IBM

developerWorks®

Test your tests with Jester

Test-suite flaws are no joke

Elliotte Rusty Harold (elharo@metalab.unc.edu)
Adjunct Professor
Polytechnic University

03 March 2005

A comprehensive unit-test suite is a necessity for a robust program. But how can you be sure that your test suite is testing everything it should? Jester, Ivan Moore's JUnit test tester, excels at finding test-suite problems and provides unique insights into the structure of a code base. Elliotte Rusty Harold introduces Jester and shows how to use it for best results.

Test-first programming is the least controversial and most widely adopted part of Extreme Programming (XP). By now the majority of professional Java™ programmers have probably caught the testing bug. (See Resources for more information on being "test infected.") JUnit is the Java community's de facto standard test framework, and a system without a comprehensive JUnit test suite is incomplete. If your projects have comprehensive test suites, congratulations; you're producing good-quality software that has some hope of working. But most code bases are quite complex. Are you sure that every method is tested and every branch followed? If not, how will your application behave when those methods and branches are executed in production?

Code coverage

The next step beyond testing code is measuring the tests with a *code coverage* tool. Code coverage is a way of seeing how much code is covered by a set of tests. Confidence requires knowing not only that the program as a whole is tested but that each method is tested under all possible conditions. Traditionally such measurements have been performed by monitoring the tests as they execute, perhaps through the Java Virtual Machine Debugging Interface (JVMDI) or the Java Virtual Machine Tool Interface (JVMTI), or by directly instrumenting the bytecode. Any statements that are not executed at least once are not being tested.

Not foolproof

Jester's methodology isn't foolproof. The tool tends to report a lot of false positives. For instance, it might change the statement `System.out.println("Copyright 2005 Elliotte Rusty Harold")` to `System.out.println("Copyright 3005 Elliotte`

Rusty Harold") and then report that nothing broke. However, the false positives are normally easy to filter out. Besides, often you have reason to doubt whether something like this example really is a false positive. For instance, one could argue that claiming a copyright date of 3005 is a bug the test suite should notice.

This approach, taken by tools like Clover and EMMA (see Resources), is valuable for finding untested statements -- but it's not enough. Knowing that a statement isn't executed by the test suite proves that it isn't being tested. However, the inverse is not true. If a line of code is executed, it doesn't necessarily follow that it's tested. It's entirely possible that the test doesn't check whether the line of code produces the correct result.

Of course, no one writes test suites that independently verify the result of each statement. Among other problems, this would violate encapsulation. You tend to assume that each line of code in the method must be operating properly in order for the method to produce the expected result, given a certain input. But that assumption isn't justified. For instance, what if you're not testing all possible inputs, and therefore not testing the code designed to handle the edge conditions? Each line of code might still be tested, but you could be missing real bugs.

Introducing Jester

This is where Jester comes in. Unlike a traditional code coverage tool such as Clover, Jester doesn't watch which lines of code have been executed. Instead Jester changes the source code, recompiles it, and runs the test suite to see if anything breaks. For instance, it will change a 1 to a 2, or change an `if (x > y)` to `if (false)`. If the test suite isn't paying close enough attention to notice the change, then a test is missing.

I'll demonstrate Jester by using it on the open source Jaxen XPath tool (see Resources). Jaxen has a JUnit-based test suite, though the suite has less-than-perfect coverage.

Getting started

Before you can run Jester, all the unit tests must pass with the unmodified source code. If they don't, Jester won't know if its changes have broken anything. (For the demonstration I had to fix one bug I'd written a test case for but hadn't yet tracked down and stomped.)

Jester doesn't integrate particularly well (or at all) with IDEs, so it's important to set up the `CLASSPATH` and directories properly to make the tests pass. The exact command line you need to run the test suite varies widely from one project to the next. Because the Jaxen tests use relative URLs that point to certain test files, its tests must be run from within the jaxen directory. Here's how I eventually ran the Jaxen tests:

```
$ java -classpath ../jester136/jester.jar:target/lib/junit-3.8.1.jar
:target/lib/dom4j-core-1.4-dev-8.jar:target/lib/jdom-b10.jar
:target/lib/xom-1.0d21.jar:target/test-classes:target/classes
junit.textui.TestRunner org.jaxen.JaxenTests
```

You might need to clear one additional restriction on the test suite before running Jester. It must not print anything on `System.err` unless the test fails. Jester determines if the tests succeed by checking what's been printed, so program output to `System.err` often confuses it.

