Jester - a JUnit test tester.

Ivan Moore
Connextra Ltd
Studio 312, Highgate Studios
53-79 Highgate Road
London NW5 1TL, England
+44 (0)20 7692 9898
ivan@connextra.com

ABSTRACT
Extreme programmers have confidence in their code if it passes their unit tests. More experienced extreme programmers only have confidence in their code if they also have confidence in their tests. A technique used by extreme programmers to gain confidence in their tests is to make sure that their tests spot deliberate errors in the code. This sort of manual test testing is either time consuming or very superficial.

Jester is a test tester for JUnit tests; it modifies the source in a variety of ways, and checks whether the tests fail for each modification. Jester indicates code changes that can be made that do not cause the tests to fail. If code can be modified without the tests failing, it either indicates that there is a test missing or that the code is redundant. Jester can be used to gain confidence that the existing tests are adequate, or give clues about the tests that are missing.

Jester is different than code coverage tools, because it can find code that is executed by the running of tests but not actually tested. Jester will be compared with conventional code coverage tools. Results of using Jester will be discussed.

Keywords
unit testing, testing, JUnit, mutation analysis, error seeding, failure testing

1 INTRODUCTION

Extreme programmers[1] have confidence in code if it passes tests, and have confidence in tests if they catch errors. Many extreme programmers temporarily put deliberate errors in their code to check that their tests catch those errors, before correcting the code to pass the tests. In some project teams, a *project saboteur*[4] is appointed, whose role is to verify that errors that they deliberately introduce to a copy of the code base are caught by the tests. Jester performs similar test testing mechanically, by making some change to a source file, recompiling that file, running the tests, and if the tests pass Jester displays a message saying what it changed. Jester makes its changes one at a time, to every source file in a directory tree, making many different changes to each source file. The different types of change made are discussed later. Note that each change is undone before the next change is made, i.e. changes are made independently of each other.

Jester can modify not only the code that the tests are testing, but also the test code itself. If a test is modified but does not fail when run then the test may be redundant or erroneous.

Jester currently only works for Java code with JUnit[6] tests; the same approach could be used for other languages and test frameworks. Java and JUnit were chosen as the author uses both, and JUnit is probably the most widely used unit test framework by extreme programmers[3].

The ideas of mutation analysis[2] or automated error seeding[5] are not new, but Jester is able to be more widely applicable than other tools because of the widespread use of JUnit. In order for Jester, or any other similar tool, to work, it needs to be able to modify the source code, recompile that source code, and run the tests. Modifying the source code and recompiling it is quite straightforward. It is the ability to run the tests programatically that makes the use of JUnit so important both to the simple implementation of Jester, and its wide applicability.

2 MODIFYING AND RECOMPILING SOURCE CODE

Jester modifies Java source code in very simple ways, which do not require parsing or changes to more than one source file at a time.
The modifications are:

- modifying literal numbers; e.g. `0` is changed to `1`
- changing `true` to `false` and vice-versa
- changing `if(` to `if(true ||`
- changing `if(` to `if(false &&`

The last two have the effect of making the condition of the `if` statement always true or always false respectively. The reason for these replacements rather than the apparently simpler `if(true)` and `if(false)` respectively is to avoid needing to find the end of the condition, which would require some parsing and hence not be as simple to implement. There is no possibility of making two changes that cancel each other out as the changes are applied one at a time, being undone before the next change.

These simple modifications have been found to be quite effective, as shown later. More sophisticated modifications will be tried in future versions of Jester.

To recompile the modified source code, Jester uses `Runtime.getRuntime().exec("javac ...` to invoke the java compiler. This means that the java compiler is not running directly in java but in the underlying operating system. Jester could potentially be modified to work for other languages by changing the compiler that it invokes.

3 RUNNING THE TESTS

Jester uses a modified version of the class `textui.TestRunner` included in JUnit 3.2. This has been modified to simply print `PASSED` or `FAILED` having run the tests; no other details are needed by Jester. In order to run the tests using the modified classes, Jester uses `...exec("java jester.TestRunnerImpl ...` to run the tests in a new instance of a java virtual machine, using the modified test runner. The tests cannot simply be run directly by Jester in the same virtual machine that it is running in, as once a class has been loaded into a Java virtual machine it will not be replaced by simply recompiling its source, and Jester requires that the modified code is executed; there would be no point having Jester running the tests if they were not executing the modified code.

To use Jester for another language or another test framework would require the existence of a test runner that can be executed in the same way, and give the expected result of `PASSED` or `FAILED`.

4 USING JESTER

Jester needs to know the JUnit test class (a subclass of TestCase that can be used by the JUnit TestRunners) and the directory that contains the source code that Jester changes to try to find if the tests are not covering those changes.

The test class is the one that is expected to show up any changes to code in the source directory. Typically, this test class would be the TestAll class of a package, and the source directory would be the subdirectory that contains the code being tested by that TestAll class.

For any change that Jester was able to make without the tests failing, it prints the name of the file changed, the position in the file of the change, and some of the original source file from roughly 30 characters before to 30 characters after so that the change can be easily identified within the source file.

5 RESULTS

The results of applying Jester to a small but interesting test suite and associated code is presented here, followed by observations of the use of Jester on a larger amount of code for a publicly available product.

The Money samples of JUnit 3.2 give a small example of how to use JUnit. There is an interface IMoney and two classes that implement the interface, Money and MoneyBag, and a test class MoneyTest which includes tests for both Money and MoneyBag. Including comments, there are about 400 lines of code in total.

Jester made 47 separate modifications (including to the test class itself), of which, 10 did not make the tests fail; i.e. they were changes that indicated *possible missing tests* or *redundant code*. This was a much higher percentage than expected. The version of Jester used ignores comments; a version which included modifications to comments made many more modifications, revealing comments that included literal numbers, which is of debatable usefulness. As each modification requires recompilation and running all the tests this took a long time to run considering the amount of code. On a Pentium 133MHz (an old machine) Jester took 12 minutes to complete the run.

