IN THE PITS: A CRITICAL EXAMINATION OF PITEST

Mario Gonzalez Macedo

Ortask
mgm@ortask.com

ABSTRACT

This paper examines a tool called PITest that aims to do mutation analysis for the Java programming language. The examination is aimed at answering the questions: 1) "is it implemented with sufficient quality?" and 2) "does it truly perform mutation analysis?" As such, the paper focuses on several important metrics such as the overall mutation score (i.e. how well its tests handle regressions and how well it is generally tested) among others. It also highlights several aspects that are specific to mutation analyzers, such as mutation operators and whether PITest's set of mutation operators actually lead to an accurate indication of the quality of its targets. This paper illustrates that even tools that are intended to be objective instruments for code quality suffer the same as the average open source project in terms of implementation and design quality, thus highlighting the need for higher standards for such tools.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; D.2.8 [**Software Engineering**]: Metrics—*product metrics, process metrics*

General Terms

Algorithms, Experimentation, Management, Measurement, Reliability, Theory

Keywords

Unit testing, quality metrics, code coverage, empirical software engineering, software faults, mutation analysis

1. INTRODUCTION

Mutation analysis is a powerful technique that aims to "test the tests" for software projects. It is an approach whose goal is to provide an objective measure about the quality of tests for a given software system [25, 26, 28, 29, 36, 37, 38]. That is, its purpose is to help improve the quality of tests by injecting faults in their system under test (SUT) and then to measure the effectiveness of the tests, whereby higher quality tests detect a higher number of injected faults than their lower quality counterparts.

Mutation analysis is based on two main theses: 1) the competent programmer hypothesis, and 2) the coupling effect. The competent programmer hypothesis states that most software faults that are caused by experienced software developers are simple syntactic errors [49]. For instance, one fault of this kind belongs to the class of bugs called "off-by-one" which might be introduced when a developer types ">" instead of the intended relational operator ">=" (or vice-versa). The coupling effect states that the symptoms arising from simpler bugs become connected as different paths of the software system are traversed, thus creating emergent faults whose symptoms are functions of the simple bugs [48, 49].

PITest is an open source tool for the Java programming language that aims to perform mutation analysis[1]. It originally started as a parallel loading and test execution system for running JUnit tests, but gradually shifted its focus toward mutation testing [1]. According to its creator, PITest claims to differentiate itself from other such tools in four mayor classification areas [2]:

1. Generation of mutants: PITest creates mutations at the bytecode level using a third-party library called ASM.

2. Selection of tests to run: PITest uses coverage-based test selection.[2]

3. Insertion of mutants: "Mutants are held in memory and inserted into a JVM using the instrumentation api" [2].

4. Detection of mutants: "Test classes are split into individual test cases which are then run until one of them kills a mutant" [2].

The primary purpose of this survey is to critically assess the implementation of PITest in terms of quality and overall usefulness. Since PITest is a tool that is ultimately aimed at improving the quality of tests by generating a presumably accurate metric (i.e. the mutation score which represents the current regression-detection capability and overall fitness of tests) for Java projects, it is important that this tool also be implemented with high quality. To this end, there are three mayor objectives for this paper: 1) to examine PITest v1.0.0 in order to garner an accurate understanding of the quality of its implementation, especially with respect to how well PITest has been tested; 2) to provide a helpful frame of reference for people looking to utilize PITest as a potential mutation analysis tool for their Java software projects, so they can make informed decisions based on the results shown here; and 3) to provide recommendations for improving the PITest implementation based on the outcomes of the examination.

2. RESEARCH GOALS

This paper aims to answer the following questions about the quality of PITest in terms of its implementation, as well as general usage as a purported mutation analyzer.

The first three research questions are aimed at examining quality-specific details of PITest's implementation while the remaining research questions are aimed at answering the technical aspects of PITest as a tool that aims to perform mutation analysis in general.

[1] http://pitest.org/

[2] We will later discuss why this approach is harmful when the aim is to perform mutation analysis.

©2014 Mario Gonzalez Macedo. All rights are reserved

In other words, while the first three questions assess how robust and reliable users can expect PITest's implementation to be, the remaining queries are meant to asses PITest's accuracy and usefulness for doing actual mutation analysis in Java projects. As such, the latter research questions represent some of the typical concerns that software developers have when looking to fit a mutation analysis tool into their software development processes.

