# Automated Unit Testing with *Randoop, JWalk* and *µJava* versus Manual *JUnit* Testing

Nastassia Smeets[1] and Anthony J H Simons[2]

[1] Department of Mathematics and Computer Science, University of Antwerp, Prinsstraat 13, BE-2000, Antwerpen, Belgium.
`Nastassia.Smeets@student.ua.ac.be`
[2] Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP, UK.
`A.Simons@dcs.shef.ac.uk`

**Abstract.** A comparative study was conducted into three Java unit-testing tools that support automatic test-case generation or test-case evaluation: *Randoop, JWalk* and *µJava*. These tools are shown to adopt quite different testing methods, based on different testing assumptions. The comparative study illustrates their respective strengths and weaknesses when applied to realistically complex Java software. Different trade-offs were found between the testing effort required, the test coverage offered and the maintainability of the tests. The conclusion evaluates how effective these tools were as alternatives to writing carefully hand-crafted tests for testing with *JUnit*.

## 1 Introduction

Software testing is an essential part of ensuring software quality and verifying correct behaviour. However, writing software tests is often an uninspiring and repetitive task that could benefit from automation, both in the selection and execution of test cases and in the evaluation of test results. Automatic tools may save time and effort, by generating tests that exercise the software more thoroughly than hand-crafted tests and may even find faults that human testers would never think of checking. Yet, how should the overall quality of an automatically-generated test set be judged against a carefully thought out hand-crafted test suite?

This paper reports the result of an experiment in comparing three radically different Java testing tools: *Randoop* [1], *JWalk* [2] and *µJava* [3]. Section 2 outlines the different testing assumptions and methods followed by each tool, and develops a qualitative framework for mutual comparison. Section 3 describes how each tool performed on *JPacman* [4], a realistically complex Java software system, for which comprehensive *JUnit* [5] tests also exist. Section 4 concludes with a summary of findings and evaluates the relative strengths and weaknesses of each tool, compared against hand-crafted testing using *JUnit*.

## 2 Testing Tools and Techniques

Three different testing techniques (and their associated tools) were chosen for the comparison. They were chosen to be as unalike as possible, to reveal possibly interesting contrasts. Each testing tool employs a different testing approach and is based on different underlying testing assumptions.

### 2.1 Feedback-directed Random Testing with *Randoop*

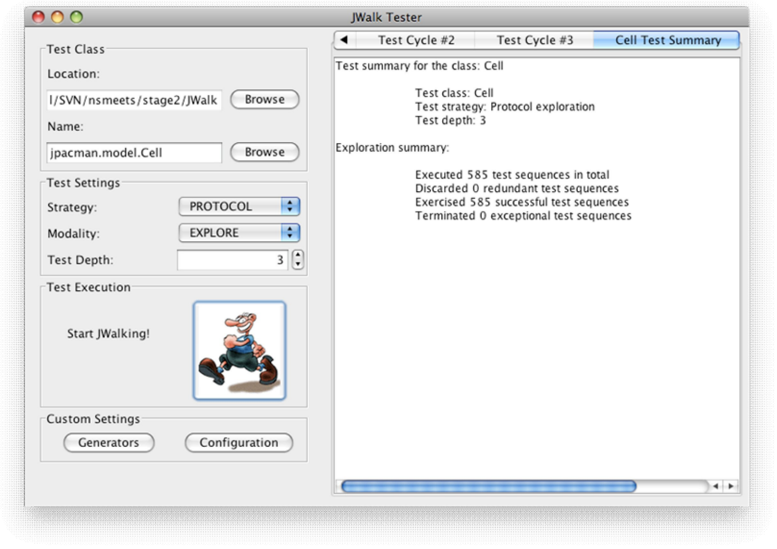
The *Randoop* tool [1] generates random tests for a given set of Java classes during a limited time interval, which is preset by the tester. The test engine uses Java's ability to introspect about each class's type structure and generates random test sequences made up from constructors and methods published in each test class's public interface. The testing philosophy is based on random code exploration, which is expected at the limit to reveal (possibly) all salient properties of the tested code. *Randoop* has successfully discovered long-buried bugs in Microsoft .NET code [6] and detected unexpected differences between Java versions on different platforms [7].

In order to check meaningful semantic properties, the tool requires some prior preparation of the code to be tested [7]. Firstly, classes under test (CUTs) are instrumented with Java annotations, to guide the tool in how to use certain methods during testing. For example, `@Observer` directs the tool to treat the result of a method as a state observation, whereas `@Omit` directs the tool to ignore non-deterministic methods and `@ObjectInvariant` directs the tool to treat a method as a state validity predicate. Given this information, the tool may construct meaningful regression tests that observe salient parts of the CUT's state. Secondly, the tool recognizes certain *contract-checking interfaces* as special. Testers may optionally supply contract-checking classes, which implement these interfaces, and so are treated as test oracles, which are executed upon the randomly-generated objects. Predefined contracts typically check the algebraic properties of the CUT, such as the idempotency of `equals()`, or the consistency of `equals()` and `hashCode()`, but testers may in principle supply their own arbitrary contracts [7] (but see 3.1).

The feedback-directed aspect refers to the tool's ability to detect certain redundant test sequences and prune these from the generated set. In practice, this refers to test sequences which extend sequences that are already known to raise an exception [1]. Since it makes no sense to extend a terminating sequence, the longer sequences are pruned. Retained tests are exported in the format expected by *JUnit* [5]. *Randoop* classifies all randomly generated tests into *regression tests* (which pass all contracts) and *contract violations* (which do not) and discards tests which raise exceptions.