The 10 modifications that did not cause the tests to fail will now be described. Three of the modifications were in the `equals` method of the Money class:

```
public boolean equals(Object anObject)
{
 if (isNull())
   if (anObject instanceof IMoney)
     return
((IMoney)anObject).isNull();
 ...
```

Jester reported that:

- `if (isNull())` can be replaced by
  `if (false && isNull())`

- `if (anObject instanceof IMoney)` can be replaced by
  `if (true || anObject instanceof IMoney)`

- `if (anObject instanceof IMoney)` can be replaced by
  `if (false && anObject instanceof IMoney)`

The first of these shows that either `isNull()` is always `false` in the tests (hence there is a test missing for the case where `isNull()` is `true`), or, it could show that it makes no difference to the running of the tests whether the `isNull()` branch is executed. In fact, there is no test of `equals` for a '`null`' Money. (The method `isNull` has been renamed `isZero` in JUnit 3.4). Without further examination of the code, the possibility that the branch of code does not make any difference to the correct running of code should not be discounted. This could happen if the `isNull()` branch was a behaviour neutral optimization.

The second and third modifications reported indicate that either that `if` statement is not executed, or it makes no difference to the running of the code, i.e. it doesn't matter whether the value of the condition is `true` or `false`. In this case, this code is not executed by the tests. A conventional code coverage tool would be able to spot this.

The other 7 modifications that Jester made which did not cause the tests to fail were all for the MoneyBag class. One of these was changing the construction of a vector from `new Vector(5)` to `new Vector(6)`. This had no effect on the correct running of the code, as the effect of this number is on the initial internal size of the constructed vector, which can have an effect on performance but does not effect the vector's behaviour. Another change was to modify the `hashCode` value of an empty MoneyBag. This has no effect on the correct running of the code, and can be considered a 'false hit' by Jester.

Three of the other modifications are similar to those for Money; they show that the `equals` method is not tested for '`null`' MoneyBags.

The remaining two modifications both relate to the `equals` method for the special case that two MoneyBags that contain a different number of Money objects are not equals. There are no unit tests for this special case code.

Jester has also been applied to parts of Sidewize[7], a browser companion built by Connextra. It successfully identified where tests were missing, and where code had become redundant and needed removing. However, it took considerable analysis of the results to identify whether the modifications reported by Jester represented missing tests, redundant code, or were simply 'false hits', i.e. represented behaviour preserving changes to the code.

6 COMPARISON WITH CODE COVERAGE TOOLS

Code coverage tools indicate which code is not executed by the test suites. This can be very useful for indicating either redundant code or missing tests, and in some of the cases described above would be simpler to understand than the results from Jester. However, tests can cause code to execute even if its results are not checked, which means that code coverage tools can easily miss important test ommissions. For example, in the testing of the `equals` method of MoneyBag, it would have been very easy to have missed out a test for equality with something other than another MoneyBag. A code coverage tool might not indicate this missing test, because the missing test is not revealed by a branch of an `if` statement not being executed but rather by that branch always being executed (some code coverage tools can be used to spot this). Furthermore, Jester can give more of a clue about the sort of test that is missing, by showing how the code can be modified but still pass the tests.

7 CONCLUSIONS

Jester can reveal code that has not been tested, or is possibly redundant. However, Jester takes a long time to run, and the results take some manual effort to interpret. Nevertheless, in comparison to a code coverage tool, Jester can spot untested code even if it is executed.

The value of using Jester is the benefit from the discovery of missing tests or redundant code minus the cost of using it. The cost of using Jester is the time it takes to run (mostly machine time) plus the time to interpret its results (developer time). The cost of missing tests can be enormous if there are bugs that would otherwise have been found. Redundant code can also be expensive because it wastes developers' time whenever it is read or modified (for example, to keep it compilable). Therefore, Jester's net value depends upon the state of the code that it is used on.

Jester uses a simple text based find-and-replace style approach to modifying the original source code. This was simple to implement, and has proven adequate so far. However, if Jester were to use a parsed representation of the source code (either using a parser on the source code or possibly working on class files) then more sophisticated modifications, and better reporting of its modifications, would be made easier to implement. Using a parsed representation would, for example, allow Jester to remove complete statements from methods, and to report its changes per method rather than by character index.

Jester is publicly available on a 'free software' licence[8] and efforts will continue to improve Jester, in particular to provide results that are easier to interpret and to try to avoid 'false hits'.

ACKNOWLEDGEMENTS
Many thanks to all those at the eXtreme Tuesday Club in London (England), for testing Jester and for comments on early drafts of this paper; in particular, special thanks to Keith Braithwaite, Paul Simmons, Duncan Pierce, Tamara Galloway and Tim Bacon. Also thanks to Connextra for providing a useful testing ground for Jester.

REFERENCES

1. Kent Beck. eXtreme Programming Explained: Embrace Change. Addison-Wesley, 1999.
2. Michael A. Friedman and Jeffrey M. Voas. Software assessment: reliability, safety, testability. New York: John Wiley & Sons, Inc., 1995.
3. Erich Gamma and Kent Beck. Test infected: programmers love writing tests. *The Java Report*, 3(7):37-50, July 1998.
4. Andrew Hunt and David Thomas. The Pragmatic Programmer. Addison-Wesley, 1999.
5. Brian Meek and K.K.Siu. The effectiveness of error seeding. *ACM Sigplan Notices*, Vol 24 No 6, pp81-89, June 1989.
6. http://www.junit.org
7. http://www.sidewize.com
8. http://www.jesterinfo.co.uk