The research questions to answer are the following.

***RQ*1**: Do the tests for PITest provide sufficient coverage for its implementation?

***RQ*2**: Are the tests for PITest implemented with sufficient quality?

***RQ*3**: How well do PITest's tests handle changes to the code base that would introduce faults (i.e. regressions) and consequently lower PITest's overall quality?

***RQ*4**: Does the mutation model for PITest account for interaction of bugs as per the coupling effect?

***RQ*5**: Does the mutation model for PITest reflect actual faults that experienced programmers might introduce?

***RQ*6**: Does PITest provide an accurate model from which reliable mutation testing for Java can be performed?

3. METHODOLOGY

The set of test suites can be taken as the public indicator of quality (or lack thereof) for open source projects. As such, in order to answer the first three research questions posed above, several types of analyses were performed to the core implementation of PITest v1.0.0, one of which was mutation analysis. It has been shown that test suites achieving a high mutation score tend to be associated with high quality projects [25, 26, 29, 36, 37, 38].

For the remaining research questions which aim to determine the general adequacy of PITest as an alleged mutation analyzer for Java, both the source code and documentation were inspected. Findings from these were then compared with current empirical research regarding effectiveness of mutation operators.

The sections below describe in detail each aspect of the methodology utilized to carry out the research, from object selection to the tools utilized for analysis of results. For each aspect of the research, the guidelines set forth by [20] were followed to achieve statistically sound results.

3.1. Object Selection

The core implementation of PITest v1.0.0[3] has a total of 185 test suites[4] out of which 70 (37.84%) were selected for analysis.

3.1.1. Selection Criteria For Mutation Analysis

Test suites that executed with failures were not included in the mutation analysis. The reason for this is that no calculations have to be applied *a posteriori* in order to correct the mutation score with respect to failing tests. This translates into faster, more accurate mutation analysis since the mutation score assigned when the analysis is complete is the final, effective score.

[3]https://github.com/hcoles/pitest

[4]As of Tuesday, June 17, 2014

The initial selection of test suites contained 9 failing test suites, which reduced the sample for mutation testing to 61 test suites. Nevertheless, this sample represents 33% of the total set of test suites and provides an error margin of at most 0.077.

3.2. Tools Used

3.2.1. Mutation Analysis

Mutation analysis was performed with Ortask Mutator 1.7[5], a cross-language mutation analyzer based on a hybrid technique between the "do faster" and "do smarter" approaches developed in [52] and expanded in [28, 31]. The core of the technique is called *selective mutation* in the literature, as it only applies mutations that reflect actual faults introduced by experienced software developers in projects in the wild [27, 28, 29, 31, 52, 56]. Further, Mutator creates higher-order mutants which enables the coupling effect. Doing so allows more complex faults to emerge and consequently to more closely simulate real faults found in the wild as was empirically demonstrated in [36, 37, 38].

Mutator applies mutation on the lexical (i.e. file) level, rather than at the parse stream. Thus, to achieve uniform mutation for each SUT, Mutator iteratively and pseudo-randomly selects small chunks from the SUT's source code and globally applies a mutation within the chunk.

Moreover, to achieve a fine-level of granularity for the mutation order[6], each chunk is initially only 12 bytes long. If no mutations can be applied to the current chunk, then the chunk grows arithmetically by adding 2 bytes and the pseudo-random selection process takes place once more. This is done iteratively until a mutation can be applied. In effect, Mutator allows for low and higher order mutations to capture the wide spectrum of faults found in software [36, 37, 38].

In this way, 2000 low- and high-order mutants were created for each test suite's SUT to achieve a high level of statistical confidence with respect to the test suite's mutation score, as per the guidelines established in [20]. In other words, each test suite was executed 2000 times, each time against a mutated version of its associated SUT to achieve a very low margin of error.

3.2.2. Statistical Analysis

The statistical software package R[7] was used to analyze the results.

4. RESULTS

To answer ***RQ*1**, ***RQ*2** and ***RQ*3**, mutation analysis was utilized to: 1) assess the coverage of PITest's implementation with respect to the set of test suites; 2) to rate the quality of the tests for PITest proper; and 3) to measure their regression-detection capability.