### 2.2 Lazy Systematic Unit Testing with *JWalk*

The *JWalk* tool [2] generates bounded exhaustive test sets for one CUT at a time, using specification-based test generation algorithms that verify the detailed algebraic structure, or high-level states and transitions of the CUT. The tool constructs a test oracle incrementally, by a combination of dynamic code analysis and some limited interaction with the tester, who confirms or rejects certain key test results. Once the oracle is constructed, testing is fully automatic. If the software, and implicitly its specification, is subsequently modified, *JWalk* only prompts to confirm altered properties. However, the tool does not perform *regression testing* so much as complete *test regeneration* from the revised specification [8]. This explains the *lazy* epithet, whereby a specification evolves in step with changes made to the code and stabilizes "just in time" before testing (c.f. lazy evaluation). The tool could prove a useful addition to agile or XP development, which expects rapid code change and otherwise has no up-front specification to drive the selection of test cases [9].



Test Class
Location: l/SVN/nsmeets/stage2/JWalk
Name: jpacman.model.Cell

Test Settings
Strategy: PROTOCOL
Modality: EXPLORE
Test Depth: 3

Test Execution

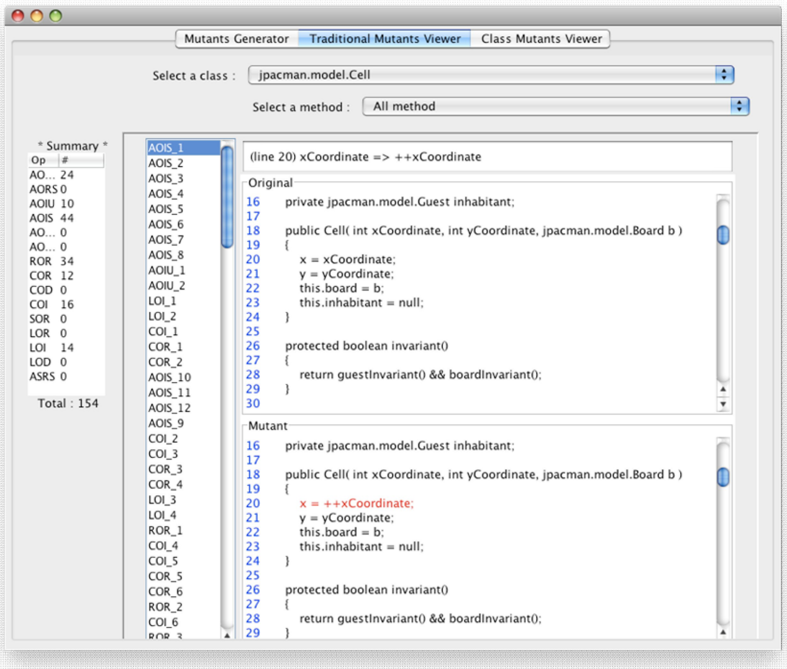
```
Test summary for the class: Cell

            Test class: Cell
            Test strategy: Protocol exploration
            Test depth: 3

Exploration summary:

            Executed 585 test sequences in total
            Discarded 0 redundant test sequences
            Exercised 585 successful test sequences
            Terminated 0 exceptional test sequences
```

**Fig. 1.** JWalk Tester being used to exercise all methods of the class *jpacman.model.Cell* to a depth of 3, in all distinct interleaved combinations. The test summary tab displays the number of tests executed. Obscured tabs for each test cycle to depths 0, 1, 2 and 3 list all test sequences and test outcomes

In contrast with *Randoop*, test sequence generation is entirely deterministic, based on a filtered, breadth-first exploration of the CUT's constructors and methods, since *JWalk* exploits regularity to predict test equivalence classes, so reducing the number of examples to be confirmed by the tester. Like *Randoop*, *JWalk* prunes longer test sequences whose prefixes are known to raise exceptions. Furthermore, *JWalk* can also prune sequences whose prefixes end in *observers* (which do not alter the CUT's state) and whose prefixes end in *transformers* (which return the CUT to an earlier state). The tool detects these algebraic properties of methods automatically and does not require any kind of prior code annotation [10]. It retains the shortest exemplars from each equivalence class and uses these to *predict* the results of longer sequences.

The *JWalk* tool can be run in different modes which *inspect* the CUT's interface, which *explore* (basically, exercise) the CUT's methods, or which *validate* the method results with respect to a test oracle. Likewise, test sequences may be constructed following different test strategies, ranging from breadth-first *protocol* exploration (all interleaved methods), to smarter *algebraic* exploration (all primitive constructions and observations), to high-level *state-space* exploration (all high-level states and transitions). *JWalk* regenerates and executes all tests internally, producing test reports [2] and does not export *JUnit* tests for execution outside the tool. Following the testing philosophy, exported regression tests would progressively lose their ability to cover the state-space at a geometric rate, as the tested software evolved [8].

