults to `.cvspass` if not overridden.

In reality, the `cvsLogin` target rarely is called because, in most installations of CVS, the login process is an operation that takes place once per user per machine. However, if users move from machine to machine, or if new users frequently are added, the frequency of logins could increase tremendously. Even if logging into CVS needs to happen more frequently, this step should occur naturally during the build process.

A Custom Task to Check for Required Input Parameters

Story

After a number of changes to the buildfile have been made to check that mandatory parameters are set, it becomes apparent to the developers that there is a lot of replication throughout the buildfile to check for required input parameters. Besides being sloppy, this replication creates a maintenance problem. The development team agrees that a pair of developers should write a custom task to clean this up and consolidate this functionality into one place.

The buildfile has grown to include a lot of constructs to verify that mandatory parameters are set. Recall from Chapter 2, “Creating Initial Spikes,” that properties are immutable and can’t be used like ordinary variables in a scripting language. Because of this, we wind up setting a number of temporary parameters to check for required input, and the buildfile becomes unwieldy. Also, whenever we want to require parameters in a buildfile, we wind up rewriting the same XML. The team decides to create a custom task that handles required input. This consolidates the checking of required input into a single task and also enables them to write and debug this code once. Listing 3.19 shows the custom task `RequiredInput`, which extends the Ant `Input` task. The `Input` task is not an abstract base class listed in the Ant documentation as a task specifically intended for extension, but there’s also no reason why other tasks in Ant can’t be extended.

A Task class in general has private attributes, public setter methods for each attribute, and a public `execute` method. When the task executes, the Ant framework calls the setter method for each attribute that is set in the XML tag. Then the public `execute` method is called to do the actual work. Appendix B, “Extending Ant,” explains the general procedure for writing a custom task as well as methods for setting nested elements.

One of the problems faced in writing this task is that it’s necessary to know the value of the attributes of the `Input` class, but they’re private and the derived class doesn’t have access to them. The way around this problem is to override the setter classes. Each setter method calls the base class’s setter method (such as `super.methodname()`) and then stores the value of the attribute in the class. This grants access to a copy of the attribute values in the derived class.

When the `RequiredInput` task executes, it checks the boolean values to determine whether all the required attributes have been set. If they haven’t, the value of the unpopulated attribute is `false`. The `execute` method then throws an Ant `BuildException`, which fails the build and prints an error message indicating which required parameter hasn’t been set. This task is shown in Listing 3.19, which extends the `Input` task and simplifies the process of checking for mandatory parameters.

LISTING 3.19 A Custom Task for Checking for Required Input

```
package com.networksByteDesign.util;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.taskdefs.Input;

public class RequiredInput extends Input
{
    /**
     * Defines the name of a property to be created from input. Behavior is
     * according to property task, which means that existing properties
     * cannot be overridden.
     *
     * @param addproperty Name for the property to be created from input
     */
    public void setAddproperty (String addproperty)
    {
        super.setAddproperty(addproperty);
        propertyName      = addproperty;
        havePropertyName = true;
    }

    /**
     * Returns the property that gets set during the build run.
     * @return The property being set.
     */
    public String getProperty()
    {
        return propertyName;
    }

    /**
     * Sets the Error Message which gets displayed to the user during
     * the build run.
     * @param errorMessage The error message to be displayed.
     */
    public void setErrorMessage (String errorMessage)
    {
        this.errorMessage = errorMessage;
        haveErrorMessage  = true;
    }
}
```

LISTING 3.19 Continued

```
/**
 * Returns the Error Message which gets displayed to the user during
 * the build run.
 * @return The error message to be displayed.
 */
public String getErrorMessage()
{
    return errorMessage;
}

/**
 * Actual test method executed by ant.
 * @exception BuildException
 */
public void execute () throws BuildException
{
    if (!havePropertyName)
    {
        throw new BuildException("Missing attribute propertyName",
                                location);
    }

    if (!haveErrorMessage)
    {
        throw new BuildException("Missing attribute errorMessage",
                                location);
    }

    super.execute();

    if(getProject().getProperty(propertyName).trim().length() == 0)
    {
        throw new BuildException(errorMessage, location);
    }
}

private String  propertyName    = "";
private String  errorMessage    = "";
private boolean haveErrorMessage = false;
private boolean havePropertyName = false;
}
```

