Writing and Testing an Ant task

This is an outtake from Java Development with Ant, by Erik and Steve, homepage http://manning.com/antbook . This section was written early on but then pulled through lack of space; a focus on how to use Ant got in the way of how to do Ant coding. We still think everyone writing Ant tasks should know how to test them, so here is the text.

This article shows you how to write and test a simple Ant task, but it hasn't been through the technical or editorial reviews to be accurate or readable. Nor has it been regression tested. One thing skipped is where the Ant test class comes from –for now you need to pull that from the CVS source.

Readers of our book will recognize the random number generation problem, as we showed how to do this in a `<script>` *element; those who haven't read the book should either get a copy to see our explanation of task lifecycle and methods, or use the Ant source and javadocs to work it all out for themselves.*

Let's write a task to generate a random number. The first step is to think of the core functionality of the task:

- Generate a random integer
- Provide some means of specifying the range of the number
- Store it in a named property

It is easy to imagine some feature creep options –such as selecting a random item from a list provided, or controlling the output format, but in true XP style, we leave these out until actually needed. The behavior of the core task is then quite simple: there needs to be a means of specifying the variable name and a maximum value. In the build file, a use of the task might look something like:

```
<random name="delay.seconds" min="30" max="60" />
<sleep seconds="${delay.seconds}"/>
```

From that example, it is easy to see that two properties need to be set prior to the random number being generated, an empty name is invalid and if the maximum value. The minimum value is optional, defaulting to zero. This is enough of a specification to write the task:

```
package org.tasklib;
import org.apache.tools.ant.*;

public class Random extends Task {  |#1

    protected int min=0;
    protected int max=0;
    protected String name;

    public void setMin(int min) {         |#2
        this.min = min;                   |#2
    }                                     |#2

    public void setMax(int max) {         |#3
```

```
        this.max = max;                      |#3
    }                                        |#3

    public void setName(String name) {     |#4
        this.name = name;                    |#4
    }                                        |#4

    public void execute() {                                |#5
        int diff=max-min;
        double r=java.lang.Math.random();                  |#5
        int result=min+(int)(diff*r);                      |#5
        project.setNewProperty(name,Integer.toString(result));    |#5
    }
}                                                          |#5
```
#1<declare a subclass of task>
#2<setter for the min attribute>
#3<setter for the max attribute>
#4<setter for the name attribute>
#5<the actual work>

The code is quite simple: there are three properties that can be set with public methods, and a method called `execute` which does some work. This method calls `project.setNewProperty` with a name and a string value, so presumably this sets the property of the current project to the named value. The class also extends `org.apache.ant.Task`, which contains a declaration of the `execute` method, so that at run time Ant can execute anything defined as a subclass of that class.

Here is the build file which to compile, import and run the task:

```
<?xml version="1.0" ?>
<project name="firstTask" default="run">

<target name="compile">
    <mkdir dir="build"/>
    <javac srcdir="src" destdir="build"/>
    <taskdef name="random" classname="org.tasklib.Random"
        classpath="build"/>
</target>

<target name="run" depends="compile">
    <random min="0" max="9" name="random" />
    <echo>Random number is ${random}</echo>
</target>
</project>
```

The actual compilation process is a simple `<javac>` task, and because the imported package is part of the Ant runtime, it is automatically included in the build. Immediately after the compilation, the task is defined with `<taskdef>`, just as if a prebuilt task was being imported. Usually `<taskdef>` imports, if they state a classpath at all, will refer to one or more JAR files. Here, to avoid the overhead of building an archive, the classpath is declared as the build directories containing the compiler output. Once the task is defined, it can be used, which is the role of the `test-random` target:

```
run:
     [echo] Random number is 7
```

Running the task repeatedly returns different numbers, which indicates that the task is probably working. Some more testing is still in order –there have been no attempts to break the task yet.

Now, how exactly did that work? How did Ant go from a little bit of XML into a Java class, set the properties of the class and then call it at the right time for it to set the property? The answer to that question is about half the magic in the Ant source, and some of the most complicated and sophisticated Java code many Java developers are likely to see. For the curious, go look at `IntrospectionHelper.java` in the Ant source.