Once the test suite is running without failures, make a copy of your source tree. Remember, Jester is deliberately introducing bugs into the code, so you don't want to risk leaving one behind if something goes wrong. (This isn't so much of a concern if you're using source-code control. You are using source-code control, aren't you? If not, stop reading this article and check your code into a CVS or Subversion repository immediately.)

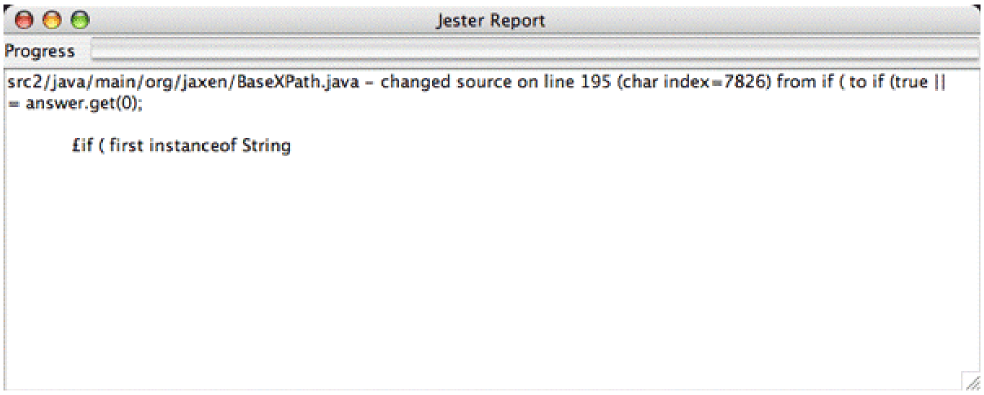
Running Jester

To run Jester, you must have both jester.jar and junit.jar on your classpath. (JUnit isn't bundled with Jester. You need to download it separately.) Jester looks for its configuration files in the classpath, so you must put the main Jester directory on the classpath too. Of course, you also need to add any other JARs or directories the tested application requires. The main class is `jester.TestTester`. The argument you pass to this program is the name of the test-suite class for the test application. (I had to write one for Jaxen because it didn't include a single class that ran all its tests.) Jester works much more reliably if you add all the necessary JAR files and directories to the `CLASSPATH` environment variable, rather the adding them to jre/lib/ext or referencing them with `-classpath`. Here's how I ran the initial test on Jaxen:

```
$ export CLASSPATH=src2/java/main:../jester136/jester.jar:../jester136
:target/lib/junit-3.8.1.jar:target/lib/dom4j-core-1.4-dev-8.jar
:target/lib/jdom-b10.jar:target/lib/jdom-b10.jar:target/lib/xom-1.0d21.jar
:target/test-classes:target/classes
$ java  jester.TestTester org.jaxen.JaxenTests src2/java/main
```

Jester runs slowly, even when checking a single file. It puts up the progress dialog shown in Figure 1 and prints output on `System.out` to let you know what it's doing and to reassure you it hasn't completely hung.

Figure 1. Jester progress



```
src2/java/main/org/jaxen/BaseXPath.java - changed source on line 195 (char index=7826) from if ( to if (true ||
= answer.get(0);

        £if ( first instanceof String
```

If you don't see any output after the first couple of minutes or so (or enough time to run the complete test suite, whichever is greater), Jester probably *is* hung, likely because of a classpath problem. If everything goes well, you should see output like that shown in Listing 1:

Listing 1. Jester output

```
Use classpath: src2/java/main:../jester136/jester.jar
:../jester136:target/lib/junit-3.8.1.jar:target/lib/dom4j-core-1.4-dev-8.jar
:target/lib/jdom-b10.jar:target/lib/jdom-b10.jar:target/lib/xom-1.0d21.jar
:target/test-classes:target/classes
```

```
...
src2/java/main/org/jaxen/BaseXPath.java
 - changed source on line 192 (char index=7757) from 1 to 2
             answer.size() == ?1 )
        {
            Object first = answ

src2/java/main/org/jaxen/BaseXPath.java
 - changed source on line 691 (char index=24848) from 0 to 1


        return results.get( ?0 );
    }
}

lots more output...
src2/java/main/org/jaxen/BaseXPath.java
 - changed source on line 691 (char index=24848) from 0 to 1


        return results.get( ?0 );
    }
}



10 mutations survived out of 11 changes. Score = 10
took 1 minutes
```

As you can see from Listing 1, the `BaseXPath` class isn't very well tested. Jester made 11 changes to the class, and only one caused the tests to fail. Some of these are false positives, but surely I can do better than one out of 11.