### 2.3 Mutation Testing with *μJava*


Mutants Generator | Traditional Mutants Viewer | Class Mutants Viewer

Select a class : jpacman.model.Cell

Select a method : All method

* Summary *

| Op | # |
|---|---|
| AO... | 24 |
| AORS | 0 |
| AOIU | 10 |
| AOIS | 44 |
| AO... | 0 |
| AO... | 0 |
| ROR | 34 |
| COR | 12 |
| COD | 0 |
| COI | 16 |
| SOR | 0 |
| LOR | 0 |
| LOI | 14 |
| LOD | 0 |
| ASRS | 0 |

Total : 154

```
AOIS_1
AOIS_2
AOIS_3
AOIS_4
AOIS_5
AOIS_6
AOIS_7
AOIS_8
AOIU_1
AOIU_2
LOI_1
LOI_2
COI_1
COR_1
COR_2
AOIS_10
AOIS_11
AOIS_12
AOIS_9
COI_2
COI_3
COR_3
COR_4
LOI_3
LOI_4
ROR_1
COI_4
COI_5
COR_5
COR_6
ROR_2
COI_6
ROR_3
```

(line 20) xCoordinate => ++xCoordinate

Original

```
    private jpacman.model.Guest inhabitant;

    public Cell( int xCoordinate, int yCoordinate, jpacman.model.Board b )
    {
        x = xCoordinate;
        y = yCoordinate;
        this.board = b;
        this.inhabitant = null;
    }

    protected boolean invariant()
    {
        return guestInvariant() && boardInvariant();
    }
```

Mutant

```
    private jpacman.model.Guest inhabitant;

    public Cell( int xCoordinate, int yCoordinate, jpacman.model.Board b )
    {
        x = ++xCoordinate;
        y = yCoordinate;
        this.board = b;
        this.inhabitant = null;
    }

    protected boolean invariant()
    {
        return guestInvariant() && boardInvariant();
    }
```

**Fig. 2.** The μJava tool, seeding mutations in the class *jpacman.model.Cell*. The traditional mutants viewer tab displays the original source (above) and modified source (below), in which the mutant AOIU_1 (arithmetic operator insertion, unary) has been introduced. Obscured tabs include the mutant generator and the class mutants viewer

The *μJava* tool [3] assumes the prior existence of a test suite, created by some other means, for testing multiple classes in a package. Mutation testing is intended to evaluate the quality of the test suite by making small modifications, or *mutations*, to the tested source code and determining whether the tests can detect these mutants. A test which detects a mutation is said to "kill" the mutant; and a successful test may kill more mutants than weaker tests, whereas a mutant that is never detected may reveal the need for more tests, or that the mutation is benign, or that it cancels out another mutation ("equivalent mutants" [11]), or simply that the code branch containing the mutant has not been covered by any of the tests.

The testing philosophy is purely code-based, in contrast to the specification-based or regression-based approaches of the other two methods. The testing method assumes, in the limit, that mutations will mimic all possible coding errors, by introducing every possible kind of fault into the original source code. Tests which kill the mutants will also reveal unplanned errors in the source code. Twelve traditional mutants include variable and value insertions, deletions, substitutions, operator replacement and similar changes to code at the method-level [12]. Twenty-nine special mutants are devised to handle class-level mutation, based on a survey of common coding errors, such as access modifier changes, insertion and hiding of overrides, changes to member initialization, specialization or generalization of types and typecasts, and insertion and deletion of keywords `static`, `this` and `super` [13].

The testing regime also requires test harness classes, which encapsulate the hand-crafted test suites to be run after every mutation series. These are ordinary classes whose methods are named: `test1()`, `test2()`, etc. and which return strings. The testing tool is able to compare string-results before and after mutation, to determine which tests were affected by the mutations. Though similar in essence, these test suites are not in the same format as that expected by *JUnit*.

## 2.4 Experimental Framework and Hypothesis

The experiment outlined above was to investigate how well each of the three testing tools performed, compared to an expert human tester. The hypothesis was that the tools might speed up certain aspects of testing, but not fully replace a human tester, although they might make his/her job somewhat easier.

The software chosen for the test target was *JPacman* [4], a Java implementation of the traditional *Pacman* arcade game, developed at TU Delft as a teaching example. The source consists of 21 production classes, amounting to 2.3K SLOC [13]. The package is attractive, since it comes with a comprehensive *JUnit* test suite, consisting of 60 test cases (15 acceptance tests for the GUI; 45 unit tests for the classes). These were carefully hand-crafted, following Binder's guidelines [14] to cover all code branches, populate both on-boundary and off-boundary points, test class diagram multiplicities and test class behaviour from decision tables and finite state machines. The test code amounts to about 1K SLOC.

It was difficult to find a framework in which the three disparate testing approaches could be compared systematically. We considered using code coverage as an unbiased, verifiable criterion, easily measured using the tool *Emma* [15]. However, *µJava* does not actually generate unit tests and *JWalk*'s tests were not easily accessible to *Emma*. The mutation scores of *µJava* might provide a rough derived indication of how much of the code was covered by the hand-crafted tests. Nonetheless, simply counting how much code executed was not a very sophisticated measure of test effectiveness (was the tested code actually correct?)

Time spent on testing was also considered, since one of the main benefits of automated testing is to reduce this overhead. Unfortunately, no measurements were available for the time spent on constructing the hand-crafted *JUnit* tests; against this, it was hard to estimate impartially the time taken to learn the different testing tools (reading online guides and papers). Simply measuring the time taken to run the tests was also considered a poor indicator of how effective the tests were, and was also biased, since *Randoop* generates tests up to a preset time limit.

For this reason, the comparison between the testing methods is eventually qualitative, focusing mostly on the types of fault, or change, that the tools could detect. Further points of distinction were the amount of extra work the tester had to invest when using the tools; and the maintainability of any further generated software.