Error Handling

The current task example contains no code to test for errors. What happens when the task is used in a build file with values that are somehow incorrect? Adding four tests to the `build.xml` file permits us to find out:

```
<target name="test1"   depends="definitions">
    <random name="random1" />
    <echo>random with no min/max : ${random1}</echo>
    <random name="random2" min="5" max="5" />
    <echo>random with min==max : ${random2}</echo>
</target>
```

The result of this test is that the value of property `${random1}` is zero, that of `${random2}` five:

```
test1:
     [echo] random with no min/max : 0
     [echo] random with min==max : 5
```

Clearly calling the task with the minimum and maximum values equal generates a number with the same value –not exactly random, but a reasonable result. When calling the random task with no minimum and maximum, the result is as if they had both been set to zero –which is exactly what was done when the task was constructed. So the same behavior of generating a number that equals the maximum and minimum occurs. It may be appropriate to warn users when this occurs, as it may be a sign of a configuration error in the build file.

```
<target name="test2"   depends="definitions">
    <random name="random3" min="0" max="-5" />
    <echo>random with min greater than max : ${random3}</echo>
</target>
```

This next test is interesting. What happens when the maximum is less than the minimum? Running the test repeatedly produced values in the range zero to minus four.

```
test2:
     [echo] random with min greater than max : -4
```

This is a result of the algorithm used to set the range of the random number. This is done by first calculating the difference of the maximum and minimum values, multiplying the generated random number between zero and one by this difference, then adding it to the minimum. If the maximum value is less than the minimum then the difference is negative, which when multiplied by the generated random number produces a value to subtract from the minimum. This could be viewed as a feature and be documented accordingly, but to avoid confusion to people who never read the documentation, it would be better if the task verified that the maximum value was always greater than or equal to the minimum.

```
<target name="test3"   depends="definitions">
    <random />
</target>
```

The most minimum use of the `<random>` task is not to provide any arguments at all. What should be the behavior in this case? The name value is not an optional parameter, and so there is an error in the XML file, and error that should stop the build. When the test is run the build certainly stops, but it does so without being very informative.

```
test3:
BUILD FAILED
java.lang.NullPointerException
  at java.util.Hashtable.get(Hashtable.java:320)
  at org.apache.tools.ant.Project.setProperty(Project.java:265)
  at org.tasklib.Random.execute(Random.java:32)
  ...
```

The failure is this correct behavior, but the user needs more information if the task is to be usable –and if the author of the task does not want to field support calls on a regular basis. Therefore, some verification of the name value being assigned is appropriate.

```
<target name="test4"   depends="definitions">
    <random name="random1" min="seventeen" />
    <echo>random with a type error: ${random1}</echo></target>
</project>
```

The final test also fails when passed an invalid value, this time with a different exception somewhere outside our new task.

```
java.lang.NumberFormatException: seventeen
  at java.lang.Integer.parseInt(Integer.java:405)
  at java.lang.Integer.<init>(Integer.java:540)
  at org.apache.tools.ant.IntrospectionHelper$9.set
    (IntrospectionHelper.java:470)
    ...
```

This happens during the process of converting the string format attributes into the integer values used for the minimum and maximum properties. It would be nice if the line number of the XML files was displayed with the error, but that would require some changes to the Ant source. Ant does this for normal tasks, but it appears that the `UnknownElement` methods need to include location information when reporting errors.

Reviewing the behavior of the `<random>` task on invalid inputs, it is clear that better validation is needed. Now that we have just written the tests, the actual validation code is simple to write –and it is simple to verify that the new code behaves as desired. This only takes a few more lines of code. The first change is to extend the definition of the `execute()` method to throw a `BuildException`. This is not actually mandatory, as it is part of the definition of the `Task` interface. It does, however, serve as an indicator as to what is to follow, which is a series of tests validating different parameters. If the `name` attribute is null or an empty string, or the `max` attribute less than the `min` attribute, then a `BuildException` is thrown to stop the build. The case in which the minimum and maximum values were identical was felt to be not so critical, -in that situation a warning message is output, but the task continues,

```
    public void execute()
            throws BuildException {                         |#1
        if(name==null || name.length()==0){                 |#2
            throw new BuildException("name is undefined");  |#2
        }                                                   |#2
        int diff=max-min;                                   |#3
        if(diff<0) {                                        |#3
            throw new BuildException("invalid range");      |#3
        }                                                   |#3
        if(diff==0) {                                       |#4
            log("minimum and maximum values are identical", |#4
                Project.MSG_WARN);                          |#4
        }                                                   |#4
        double r=java.lang.Math.random();
        int result=min+(int)(diff*r);
        project.setProperty(name,Integer.toString(result));
    }
```