Tables 1 and 2 show the test suites that were analyzed, along with their associated mutation scores. The tables also indicate which sampled test suites failed when executed.

Table 3 shows the statistical 95% confidence interval along with the associated summary statistics for the overall mutation score of the sampled test suites for PITest. Based on the data, the 95% confidence interval expects *most* – 95%, in fact – of the tests for the core implementation of PITest to achieve low mutation scores between 0.47 and 0.62. Regarding summary statistics,

[5]http://ortask.com/mutator/

[6]A *k-order* mutant means *k* mutations per mutant. So, for instance, a 3-order mutant is a mutant with 3 mutations.

[7]http://www.r-project.org/

©2014 Mario Gonzalez Macedo. All rights are reserved

the median mutation score is 0.55 and the average is 0.54 while the standard deviation is 0.36.

Table 3: Mutation score statistics for PITest test suites

$\bar{\mu}$	$\tilde{\mu}$	σ	error margin	low	high
0.54	0.55	0.36	0.077	0.47	0.62

Figure 1 provides a better sense for the distribution of the mutation scores via a histogram. It also shows where the bulk of the scores is located for the applicable test suites in the sample. It is clear from the figure that the quality of test suites for PITest varies dramatically, with many of them achieving unacceptably low mutation scores. Indeed, the probability density function in figure 2 shows that the *majority* of the test suites for PITest's implementation achieve mutation scores below 0.63.

In terms of failing tests, there were 9 such suites within the sample of 70, or 12.9% of the sampled population, yielding 0.062 error margin. Statistically, this means that between 6.6% and 19.1% of the complete set of test suites for PITtest can be expected to fail when executed (i.e. up to a fifth of the tests for PITtest do not pass).

Another observation in terms of the degree to which PITest has been tested is the test-to-source ratio. Version 1.0.0 of PITest contains 361 source files, out of which 72 are interfaces that do not provide any functionality other than specifying a contract. Subtracting these interface files results in a total of 289 effective source files. Nevertheless, there are only 185 test suites in the core implementation of PITest, yielding a test-to-source ratio of 0.64. This means that the rest of the source files (36%) are either 1) not tested; or 2) tested simultaneously with other target sources. However, it is clear from figure 1, figure 2, and tables 1, 2 and 3 that the source files that do not have corresponding test suites are not tested at all or, if they are, they are poorly tested.

Given these data, we can answer ***RQ*1** and ***RQ*2** definitively.

> ***RQ*1**: The tests for PITest do not provide sufficient coverage for its implementation.

> ***RQ*2**: The tests for PITest are not implemented with sufficient quality.

In terms of regression detection, it was observed by running Mutator with the option "-a" (so that Mutator stops automatically once a live mutant is found), that PITest's tests cannot detect common regressions aligned with the competent programmer hypothesis – for which itself performs mutations on other Java projects. In fact, 75.8% of the sampled tests do not detect common regressions introduced by experienced software developers. For example, test suites *mutationtest/engine/LocationTest.java* and *mutationtest/engine/MutationIdentifierTest.java* cannot detect simple arithmetic changes in their SUT from "*" to "/", or from "+" to "-", respectively. Moreover, as far as concurrency is concerned, there are some test suites that do not detect concurrency issues. For instance, test suite *util/SocketFinderTest.java* does not detect when monitors become acquired by objects instead of the entire class, thus introducing race conditions and potential deadlock.

We can consequently answer ***RQ*3** as follows.

> ***RQ*3**: 75.8% of the tests for PITest do not detect common regressions introduced by experienced software developers that would lower PITest's overall quality.

To answer ***RQ*4**, we first note that low and higher order mutations capture the wide spectrum of faults found in software [36, 37, 38, 51]. Therefore, a good model for mutation testing must include both low and high order mutants. Inspecting PITest reveals support only for low order mutants but no support for higher order mutants. Further, there appears to be no plans to provide higher-order mutants in the future [3]. We can therefore answer ***RQ*4**.

> ***RQ*4**: PITest does not support the coupling effect, and consequently cannot help improve tests that do not detect cascading or coupled faults.