## 3 Testing Experiments

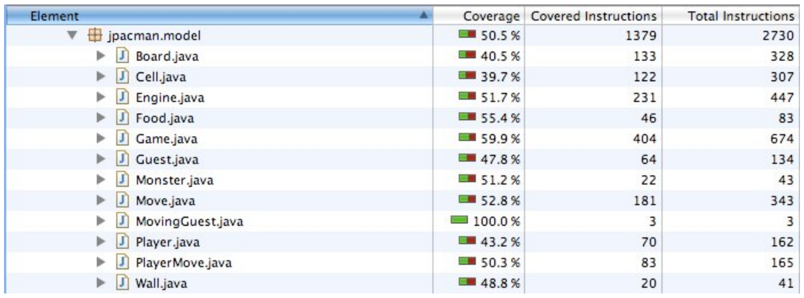
Each of the tools *Randoop*, *JWalk* and *μJava* were downloaded and installed. They were used to test classes from the `jpacman.model` package, part of the *JPacman* [4] software testing benchmark developed by TU Delft. Comparisons were drawn as detailed above; qualitative code coverage and time estimates are also given, where systematic measures were not otherwise readily available.

### 3.1 Randoop Testing Experiment

Easy both to install and to use, the *Randoop* tool randomly generates interleavings of methods with randomized input values, bounded by time, not depth. The tool was executed in its default configuration (using the built-in contract checkers), setting time limits of 1 second, 10 seconds and 30 seconds. The initial experiment aimed to discover how thoroughly the *JPacman* model classes were exercised purely by random execution of the `jpacman.model` package. Coverage measures were extracted using *Emma* for the regression test suite generated by *Randoop* (viz. disregarding the contract violation suite). When using a setting of 10 seconds, the generated tests exercised 49.2% of the code, which only increased slightly to 50.5% using the longer setting of 30 seconds. A shorter setting of 1 second only covered 39.9% of the same package (see fiig. 3), possibly indicating that 30 seconds was a useful limit.

One of the attractions of *Randoop* is the automated generation of a full regression test suite [1]. This can be helpful when working on a project that has no tests whatsoever and for which it is crucial to preserve current behaviour when making changes to the code. However, the kinds of property preserved by these tests were those determined randomly by the tool, arbitrary observations that happened to hold when *Randoop* was executed, rather than specific semantic properties of the application. If an application is faulty, there is no way of knowing whether these tests monitor correct behaviour; they merely grant the tester the ability to detect when the behaviour later changes.

A potential disadvantage of *Randoop* was the high cost of test code maintenance. The regression tests generated in 30 seconds ran to some 96K lines of test code, comprising 400 test cases. Some of these used just under 100 variables, with opaque generated names like `var1`, `var2`, … `varN`, making any deep understanding of the test programs impossible. Another limitation of *Randoop* was the inability to control values supplied as arguments to tests (which were randomized), making it impossible to guarantee equivalence partition coverage on inputs.



| Element | Coverage | Covered Instructions | Total Instructions |
|---|---|---|---|
| jpacman.model | 50.5 % | 1379 | 2730 |
| Board.java | 40.5 % | 133 | 328 |
| Cell.java | 39.7 % | 122 | 307 |
| Engine.java | 51.7 % | 231 | 447 |
| Food.java | 55.4 % | 46 | 83 |
| Game.java | 59.9 % | 404 | 674 |
| Guest.java | 47.8 % | 64 | 134 |
| Monster.java | 51.2 % | 22 | 43 |
| Move.java | 52.8 % | 181 | 343 |
| MovingGuest.java | 100.0 % | 3 | 3 |
| Player.java | 43.2 % | 70 | 162 |
| PlayerMove.java | 50.3 % | 83 | 165 |
| Wall.java | 48.8 % | 20 | 41 |

**Fig. 3.** Instruction coverage (basic bytecode blocks) computed by *Emma*, for the methods of all production classes contained in the *jpackman.model* package, for all regression tests generated in 30 seconds by *Randoop* in its default contract-checking configuration

One potential way to force testing of explicit semantic properties was to use the annotation mechanism (see 2.1). We found that `@Observer` and `@Omit` annotations could be added to the tested source code and processed by *Randoop*; however, potentially the most useful annotation was `@ObjectInvariant`, since the *JPacman* source code already contained many class `invariant()` methods, which we could have identified for the tool. Unfortunately, the distributed version of the tool we obtained did not appear to process this annotation properly. Ironically, the *JPacman* source code already had many conditionally-compiled Java `assert()` statements which could be checked by "turning on assertions" at compile-time, but which threw fatal exceptions when violated, terminating the test run. One future suggestion for *Randoop* is that it could be made to detect Java `assert()` statements and convert these into invariants that could be automatically checked by the tool. We imagine these could be copied from the source and inserted as the code body of check-methods in automatically synthesized contract-checkers. We attempted to find out how to write custom contract-checkers, but the software distribution contained no explicit documentation on how to do this. *Randoop* is available from the project website [16].

## 3.2 JWalk Testing Experiment

*JWalk* version 1.0 was easy to install and understand, greatly helped by the comprehensive instructions given on the project website [17]. The distribution contained two testing tools and a toolkit for integrating *JWalk*-style testing with other applications, such as a Java editor. We used the standalone *JWalkTester* tool, which uploads and tests one CUT at a time. The tool could be run in one of three test *modes*, coupled with one of three test *strategies*.

The simplest mode was the *inspect*-mode, which merely described the public interface of the CUT, listing its public constructors and methods. When coupled with the *algebra*-strategy, this also described the algebraic structure of the CUT, identifying *primitive, transformer* and *observer* methods. When coupled with the *state*-strategy, this also identified the high-level states of the CUT, where these could be determined from state predicates normally provided by the CUT (e.g. `isOccupied()` for the class `jpacman.model.Cell`). The tool executed the CUT silently for 1-3 seconds to discover state and algebra properties by automatic exploration.