#1<declare we throw the exception>
#2<check for no name>
#3<check for an invalid range>
#3<warn if min=max>

The result of these changes is that the task now fails when it should, and warns when it thinks the random number is not going to be random.

```
test1:
   [random] minimum and maximum values are identical
     [echo] random with no min/max : 0
   [random] minimum and maximum values are identical
     [echo] random with min==max : 5

BUILD SUCCESSFUL

test2:
BUILD FAILED
D:\Projects\firstTask\random2\build.xml:36: invalid range

test3:
BUILD FAILED
D:\Projects\ random2\build.xml:42: name is undefined
```

The addition of the validation has rounded the task off to the point where it is now usable. Some documentation would complete the task, but that is left as an exercise.

When the difference between the minimum and maximum numbers is zero, the task warns the user. This is done by calling the `log` method of the parent class. This method can take a string and print it as information to the user. Options exist to provide debug, warning or error messages through the addition of a second parameter –the options `MSG_DEBUG`, `MSG_WARN`, `MSG_ERROR` are constants in the class `Project`, along with the default value of `MSG_INFO`. When Ant is invoked with different verbosity options then the different levels of information are seen, as shown in the table. Different shells around Ant can also choose to display error and warning information differently. It is common for tasks to include logging information at the verbose level to aid debugging of build files; debug level information to debug the task itself:

Ant verbose flag	**Message categories displayed**
-quiet	MSG_WARN,MSG_ERROR
(none)	MSG_INFO,MSG_WARN,MSG_ERROR
-verbose	MSG_WARN, MSG_ERROR, MSG_VERBOSE
-debug	MSG_DEBUG, MSG_WARN, MSG_ERROR, MSG_VERBOSE

1.1 What output options show in terms of logging messages in the code

What remains is the automation of the test cases. All the built in Ant tasks have test cases of some form or other, and since we have already created some test targets to verify this new task, it seems appropriate to finish off with integration into the Ant test harness. The Ant test harness is, of course, JUnit. There are some extensions to make task invocation easier, but the basic concept of writing test cases as methods in a test class is the same.

```
public class RandomTest
    extends org.apache.tools.ant.BuildFileTest; {

    public RandomTest(String name) {
        super(name);
    }

    public void setUp() {
        configureProject("build.xml");
    }

    public void test0() {
        executeTarget("test0");
        String test0_a=project.getProperty("test0-a");
        assertNotNull(test0_a);
        String test0_b=project.getProperty("test0-b");
        assertNotNull(test0_b);
        if(test0_a.equals(test0_b))
            fail("numbers are not random");
    }

    public void test1a() {
        expectLog("test1a",
            "minimum and maximum values are identical");
    }

    public void test2() {
        expectBuildException("test2", "invalid range");
    }

    public void test3() {
        expectBuildException("test3", "name is undefined");
    }
}
```

Each of the test cases matches a target in the build file of the same name. The first test, `test0`, verifies that the random number generator generates random numbers, by verifying that two generated numbers are different. There is the intermittent probability that they will not, a one in a hundred million chance, so if this test does fail –try a second time. The remaining tests invoke targets providing a string to be logged or to be raised as a `BuildException`. If the exception is not raised, or the string is different, then all tests that invoke a target with `expectBuildException` will "succeed", in that they successfully indicate a problem.

The test that calls `expectLog` has the output of the task verified. The log tested is the complete log of messages of all types generated while executing the task, which includes all output streamed to the `system.out` and `system.err` output streams. This means it is possible to use these Ant unit tests to verify that the output from third party classes, libraries and applications behaves as expected.

Invoking these tests from the build file is no harder than invoking any other JUnit test, with the extra caveat that the Ant build file that contains the tests needs to be in the directory in which the tests are run. In this simple task, we are keeping the test cases in the main build.xml file, so the directory in which the tests run is set to the base directory of the project. In a more complex situation –the Ant unit tests themselves- a complete hierarchy of test directories can be used.