Appendix B describes the integration of a custom task in detail. One method of hooking a custom task into an Ant buildfile is to declare a mapping between the classname and the taskname with a `<taskdef>` tag, as shown here:

```
<taskdef name="requiredInput"
        classname="com.networksByteDesign.util.RequiredInput" />
```

Afterward, all that's required is to put the classfile in your system CLASSPATH, and then run the buildfile. There is an alternate, more "permanent" way to integrate a task, which is described in Appendix B.

Listing 3.20 shows the custom RequiredInput task hooked into our cvslogin target.

LISTING 3.20 RequiredInput Task Hooked into Build File

```
<?xml version="1.0" ?>
<project name="eMarket" default="compile" basedir=".">

  <taskdef name="requiredInput"
          classname="com.networksByteDesign.util.RequiredInput" />

  <property name="dirs.source" value="/usr/projects/eMarket/src" />
  <property name="dirs.backup" value="{user.home}/backup" />
  <property name="dirs.temp" value="/tmp" />

  <!-- cvsLogin target -->
  <target name="cvsLogin">
    <requiredInput message="Please enter CVS password:"
                  addproperty="cvsPassword"
                  errorMessage=" You didn't enter your CVS password." />

    <cvspass cvsroot=":local:/usr/local/cvsArchive"
            password="{cvsPassword}" />
  </target>

  <!-- compile target -->
  <target name="compile" description="Compile all of the source code.">
    <javac srcdir="{dirs.source}" />
  </target>
</project>
```

CVS Initialization

Now that we have refactored our Ant buildfile to deal with required input tasks, let's make the CVS login process transparent to the user. Because the CVS login is typically a rarely used target, we'd like to make sure it handles the login when it is necessary but doesn't get in the way of normal usage.

To accomplish this, we'll take a look at the `<available>` task. The purpose of the `<available>` task is to allow a specified property to be set if a particular resource, such as a file, a class, a directory, or a JVM system resource, exists. In this case, you can make the CVS login process transparent by looking for the `.cvspass` file. If it exists, you proceed normally. Otherwise, you prompt the user for the CVS password and create the `.cvspass` file. Listing 3.21 shows the `cvsInit` target.

LISTING 3.21 `cvsInit` Target

```
<!-- cvsInit target -->
  <target name="cvsInit">
    <available file="{user.home}/.cvspass"
              property="cvsAlreadyLoggedIn" />

    <antcall target="cvsLogin" />
  </target>
```

Next we modify the `cvsLogin` task to check for the `cvsAlreadyLoggedIn` property. Instead of using the `if` attribute as before, we use the `unless` attribute, which has the opposite effect. These changes are shown in Listing 3.22.

LISTING 3.22 `cvsLogin` Target (Revised)

```
<!-- cvsLogin target -->
  <target name="cvsLogin" unless="cvsAlreadyLoggedIn">
    <requiredInput message="Please enter CVS password:"
                  addproperty="cvsPassword"
                  errorMessage=" You didn't enter your CVS password." />

    <cvspass cvsroot=":local:/usr/local/cvsArchive"
            password="{cvsPassword}" />
  </target>
```

Now we can make all CVS targets depend on `cvsInit`, which ensures that the user is logged in if he has not previously done so. If a user already has logged in, the `cvsLogin` task will be bypassed.