In the *explore*-mode, the tool exercised the CUT's constructors and methods to a chosen depth, displaying the results back to the tester in a series of tabbed panes, organized by test cycle, corresponding to sets of sequences of increasing length. When coupled with the *protocol*-strategy (see fig. 1), exhaustive sequences were constructed starting with every constructor, followed by all distinct interleavings of methods. The test reports for the protocol exploration strategy were rather too long to review by hand for CUTs with > 10 methods; fig.1 gives the test summary for `jpacman.model.Cell`, a class with one constructor and eight methods, which produced 585 distinct test cases, and did not benefit from any pruning of sequences due to prefix exceptions, since the code did not throw any exceptions.

**List. 1.** Test summary for *jpacman.model.Cell* generated by JWalk in exploration-mode, using the algebraic testing strategy and with the maximum depth set to 3

```
Test summary for the class: Cell
        Test class: Cell
        Test strategy: Algebraic exploration
        Test depth: 3
Exploration summary:
        Executed 9 test sequences in total
        Discarded 576 redundant test sequences
        Exercised 9 successful test sequences
        Terminated 0 exceptional test sequences
```

When coupled with the *algebra*-strategy, some seriously effective pruning took place (list. 1). The retained tests consisted of set-up sequences of *primitive* operations (constructors and certain irreducible methods), followed by any single method of interest, such as a programmer might wish to write by hand. The pruning rules eliminated redundant sequences whose prefixes contained *observers* (access methods that did not modify the CUT's state) or *transformers* (methods returning the CUT to a state that had already been visited). The presented test sets were far shorter and much easier to review. The summary report illustrates how only 9 tests of the 585 created by breadth-first exploration were eventually retained for their discriminating value.

When coupled with the *state*-strategy, exploration drove the CUT into each of its identified high-level states, then constructed sequences corresponding to the switch-1, switch-2 etc. transition coverage, up to the chosen depth. CUTs with no natural state predicates had one *Default* state (tests were then identical to protocol exploration). CUTs with predicates generated varying numbers of high-level states, corresponding to the product of Boolean outcomes of the predicates [2]. The illustrated class had four states: *Default, GuestInvariant, Occupied* and *GuestInvariant&Occupied*.

The third and last mode was the *validate*-mode, in which *JWalk* created a test oracle interactively (list. 2). Here, the tool prompted the tester to confirm or reject the outcomes of selected sequences by a single button click in a pop-up window. When compared with the time taken to write similar unit tests, this was very quick: scanning a sequence and deciding whether the outcome was correct typically took 1-2 seconds. However, the process of being continually presented with such sequences was eventually wearing, and potentially tedious when testing to any great depth. However, this was mitigated somewhat when *JWalk* predicted the outcomes of longer sequences using rules derived from learning the CUT's algebraic structure. Once an oracle had been generated, modifying the code resulted in the tool prompting only for novel combinations of methods, or sequences with altered outcomes, to be verified. So, the testing effort was only substantial in the first validation run, making this method scalable. *JWalk* could make more productive use of exceptions thrown by the CUT than *Randoop* – the tool could learn the difference between a fault and a violated precondition that was part of the CUT's specification. Validation could be combined with any of the three test strategies (*protocol, algebra, state*) described above.

**List. 2.** Oracle learned for *jpacman.model.Cell* generated by JWalk in the validation mode, using the algebraic strategy, to a maximum depth of 2

```
new(1,2,Board#1)=Cell#1
new(1,2,Board#1).getX()=2
new(1,2,Board#1).getY()=1
new(1,2,Board#1).guestInvariant()=true
new(1,2,Board#1).getInhabitant()=null
new(1,2,Board#1).isOccupied()=false
new(1,2,Board#1).cellAtOffset(5,6)=null
new(1,2,Board#1).getBoard()=Board#1
new(1,2,Board#1).adjacent(Cell#1)=false
```

A downside to using this tool was where the automatically synthesized argument parameters turned out to be useless for exhaustively testing a CUT that needed to be set up in a specific way. *JWalk* has a default strategy for synthesizing constructor and method arguments based on choosing values from monotonically-increasing sequences [2]. This helps to grow distinct algebraic structures [10], but proved to be quite unintelligent in cases that required careful argument selection. For example, the tool repeatedly generated a game board of dimension 1 x 2, then attempted to retrieve the cell at location (3, 4), which raised the same exception again and again. For this application, we were obviously interested in the correct retrieval of cells from their positions on the board, which required a specific set-up of the classes involved.

Fortunately, this could be addressed by implementing a custom generator (list. 3). This is a tester-supplied class, conforming to the `CustomGenerator` interface, which takes over control of how to synthesize values of given types. The illustrated generator produces `int` values in a custom order that allows game boards and cells to be constructed and retrieved more intelligently, when they are exercised by *JWalk*. Clear instructions are given how to do this on the website [17].

**List. 3.** Custom generator class created by the tester. This synthesizes integer argument values in a predetermined order, suitable for exercising *jpacman.model.Board* more effectively than using the default integer sequence

```
import org.jwalk.GeneratorException;
import org.jwalk.gen.CustomGenerator;
import org.jwalk.gen.MasterGenerator;

public class BoardGenerator
                    implements CustomGenerator {
  private int count = 0;
  private int[] values =
        {5, 10, 3, 5, 0, 0, 4, 9, -1, -1, 5, 10};

  public Boolean canCreate(Class<?> type) {
    return type == int.class;
  }

  public Object nextValue(Class<?> type)
                throws GeneratorException {
    if (type != int.class)
      throw new GeneratorException(type);
    if (count == values.length)
      count = 0;
    return Integer.valueOf(values[count++]);
  }

  public void setOwner(MasterGenerator generator) {
  }  // nullop
}
```

The intention is that all special set-up arrangements can be handled using custom generators. However, we felt that this was not the most elegant solution – it requires at least as much skill as that needed to write hand-crafted *JUnit* tests, since the tester must be aware of equivalence partitions and the order in which values are synthesized. So, we would not recommend this testing method for CUTs that require equivalence partition testing to achieve significant coverage of their behaviour (although *JWalk* sometimes covered equivalence partitions accidentally, by virtue of choosing different values for arguments in longer method sequences – see also 4.1). One future suggestion for *JWalk* is that it could be made to explore the space of arguments to each method, trying to discover which values qualitatively altered the behaviour of the CUT (c.f. equivalence partition testing). This would have to be done carefully, so as not to increase the burden of confirmations on the tester.