We answer ***RQ*5** by examining the set of mutation operators that PITest applies to a given SUT. In particular, [27, 28, 29, 56] empirically found that only certain mutation operators yield meaningful results for non-concurrent Java projects in order to achieve a reliable assessment of test quality and consequently to support the competent programmer hypothesis. Therefore, the nomenclature that was developed in [27, 28, 29, 56] is utilized in this paper to categorize PITest's non-concurrent mutation operators. Table 4 shows the meaningful mutation operators supported by PITest v1.0.0 for non-concurrent Java projects. The mutation operators used by Mutator are also shown for completeness.

For concurrent Java projects, [50, 51] developed and then empirically determined the necessary mutation categories that support the competent programmer hypothesis. Table 5 shows the mutation operators that are supported for concurrency by PITest as well as those supported by Mutator.

As table 4 shows, PITest supports a wide range of meaningful mutation classes except for two: *COR* and *ASR*. For *COR*, there is no support whatsoever, and the reason for this is not clear. With respect to *ASR*, we note that PITest only supports "assignment increments and decrements of non-local variables" [4], but there is no support for other assignment operators such as *= or /=. From this, we can conclude that even though PITest does not provide a complete set of meaningful mutation operators for non-concurrent Java projects, it does provide a partial set.

Nevertheless, it is clear by table 5 that PITest does not provide any meaningful mutation operators for concurrency. As such, PITest is not useful for mutation analysis of tests with respect to concurrency faults in their SUTs. Since such operators are of critical importance in enterprise projects, PITest is therefore not adequate for such projects.

Given this analysis, we can answer ***RQ*5** in two parts, as ***RQ*5a** and ***RQ*5b** as follows.

> ***RQ*5a**: PITest only provides a partial set of mutations for non-concurrent Java applications.

However, the answer for ***RQ*5b** is not as positive.

> ***RQ*5b**: PITest does not support analysis of concurrent Java applications.

We have all the ingredients necessary to answer ***RQ*6**. However, before we do so, it is important to consider how PITest selects tests to execute from the target project once it has created a mutant. Recall that PITest uses coverage-based test selection, which means that "[o]nly those tests that exercise the line, block or instruction that contains the mutant will be run" [2]. However, this approach should not be used for two important reasons. First, selecting