The CVS Task

Now that you have handled the initial access into CVS, you can turn your attention to the <cv> task itself. In its simplest form, the <cv> task is simply a pass-through mechanism to the `cv` command itself. All the CVS commands and options can run through the <cv> task as well. Listing 3.23 shows the output from calling `cv - -help-command`. For additional information about CVS and a list of CVS commands and options, see www.cvshome.org/docs/manual/cvs.html.

INSTALLATION OF CVS

To install CVS, follow these steps:

1. Download CVS from
<http://www.cvshome.org>
 2. CVS is available for a number of different platforms. Be sure to check the installation instructions for your specific platform.
 3. The CVS manual can be found at:
<http://www.cvshome.org/docs/manual>
-

LISTING 3.23 Output from the `cv - -help-command`

```
% cv - -help-command
CVS commands are:
  add          Add a new file/directory to the repository
  admin        Administration front end for rcs
  annotate      Show last revision where each line was modified
  checkout     Checkout sources for editing
  commit       Check files into the repository
  diff         Show differences between revisions
  edit         Get ready to edit a watched file
  editors      See who is editing a watched file
  export       Export sources from CVS, similar to checkout
  history      Show repository access history
  import       Import sources into CVS, using vendor branches
  init         Create a CVS repository if it doesn't exist
  log          Print out history information for files
  login        Prompt for password for authenticating server.
  logout       Removes entry in .cvspass for remote repository.
  rdiff        Create 'patch' format diffs between releases
  release      Indicate that a Module is no longer in use
  remove       Remove an entry from the repository
  rtag         Add a symbolic tag to a module
  status       Display status information on checked out files
```

LISTING 3.23 Continued

tag	Add a symbolic tag to checked out version of files
unedit	Undo an edit command
update	Bring work tree in sync with repository
watch	Set watches
watchers	See who is watching a file

(Specify the --help option for a list of other help options)

CVS Checkouts

The first command we'll look at is the CVS checkout command. This command is used to retrieve a set of files from CVS into a new workspace. As with all CVS commands, you can configure a multitude of options and parameters to meet your specific needs. This book is not intended to be a tutorial on CVS, so we simply pick from some of the more common sets of options.

One of the advantages of using Ant for your build and deployment process is that you can easily check Ant buildfiles into version control, and you can track revisions of your buildfile as well. On many of our projects, the Ant buildfile had more revisions than the source code as we enhanced our build and deployment process throughout the lifecycle of the project. In fact, the buildfiles used as examples throughout this book were maintained in a CVS repository. The issue with keeping the Ant buildfile under version control is that it creates a circular dependency. To run a target on an Ant buildfile, you first must check the workspace out of version control. When the workspace has been checked out, the need for an Ant target to check out a new workspace is negated.

However, if the Ant file is contained in a separate workspace from the rest of the source code, a checkout target for pulling a particular release of the source code makes perfect sense. You will apply this technique later in the book when application deployment is discussed. For example, we have an application that has its installation CD built with Ant. The Ant buildfile checks out a copy of Tomcat and JBoss from CVS and includes it as part of the installation.

If you are familiar with other revision-control systems but not with CVS, there is an important distinction in the concept of checkout. In revision-control systems such as the Revision Control System (RCS), the Source Code Control System (SCCS), and Clearcase, checking out a file usually means retrieving a copy of it with the intent of modifying it. This means that the file is locked in the repository, preventing others from modifying the file until the person who locked the file releases the lock. By contrast, in CVS, checking out means pulling a copy of the source code, but not necessarily with the intent to modify any of it. Also, the checkout command can be used to pull code based on some criteria such as a tag, all the latest versions on the main branch, and so on. The important distinction here between CVS and other

revision-control systems is that the term checkout doesn't mean that the repository is locked for modification, as it implies in most other revision-control systems.

The concepts presented so far will assist you in building robust, reusable targets that can be used interactively by a developer, but also as part of an unattended build and deployment process.

Listing 3.24 shows the `cvsCheckout` Target. This target depends on `cvsInit`, which ensures that the user has previously logged in to CVS, either through the Ant rules or through CVS directly.