The next step is to look at the code Jester mutated without breaking the test suite and see if you need to write a test for it. Jester shows you what it's changing in the GUI shown in Figure 1 (it can't run headless, annoyingly), prints the output on the console as shown in Listing 1, and generates an XML file containing the list of changes that didn't break anything, like the one shown in Listing 2:

Listing 2. Jester XML output

```
<JesterReport>
<JestedFile fileName="src2/java/main/org/jaxen/BaseXPath.java" absolutePathFileName=
"/Users/elharo/Documents/articles/jester/jaxen/src2/java/main/org/jaxen/BaseXPath.java"
numberOfChangesThatDidNotCauseTestsToFail="8" numberOfChanges="11" score="28">
<ChangeThatDidNotCauseTestsToFail index="7691" from="if (" to="if (true ||"/>
<ChangeThatDidNotCauseTestsToFail index="7691" from="if (" to="if (false &amp;&amp;"/>
<ChangeThatDidNotCauseTestsToFail index="7703" from="!=" to="=="/>
<ChangeThatDidNotCauseTestsToFail index="7754" from="==" to="!="/>
<ChangeThatDidNotCauseTestsToFail index="7757" from="1" to="2"/>
<ChangeThatDidNotCauseTestsToFail index="7826" from="if (" to="if (true ||"/>
<ChangeThatDidNotCauseTestsToFail index="7826" from="if (" to="if (false &amp;&amp;"/>
<ChangeThatDidNotCauseTestsToFail index="24749" from="if (" to="if (false &amp;&amp;"/>
</JestedFile></JesterReport>
```

Jester's line-number reports are often quite a ways off, so you're better off searching for the changed code in the console output. Here's a change from the report in Listing 1:

```
src2/java/main/org/jaxen/BaseXPath.java
 - changed source on line 691 (char index=24848) from 0 to 1


        return results.get( ?0 );
    }
}
```

This change turns out to be toward the end of the class, in this method:

```
protected Object selectSingleNodeForContext(Context context) throws JaxenException
{
  List results = selectNodesForContext( context );

  if ( results.isEmpty() )
  {
    return null;
  }

        return results.get( 0 );
}
```

A quick look through the test suite reveals that indeed no test was calling `selectSingleNodeForContext`. So the next step is to write a test for this method. The method is protected, so the test can't call it directly. Sometimes you need to write a subclass (often as an inner class) in order to test protected methods. But in this case a little grep soon revealed that this method was directly invoked by two other public methods in the same class, `stringValue` and `numberValue`. You might as well use both of these to test it:

```
public void testSelectSingleNodeForContext() throws JaxenException {

        BaseXPath xpath = new BaseXPath("1 + 2");

        String stringValue = xpath.stringValueOf(xpath);
        assertEquals("3", stringValue);

        Number numberValue = xpath.numberValueOf(xpath);
        assertEquals(3, numberValue.doubleValue(), 0.00001);

    }
```

The final step is to run the test case and make sure it passes. Here's the result:

```
java.lang.NullPointerException
 at org.jaxen.function.StringFunction.evaluate(StringFunction.java:121)
 at org.jaxen.BaseXPath.stringValueOf(BaseXPath.java:295)
 at org.jaxen.BaseXPathTest.testSelectSingleNodeForContext(BaseXPathTest.java:23)
```

Jester had caught a bug! The method didn't work as it was supposed to. Even more interestingly, an investigation of the bug revealed a potential design flaw. The `BaseXPath` class should probably be abstract rather than concrete. I swear I did not cherry-pick this example to expose this bug. I just started with `BaseXPath` because it was the first class in the top-level org.jaxen package, and I picked `selectSingleNodeForContext` as the method to test because it was the last error Jester reported. I really thought there was nothing wrong with the method, but I was wrong. If something isn't tested, assume it's broken. Jester tells you what's broken.

The next step is obvious: Fix the bug. (Be sure you fix it in both the copy of the source tree Jester is working on and in the actual tree.) Then, iterate -- rerun Jester on this class until it no longer survives any mutations, or until it's obvious that any mutations it does survive are irrelevant. After I added a test for (and fixed) this bug, Jester then reported that only eight out of 11 mutations went undetected, as shown in Listing 2. As is often the case in debugging, fixing one problem fixes (or reveals) several others.