©2014 Mario Gonzalez Macedo. All rights are reserved

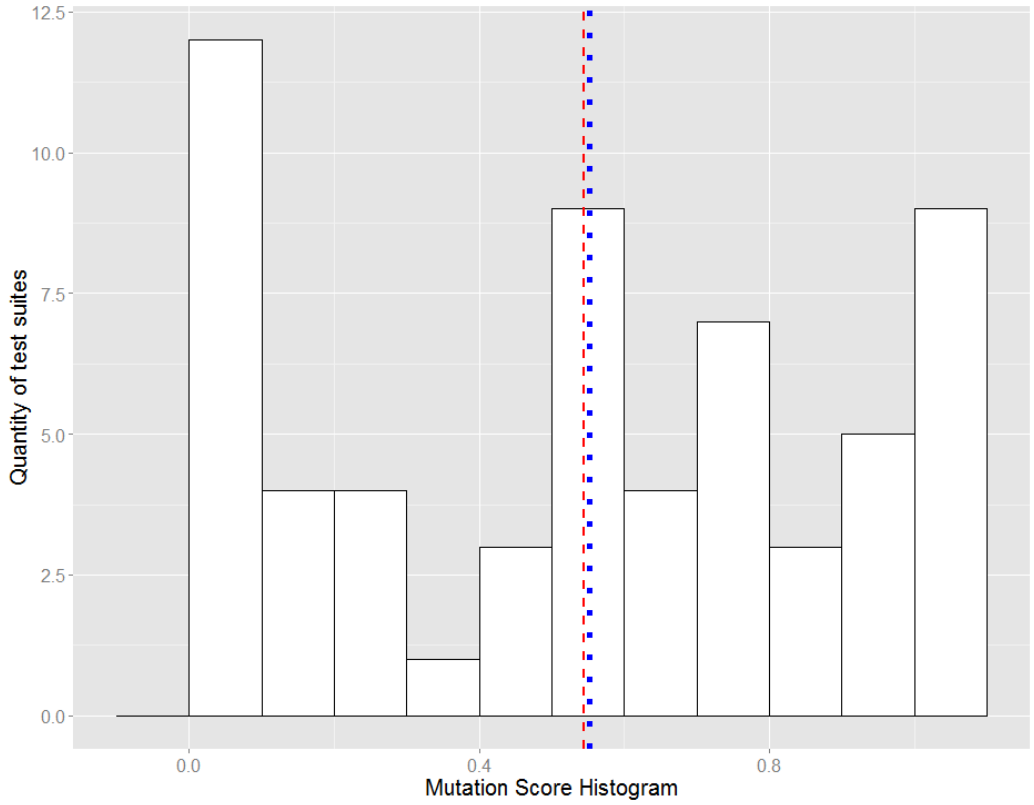

Figure 1: *Distribution of mutation scores for applicable test suites in the sample. The blue squared-dotted line represents the median mutation score, while the red dashed line represents the average mutation score.*

©2014 Mario Gonzalez Macedo. All rights are reserved

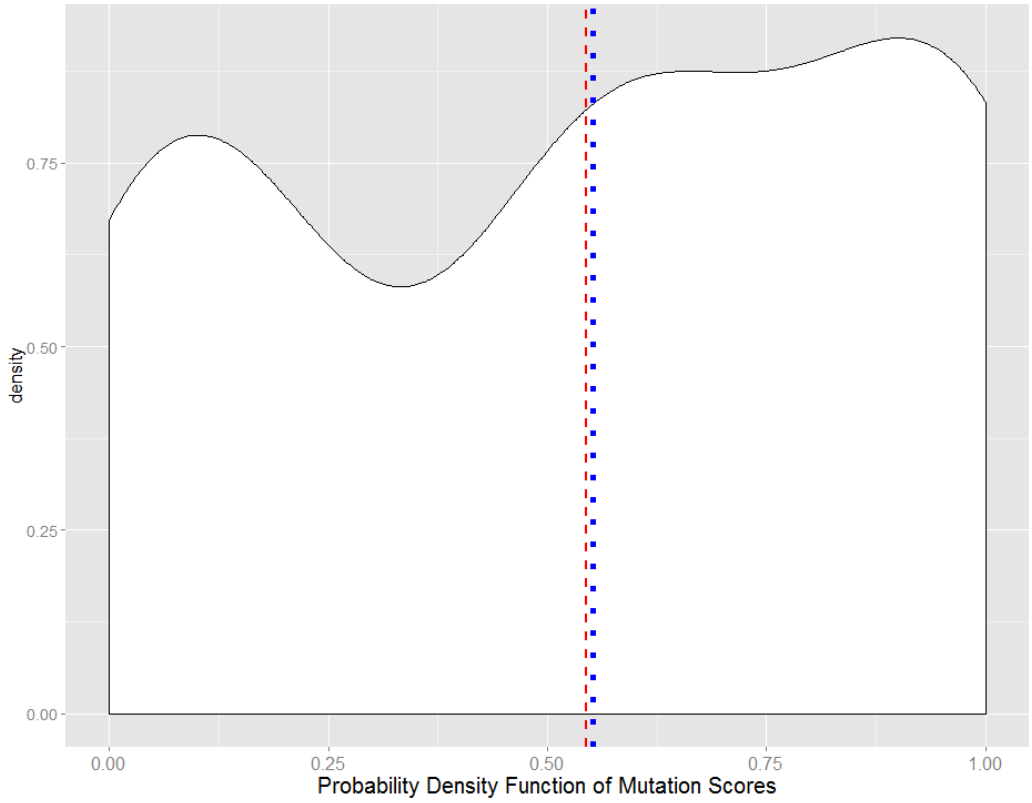

Figure 2: *Probability density function of mutation scores for applicable test suites in the sample. The blue squared-dotted line represents the median mutation score, while the red dashed line represents the average mutation score.*

©2014 Mario Gonzalez Macedo. All rights are reserved

tests based solely on whether they provide coverage for a mutation skews the mutation score because "for mutants that are not exercised by tests - none will be run" [2]. Doing so precludes executing other tests which might have detected the mutation, consequently yielding inaccurate results and poor insights into the quality of tests. Second, it is well known that mutation analysis subsumes code coverage as a criterion for measuring test quality and is consequently a much stronger metric [18, 54, 55]. Yet, PITest's coverage-based criterion for test selection is ultimately equivalent to traditional code coverage, which clearly does not provide any of the benefits of mutation testing.