However, we do find *JWalk* extremely useful for testing all possible interleavings of methods, and for identifying the minimal complete test sets needed to exercise all the states and transitions of the CUT. *JWalk* calculates the required test sets in a fraction of the time that it takes to do this manually; so the testing effort for using this tool is very low.

## 3.3 μJava Testing Experiment

Version 3 of *μJava* was downloaded from the project website [18] and installed, though not without some difficulty, since the instructions were quite hard to follow and occasionally omitted details. A somewhat clearer guide was found in the appendices of [11]. Notwithstanding the website's claim that the software had been upgraded to support versions of Java later than 1.4, we encountered problems when using the tool with the *JPacman* source code. Very little diagnostic output was provided to the tester, apart from a generic "cannot parse" message, when the tool encountered unfamiliar syntax. After much investigation, we determined that *μJava* could not handle the syntax for generic classes (this issue is mentioned on the website), nor could it handle Java 1.5 annotations. The parser had been objecting to the labels `@Observer`, `@Omit` etc., with which we had previously annotated the *JPacman* sources. Furthermore, the parser did not recognize `assert()` statements, with which the source code was liberally sprinkled! All source files had to be re-written to eliminate assertions and conform to Java 1.4 syntax.

Fortunately, *μJava* proved fairly easy to work with after this. First, mutations were introduced into the source code. All possible mutants were selected (both traditional statement mutants and the special class syntax mutants) and introduced into all classes in the `jpacman.model` package. The user interface (see fig. 2) allowed the tester to compare the original and mutated versions of the source code. In the illustrated example, an introduced unary increment has been applied to a variable, in the expectation that this might later cause tests to fail. A complete count of the numbers of mutants introduced for each class is shown in table 1.

**Table 1.** All classes in the *jpacman.model* package, associated with their numbers of method- and class-level mutants that were generated and inserted into the source by μJava.

| **Class** | **Method-level** | **Class-level** |
|---|---|---|
| Board | 167 | 9 |
| Cell | 154 | 15 |
| Engine | 81 | 5 |
| Food | 28 | 1 |
| Game | 147 | 9 |
| Guest | (abstract) | (abstract) |
| Monster | 0 | 0 |
| Move | (abstract) | (abstract) |
| MovingGuest | (abstract) | (abstract) |
| Player | 49 | 5 |
| PlayerMove | 36 | 9 |
| Wall | 0 | 0 |

For the tests, we wanted to rely on the hand-crafted *JUnit* tests supplied with the *JPacman* distribution. However, since *μJava* predates *JUnit* by some years, the expected format of the tests was slightly different, so these also had to be rewritten. Whereas *JUnit* expects tests to assert *Boolean* properties and catches test failures using Java's exception-handlers, *μJava* expects tests to return an indicative *String*, which is used when comparing the results of tests on the original and mutated code. This process is known as "killing mutants"; and was conducted using the μJava GUI, which allowed the tester to select a CUT and run its associated unit test suite. Now, despite the fact that *JPacman* supplies some 45 unit tests, not every class in the `jpacman.model` package had an associated unit test suite, resulting in automatic zero mutant killing scores for some classes. This is more a reflection on the design of the hand-crafted tests, than it is on *μJava*.

**Table 2.** Percentages of mutants killed for classes in the *jpacman.model* package that had an associated hand-crafted unit test suite in *JPacman*, contrasted with traditional block coverage scores for the same hand-crafted unit tests

| Class | Method-level | | | Class-level | | | Coverage |
|---|---|---|---|---|---|---|---|
| | **Killed** | **Total** | **Score** | **Killed** | **Total** | **Score** | **Block Instr** |
| Board | 108 | 167 | 64% | 5 | 9 | 55% | 39% |
| Cell | 109 | 154 | 70% | 7 | 15 | 46% | 46% |
| Engine | 1 | 81 | 1% | 0 | 5 | 0% | 47% |
| Game | 31 | 147 | 21% | 0 | 9 | 0% | 49% |
| PlayerMove | 0 | 36 | 0% | 0 | 9 | 0% | 50% |

For those classes which had a unit test suite, the mutant killing rates are illustrated in table 2. The higher the percentage score, the better the hand-crafted unit tests were in killing mutants. This is presumed to correlate strongly with the ability of the same tests to detect similar subtle mistakes made by the programmer in the source code. The scores are fairly high for the `Board` and `Cell` classes (especially for method-level mutations), which is not surprising, since the *JUnit* tests were properly conceived as complete unit tests for these classes. However, the tests for `Game` and `Engine` did not provide anything like exhaustive coverage of all transitions between states, so their mutant killing scores are very low. When comparing these scores with the traditional block instruction coverage achieved by the same test sets, it is clear that the hand-crafted tests were still far from complete.