Jester performance

Because Jester recompiles the code base and reruns the test suite for each change it makes, it runs orders of magnitude more slowly than more traditional tools like Clover. It's therefore important to pay some attention to performance. You can use a number of techniques to speed up Jester runs.

First, if compiling takes a significant fraction of Jester's execution time, try a faster compiler. Many users have reported noticeable speed-ups by using Jikes instead of javac (see Resources). You can change the compile command Jester uses in the jester.cfg file in Jester's main directory.

Second, profile and optimize your test suite. Normally you don't worry too much about how fast the unit tests take to run, but any savings can be significant when multiplied by the thousands of times Jester executes the test suite. In particular, look for issues in the test suite that don't arise in normal code. JUnit reinitializes all fields for each and every method executed, so pulling test data out of fields and into local variables can speed things up significantly when the fields aren't used by every method in the test class. If the resulting code duplication offends your sense of style, try splitting the test suite into smaller, more modular classes, in each of which all initial data is shared among all test methods.

Third, reorganize the test suite's `suite` method so that the most-fragile tests (the ones most likely to break after changes) are run before the less-fragile ones. As soon as Jester detects a single test failure, it aborts the run, so failing as early as possible can short-circuit a lot of time-consuming extra tests.

Fourth, for similar reasons, when tests are roughly equally likely to fail, put the fastest tests first. Sort the tests by rough execution time. Tests that execute purely in memory come before tests that access the disk, which come before tests that access the LAN, which come before tests that access the Internet. If some tests are particularly slow, try dropping them, even if this increases the number of false positives. In the test suite for XOM (an API for processing XML with the Java language), just a few of the almost 50 test classes count for well over 90 percent of the execution time. Removing these when jesting gives me a factor of 10 improvement in performance.

Finally, and most important, don't test the whole code base at once. Limit the tests to one class at a time, and run only the tests that are likely to expose gaps in the coverage of that one class. It might take marginally longer to test every class, but this way you can begin filling gaps and fixing bugs almost immediately, rather than waiting a few days for a Jester run to complete.

Summary

Jester is an important addition to the agile programmer's toolbox. It finds gaps in code coverage no other tool can, which translates directly into finding and fixing bugs. You'll produce more-robust software by testing a code base with Jester.

Resources

- Download Jester from SourceForge.
- Get test infected.
- JUnit is the de factor standard unit testing framework for Java code, and the one Jester relies on.
- The Jaxen project used as a guinea pig in this article is an open source XPath engine for Java language that's adaptable to many different object models.
- Clover is a more traditional test coverage tool that nicely complements Jester. It's somewhat easier to use than Jester and much faster; but it only tests that code is executed during tests, not that it's actually tested.
- EMMA is a free, open source code-coverage tool. Learn more about various open source unit testing tools and open source code coverage tools for Java programmers.
- Read Dave Thomas and Andy Hunt's *Pragmatic Unit Testing in Java With JUnit* (Pragmatic Bookshelf, 2003).
- Dennis M. Sosnoski kicks off a Classworking toolkit series by exploring the open source Hansel and Gretel code coverage tools.
- "Keeping critters out of your code: How to use WebSphere and JUnit to prevent programming bugs" (developerWorks, June 2003) by David Carew and Sandeep Desai looks at taking an XP approach to testing.
- Eric Allen and Roy Miller covered unit testing frequently in their respective columns, Diagnosing Java code and Demystifying Extreme Programming.
- Testdriven.com is a comprehensive collection of articles and resources about test-driven development.
- You'll find articles about every aspect of Java programming in the developerWorks Java technology zone.
- Browse for books on these and other technical topics.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Elliotte Rusty Harold

Elliotte Rusty Harold is originally from New Orleans, to which he returns periodically in search of a decent bowl of gumbo. However, he resides in the Prospect Heights neighborhood of Brooklyn with his wife Beth and cats Charm (named after the quark) and Marjorie (named after his mother-in-law). He's an adjunct professor of computer science at Polytechnic University where he teaches Java and object-oriented programming. His Cafe au Lait Web site at has become one of the most popular independent Java sites on the Internet, and his spin-off site Cafe con Leche has become one of the most popular XML sites. His books include *Effective XML*, *Processing XML with Java*, *Java Network Programming*, and *The XML 1.1 Bible*. He's currently working on the XOM API for processing XML and the XQuisitor GUI query tool.