Given the above analysis together with the previous findings, we answer ***RQ6*** as follows.

> ***RQ6***: PITest does not provide a complete model from which reliable mutation testing for Java can be performed.

5. RECOMMENDATIONS

Based on the analysis and findings above, there are several recommendations that can improve the implementation quality of PITest. First, the tests for the implementation of PITest should be improved so that they achieve a much higher mutation score. Currently, most of its test suites are not adequate to detect regressions introduced during the normal evolution of the project. Consequently, every change risks further decreasing its quality. There should also be one test suite per class file so that the behavior of each component is properly tested and documented. Even though this might be hard to attain, it is a much better practice than testing orthogonal components within a single test suite or, worse, leaving components untested as was discovered for several source files. Testing simultaneous components in a single test suite violates well-known good software development practices, such as SRP[8].

There are also several recommendations to improve PITest in terms of utility. First, it should provide the ability to create higher-order mutants. Supporting only low-order mutants merely provides half the mutation model, which prevents users from getting the benefits of mutation testing. Further, PITest should include more realistic mutation operators so that it reflects actual faults introduced by software developers. μJava is an excellent example of a mutation analyzer that overcomes this limitation by supporting a wider range of mutation operators than PITest [30, 53].

Finally, test selection should not be based on code coverage, as this completely eliminates the benefits of mutation testing. Rather, selection of tests should be based on a more holistic strategy so that: 1) test suites are measured correctly and accurately, and 2) the coupling effect is enabled.

6. CONCLUSION

This paper has examined PITest – a tool that aims to be a mutation analyzer for projects written in the Java programming language – with respect to implementation quality and overall utility. The findings indicate that PITest does not actually perform mutation analysis due to the presence of serious drawbacks in its design as well as in its implementation. For instance, the paper has shown that PITest's set of mutation operators leads to a partial and skewed indication of the quality of its targets. Further, PITest does not support or enable the coupling effect, which is one of the two foundations for mutation testing.

[8]Single Responsibility Principle

Finally, the implementation for PITest requires work to stabilize and improve the mutation scores of its tests. For instance, the tests for the implementation of the PITest project do not detect serious regressions (either concurrent or non-concurrent) that would threaten its reliability as they are introduced as part of normal evolution of the system.

Along with the analysis of its implementation and utility, this paper has also provided a set of recommendations to improve PITest's overall quality and utility.

7. REFERENCES

[1] http://pitest.org/faq/

[2] http://pitest.org/java_mutation_testing_systems/

[3] https://github.com/hcoles/pitest/issues

[4] http://pitest.org/quickstart/mutators/

[5] V. Vangala, J. Czerwonka, and P. Talluri, "Test Case Comparison and Clustering using Program Profiles and Static Execution," Microsoft, 2009

[6] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring Test Case Similarity to Support Test Suite Understanding," 2012

[7] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," 2004

[8] N. Koochakzadeh, V. Garousi, and F. Maurer, "A Tester-Assisted Methodology for Test Redundancy Detection," 2009

[9] G. Fraser and F. Wotawa, "Redundancy Based Test-Suite Reduction," 2007

[10] N. P. Singh, R. Mishra, R. R. Yadav, "Analytical Review of Test Redundancy Detection Techniques," 2011

[11] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," 1993

[12] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," 2005

[13] H. Zhong, L. Zhang, and H. Mei, "An Experimental Comparison of Four Test Suite Reduction Techniques," 2006

[14] J. von Ronne, "Test Suite Minimization, An Empirical Investigation," 1999

[15] D. Li, C. Sahin, J. Clause, and W. G. J. Halfond, "Energy-Directed Test Suite Optimization," 2013

[16] T. Gergely, Á. Beszédes, T. Gyimóthy, and M. I. Gyalai, "Effect of Test Completeness and Redundancy Measurement on Post Release Failures âĂŞ an Industrial Experience Report," 2010

[17] M. G. Macedo, "Unit-Test Source Code is a Good Approximation For Abstract States," Ortask, 2013