Mutation testing with *μJava* clearly highlighted where there was scope for improvement in the hand-crafted tests supplied with the *JPacman* software. However, the time and effort invested by the tester was exceedingly high, since all the code and all the tests had to be converted to Java 1.4. Having to maintain multiple versions of the same code purely for the sake of testing with *μJava* is not really feasible and means that using the current version of the tool is impractical for realistic, modern Java systems. A version of the tool, called *MuClipse*, has subsequently been developed as a plugin for the *Eclipse* IDE, but is still not yet able to process Java 1.5 and

1.6 language features [19]. So, an obvious future suggestion for *μJava* is that the tool be rapidly updated to deal with the latest version of the Java language.

## 4 Conclusions

Each of the tools *Randoop*, *JWalk* and *μJava* had its own strengths and weaknesses, offering support to testers in different areas. In our opinion, none of the tools were quite ready to take over completely from human-written test cases for the more sophisticated kinds of set-up; however, they were frequently better than humans in ensuring test completeness in areas where tests are tedious to write, or particular cases are hard to discover. Interestingly, none of the tools could be forced to make use of self-specification provided in the tested software through `invariant()` methods; and none of the tools could be expected to interact with Java's low-level `assert()` mechanism, since this is designed to terminate execution upon failure.

### 4.1 Test Coverage

*Randoop* was expected to cover the majority of the instructions in the tested code, simply by the expectation that random testing, in the limit, eventually covers every possibility. What was surprising here was that this did not seem to be the case when instruction coverage was measured using *Emma*. For most of the classes in the `jpacman.model` package, instruction coverage seemed to approach a limit of around 50% (fig. 3). This seems to indicate a limitation in random testing that could only be overcome by smarter choices of argument values to trigger particular cases.

The instruction coverage offered by *JWalk* could not be measured in the same way with *Emma*, since *JWalk* did not export a test suite that could be run with *Emma* interposed between that and the JVM. Instead, statement coverage can be estimated by considering the high-level state spaces of the CUTs. In other reported work, coverage of the state-space routinely reached 100% [20]. The weakness of *JWalk* is in its coverage of equivalence partitions. One would not expect all argument-triggered branches to be selected, where argument values are synthesized in monotonic sequence. However, because *JWalk* interleaves methods in all combinations, in the limit these appear in different orders where the supply of arguments occurs in the right order to trigger different branches, making coverage better than expected. The main limitation is where the initial set-up must be special, for which custom generators are required (and recommended by the documentation [17]).

By itself, *μJava* itself does not have an instruction coverage measure. However, it seeded faults that were detected between 0-70% of the time by the *JUnit* tests. This means that 30-100% of the mutants needed further tests to eliminate them, showing that *μJava* could provide a good indication of missing coverage. Looking at the hand-crafted *JUnit* tests, they routinely only covered 40-50% of the instructions of the CUTs in any case (see table 2), showing that the hand-crafted tests were still significantly incomplete, a useful warning to *JUnit* testers!

### 4.2 Testing Effort

*Randoop* was the easiest tool to use out-of-the-box, since it could be used to test all the classes in the package with little set-up. To get *Randoop* to measure sensible properties of the code required some effort to place `@Observer` annotations in the source. The `@Omit` tag was used less, and the `@ObjectInvariant` tag did not seem to work as expected. *Randoop* was the best tool for generating a regression test set from scratch, where no prior test-set existed. However, *Randoop* was not so useful for generating conformance tests, since while a mechanism was provided for identifying salient properties of the code, no clear mechanism existed for inducing which semantic properties of the application should hold. No documentation was provided to help the tester design and upload custom contract-checkers.

*JWalk* was also very easy to use out-of-the-box and was the only tool that did not require any special treatment of the source code. It worked directly with compiled *Java* class files; and seemed to work with different *Java* versions (up to 1.6). However, it tested each class one-by-one, which took longer than *Randoop*. On the other hand, *JWalk* could identify useful semantic properties of the tested classes without further assistance, and gave the tester the option to decide which of these were correct or incorrect observations. *JWalk* was definitely the most effective way to develop a conformance-testing oracle, far outperforming the time taken to develop hand-crafted *JUnit* tests, by at least fifty-fold according to [20]. Although constructing the oracle could become tedious for a large class, the amount of testing effort invested always paid off, because the tester knew what s/he was getting in terms of relative completeness of coverage for the testing effort invested.

Unfortunately, *μJava* required an unreasonable amount of testing effort, both in terms of the expected need to create and supply suitable tests by other means, but also by the unreasonable effort needed to translate the source code to the earlier version 1.4 of Java (pre-generics and annotations). Mutation testing really only makes sense in the context of an existing test suite that is being maintained, perhaps in response to changes to the tested source, to help identify gaps in test coverage. The more recent version *MuClipse* [19] can now accept tests in *JUnit* format.

Despite taking many, many times longer to develop, hand-crafted *JUnit* tests were sometimes the best way to execute particular properties of an application that required complicated set-up and tear-down [5] procedures. Manual testing was, for these cases, more straightforward than designing custom generators for *JWalk*.

### 4.3 Test Set Maintenance

*Randoop* was the only tool that exported tests in the *JUnit* format for external maintenance (*JWalk* did not export its tests; *μJava* expected pre-existing tests). These tests were classified into: *regression tests*, and *contract-violating* tests. The tests were not really designed to be maintained independently from the tool. Firstly, automatic variable naming (using up to 100 variables) meant that tests of any significant complexity were impossible for a human tester to comprehend. Secondly, the random nature of each test (a random set-up before either: a randomly observed property, or built-in contract-check) meant that it was impossible to tell which tests fell into the same equivalence classes, viz. which ones might be redundantly checking the same properties in the code. It might be more appropriate for a tester using *Randoop* to maintain sets of contract-checkers, and rely on the tool to regenerate tests on demand.