LISTING 3.24 `cvsCheckout` Target

```
<!-- cvsCheckout target -->
  <target name="cvsCheckout" depends="cvsInit">

    <requiredInput message="Please enter CVS module:"
      addproperty="cvsModule"
      errorMessage="You didn't enter a CVS module." />

    <requiredInput message="Please enter working directory:"
      addproperty="dirs.working"
      errorMessage="You didn't enter a working directory"/>

    <mkdir dir="${dirs.working}" />

    <cvs package="${cvsModule}" dest="${dirs.working}" />
  </target>
```

As with the backup targets, the user is prompted to provide the name of the module to check out from CVS, as well as the directory in which to place the checked-out code. Modules are a convenient way in CVS to group files and directories that are logically related. If the module or working directory is not properly entered, the target fails.

```
<mkdir dir="${dirs.working}" />
```

We then create the working directory in case it does not exist. As you learned earlier, if it does exist, the default behavior is to do nothing.

```
<cvs package="${cvsModule}" dest="${dirs.working}" />
```

Next, call the `cvs` task, passing the module that you want to check out in the `package` attribute and the working directory in the `dest` attribute. The default command for the `<cvs>` target, if no command is supplied, is to check out a new

workspace; for the purposes of this discussion, you do not need to be concerned with the cvs options passed to the checkout command.

The end result of the cvsCheckout target is a checked-out CVS module in the directory of your choice. As with the backup process, you can break the interactive part of cvsCheckout into a separate target so that you can produce an unattended version as well.

CVS Updates and Commits

Now that we have checked out a version of the source code, we need a way to pull the latest changes that other developers on the team are checking in. In CVS, this is done through the use of the update command. As with the checkout command, all CVS commands and options can be used in the command attribute of the cvs task. We must add the attribute to the new target because we relied on the default value for the checkout target.

The following code shows the cvsUpdate target. The module name is not required because CVS looks in the `${dirs.source}` to find the information it needs regarding the CVSROOT and the module where the source code is mastered. The update command displays a list of files that have been changed, updated, or added since the last update. For example, a ? in front of the filename indicates that this is a new file, an M indicates a modified file, and a C indicates that changes you have made conflict with other changes made to the same file and that CVS cannot resolve the conflicts automatically. Information on the meaning of each of the flags, as well as how to work with issues such as conflicts, can be found on the CVS Web site.

```
<!-- cvsUpdate target -->
<target name="cvsUpdate" depends="cvsInit">
  <cvs command="update" dest="${dirs.source}" />
</target>
```

In CVS, putting your changes into the repository is done with the commit command. This is analogous to the concept of checkin with other revision-control systems. Committing changes is straightforward. CVS prompts you to enter a comment describing the changes, which is always a good practice to follow. CVS attempts to use your default editor for the entering of the comments (the value of the EDITOR environment variable in UNIX, for example). The following code shows the Ant task for committing changes to CVS.

```
<!-- cvsCommit target -->
<target name="cvsCommit" depends="cvsInit">
  <cvs command="commit" dest="${dirs.source}" />
</target>
```

If you would rather use the `<input>` task for adding comments as you have been doing, just add the appropriate tasks, as shown in Listing 3.25.

LISTING 3.25 cvsCommit Target (Revised)

```
<!-- cvsCommit target -->
  <target name="cvsCommit" depends="cvsInit">
    <requiredInput message="Please enter your CVS comment:"
      addproperty="cvsComment"
      errorMessage=" You did not enter a CVS comment." />
    <cvs command="commit -m ${cvsComment}" dest="${dirs.source}" />
  </target>
```

In this case, we are enforcing a team policy in the build process rather than relying only on verbal and written policies and procedures. If the developer does not enter a cvs comment, the commit process will abort. If a comment is added, commit is called, with the `-m` flag passing the comment as a parameter.