*JWalk* was theoretically biased towards exporting only the learned specification (the test oracle), on the grounds that the fault-detection ability of exported and independently-maintained tests degrades at a geometric (rather than linear) rate, as the production software evolves [8]. The test oracle is a good indication of what the tool has learned about the algebraic structure of the CUT. The tool always regenerates tests internally, according to filtered, breadth-first exploration, using the oracle to detect and prune out redundant tests in the same equivalence-class as an existing test. Each test is therefore guaranteed to test a unique property of the CUT. Furthermore, *JWalk* immediately adapts its internal test sets to changes made in the production code, making *JWalk* overall the best tool at maintaining its own test sets.

As stated above, *μJava* is designed to help maintain the quality of an existing test set and this is its proper role. It is very useful at hypothesizing faults that a test set cannot yet detect, and offers a measure (in terms of the mutant-killing index) of how good an individual test might be [19]. Apart from this, it is not clear whether this correlates strongly with other measures of non-redundancy in test sets derived from specifications (c.f. JWalk).

Finally, hand-crafted *JUnit* test are relatively easy to comprehend and so are easy to maintain from that standpoint. However, the *μJava* experiment showed how weak the effectiveness of these tests was, despite the fact that they had been carefully constructed [13] to follow sound principles of test design [14]. Elsewhere, it has been shown that human test maintainers both miss important properties to test, especially the negative properties that a system should not exhibit, and redundantly test the same properties repeatedly, in *JUnit* test suites [20].

## References

1. Pacheco, C., Lahiri, S. K., Ernst M. D. and Ball, T.: Randoop: feedback-directed random test generation. Proc. 29th. Int. Conf. Softw. Eng., Minneapolis, IEEE Computer Society (2007), 75-84
2. Simons, A. J. H.: JWalk: a tool for lazy systematic testing of Java classes by design introspection and user interaction. Autom. Softw. Eng., 14 (4), Springer USA (2007), 369-418
3. Ma, Y., Offutt, J. and Kwon, Y.: MuJava: an automated class mutation system. Softw. Test., Verif. and Reliab., 15 (2), John Wiley, London (2005), 97-133
4. Cornelissen, B., van Deursen, A., Moonen, L and Zaidman, A.: Visualising test suites to aid in software understanding. Proc. 11th Euro. Conf. Softw. Maint. and Re-eng., IEEE Computer Society (2007), 213-222
5. Beck, K.: The JUnit Pocket Guide, 1st edn., O'Reilly Media, Beijing (2004).
6. Pacheco, C., Lahiri, S. K. and Ball, T.: Finding errors in .NET with feeback-directed random testing. Proc. Int. Symp. Softw. Test. and Anal., Seattle, ACM SigSoft (2008), 87-96
7. Pacheco, C. and Ernst, M. D.: Randoop: feedback-directed random testing for Java. OOPSLA Companion, Montreal, ACM Sigplan (2007), 815-816

8. Simons, A. J. H.: A theory of regression testing for behaviourally compatible object types. Softw. Test., Verif. and Reliab., 16 (3), John Wiley, London (2006), 133-156
9. Holcombe, W. M. L.: Where do unit tests come from? Proc. 4th Int. Conf. on Extreme Progr. and Flexible Proc. in Softw. Eng., Genova, LNCS 2675, Springer (2003), 161-169
10. Simons, A. J. H. and Zhao, W.: Dynamic analysis of algebraic structure to optimise test generation and test case selection, Proc. 4th Test. Acad. and Indust. Conf. Prac. and Resch. Tech., Windsor, IEEE Computer Society (2009), 33-42
11. Umar, M.: An evaluation of mutation operators for equivalent mutants. MSc dissertation, Kings College, London (2006).
11. Offutt, J., Ma, Y. and Kwon, Y.: An experimental mutation system for Java. Proc. Workshop Empir. Resch. Softw. Test., Softw. Eng. Notes, 29 (5), ACM Sigsoft (2004), 1-4
12. Offutt, J., Ma, Y. and Kwon, Y.: The class-level mutants of muJava. Proc. Workshop Autom. Softw. Test., Shanghai (2006), 78-84
13. van Rompaey, B., and Demeyer, S.: Establishing traceability links between unit test cases and units under test. Proc. 13th Euro. Conf. Softw. Maint. and Re-Eng., March, IEEE Computer Society (2009), 209-218
14. Binder, R.: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley (2000).
15. Roubtsov, V.: Emma: a free Java code coverage tool. http://emma.sourceforge.net/, last accessed 29 April, (2010)
16. Pacheco, C.: Randoop 1.2, http://people.csail.mit.edu/cpacheco/randoop/, last accessed 29 April, (2010)
17. Simons, A. J. H.: JWalk software testing tool suite, http://www.dcs.shef.ac.uk/~ajhs/jwalk/, last accessed 29 April, (2010)
18. Offutt, J.: μJava home page, http://cs.gmu.edu/~offutt/mujava/, last accessed 29 April, (2010)
19. Smith, B. H. and Williams, L. An empirical evaluation of the MuJava mutation operators. Proc. 2nd Test. Acad. and Industr. Conf. Prac. and Resch. Tech., Windsor, IEEE Computer Society (2007), 193-202
20. Simons, A. J. H. and Thomson, C. D.: Benchmarking effectiveness for object-oriented unit testing. Proc. 1st Softw. Test. Benchmark Workshop, 1st Int. Conf. Softw. Test., Lillehammer, IEEE (2008)