The `cvsUpdate` and `cvsCommit` targets work fine until two developers change the same file. Even if the changes are not in conflict, if one developer does not call `cvsUpdate` before `cvsCommit`, `cvsCommit` will fail. Although the error message will indicate what went wrong, it is better to change `cvsCommit` to be dependent on `cvsUpdate`.

```
<target name="cvsCommit" depends="cvsUpdate" description="">
```

If the changes are in conflict, the update will indicate this and the commit will fail with an appropriate error message. This rule will check in all your changes with the same comment. Depending on your team standards, this might not be appropriate. However, if you make small, logical changes to the source code, this approach can be a best practice. By concentrating on the reason for the change rather than the specific code changes, which can be easily gleaned from a comparison of the two versions, a reviewer of the comments can better understand the motivation for certain changes.

Basic Deployment

Story

During the course of testing, the team is expending effort by manually copying the class files to a directory for deployment on the app server. Scott and Lisa decide to automate the process. They add a simple deploy target to the buildfile that will copy the required class files and JAR files to the correct directory. This will save time and effort because they are now having to deploy on a frequent basis.

A full deployment can be a complex process in which to fully automate the steps. The team is taking the correct approach in creating a simple process and adding the complexity only as needed. The deployment target described in Listing 3.26 will simply jar up the class files and copy the JAR file and all JSP pages to the deployment directory.

LISTING 3.26 Simple Deployment Target

```

<?xml version="1.0" ?>
<project name="eMarket" default="compile" basedir=".">

    <taskdef name="requiredInput"
            classname="com.networksByteDesign.util.RequiredInput" />

    <property name="dirs.source" value="/usr/projects/eMarket/src" />
    <property name="dirs.backup" value="${user.home}/backup" />
    <property name="dirs.temp" value="/tmp" />
    <property name="dirs.deploy" value="/usr/projects/appServer/eMarket" />

    <!-- compile target -->
    <target name="compile" description="Compile all of the source code.">
        <javac srcdir="${dirs.source}" />
    </target>

    <!-- deploy target -->
    <target name="deploy" description="Simple deployment of the app">
        <jar jarfile="${ant.project.name}.jar">
            <fileset dir="${dirs.source}" includes="**/*.class" />
        </jar>
        <copy file="${ant.project.name}.jar" todir="${dirs.deploy}" />
        <copy flatten="true" todir="${dirs.deploy}">
            <fileset dir="${dirs.source}" includes="**/*.jsp" />
        </copy>
    </target>
</project>

```

Our `<deploy>` target begins by creating a JAR file containing all of the class files. The JAR file will be named with the name of the project, which in our case will create a JAR file called `eMarket.jar`. The file will be copied to the deployment directory.

The `<deploy>` target then goes on to copy all JSP pages to the deployment directory. Notice the `flatten` attribute in the `<copy>` task. When this attribute is set, the directory structure from the source is not preserved. So, even if the JSP pages are scattered in various subdirectories, they will be copied directly into the deployment directory.

Story

The group reached the end of the first iteration and their stories are complete. Also, because the buildfile is able to perform the needed requirements, Jeff and John decide to refactor it to clean it up. The development team and the customer get together and look over the user interface. As expected, the customer makes some changes in the user stories. At the iteration planning meeting for the second iteration, Sandy, Michael, the developers, and the customer meet again to have another look at the user stories. Once again, the customer sets the priorities based on business value, and the plan is created for the next iteration.

Summary

In this chapter, the team began their first iteration. We've seen the first unit test and hooked JUnit into the Ant build process. We've used attributes such as `haltonfailure`, `propertyfailure`, `printsummary`, `showoutput`, and `fork` to tune the unit-testing process to meet the team's needs.

We have also introduced the use of CVS as the version control system. We have gone through the process of logging into CVS, and performing a checkout, update, and commit. We also developed a custom task to simplify the use of the `<input>` task.

Finally, a simple deployment target was developed that handled the creation of a JAR file along with copying the necessary files to the deployment directory