[18] M. G. Macedo, "The Effects of Test Redundancy On Software," Ortask, 2013

[19] R project, http://www.r-project.org/

©2014 Mario Gonzalez Macedo. All rights are reserved

[20] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," 2010

[21] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," Journal of Educational and Behavioral Statistics, 2000

[22] http://rubyonrails.org/

[23] http://ai4r.org/

[24] http://jruby.org/

[25] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?," 2005

[26] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the âĂIJSmall Scope HypothesisâĂİ," 2003

[27] A. J. Offutt, and J. Pan, "Procedures for Reducing the Size of Coverage-based Test Sets," 1995

[28] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," 1996

[29] A. J. Offutt, and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," 2001

[30] A. J. Offutt, and Y. Ma, "Description of Method-level Mutation Operators for Java," 2005.

[31] A. J. Offut, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," 1993.

[32] P. Cousot, and R. Cousot, "Basic Concepts of Abstract Interpretation," 2005

[33] P. Cousot, and R. Cousot, "Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Constructions or Approximation of Fixpoints," 1977

[34] http://rubygems.org/gems/heckle

[35] http://www.ruby-lang.org/en/news/2013/06/30/we-retire-1-8-7/

[36] M. Harman, Y. Jia, and W. B. Langdon, "Strong Higher Order Mutation-Based Test Data Generation," 2011

[37] M. Harman, and Y. Jia, "Constructing Subtle Faults Using Higher Order Mutation Testing," 2008

[38] M. Harman, and Y. Jia, "Higher Order Mutation Testing," 2009

[39] M. J. Knol, W. R. Pestman, and D. E. Grobbe, "The (mis)use of overlap of confidence intervals to assess effect modification," 2010

[40] R. Hubbard, and R. M. Lindsay, "Why P Values Are Not a Useful Measure of Evidence in Statistical Significance Testing," 2008

[41] A. Knezevic, "Overlapping Confidence Intervals and Statistical Significance," 2008

[42] J. Shtaynberger, and H. Bar, "Equivalence Testing," 2013

[43] M. Harder, B. Morse, and M. D. Ernst, "Specification Coverage as a Measure of Test Suite Quality," 2001

[44] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test Coverage and Post-Verification Defects: A Multiple Case Study," 2009

[45] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," 1997

[46] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The Relationship Between Test Coverage and Reliability," 1994

[47] S. Berner, R. Weber, and R. K. Keller, "Enhancing Software Testing by Judicious Use of Code Coverage Information," 2007

[48] A. J. Offutt, "Investigations of the Software Testing Coupling Effect," 1980

[49] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," 1978

[50] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," 2006.

[51] L. Wu and G. Kaiser, "Empirical Study of Concurrency Mutation Operators for Java," 2010.

[52] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur, "Constrained Mutation in C programs," 1994.

[53] http://cs.gmu.edu/ offutt/mujava/

[54] K. Pan, X. Wu, and T. Xie, "Automatic Test Generation for Mutation Testing on Database Applications," 2013.

[55] R. Gopinath, C. Jensen, and A. Groce, "Code Coverage for Suite Evaluation by Developers," 2014.

[56] M. Umar, "An Evaluation of Mutation Operators for Equivalent Mutants," 2006.

©2014 Mario Gonzalez Macedo. All rights are reserved

Table 1: Mutation scores for sampled test suites

Test Suite *	Mutation Score	Fails?
mutationtest/engine/LocationTest.java	0.07	–
mutationtest/engine/MutationIdentifierTest.java	0.36	–
mutationtest/report/xml/XMLReportListenerTest.java	0.66	–
mutationtest/statistics/MutationStatisticsTest.java	0.74	–
mutationtest/statistics/ScoreTest.java	0.87	–
functional/OptionTest.java	0.48	–
classinfo/ClassNameTest.java	0.54	–
classpath/ClassPathTest.java	0.17	–
execute/containers/ExtendedTestResultTest.java	0.0	–
functional/prelude/PreludeTest.java	0.54	–
mutationtest/MutationMetaDataTest.java	0.12	–
mutationtest/DetectionStatusTest.java	0.996	–
mutationtest/filter/LimitNumberOfMutationPerClassFilterTest.java	1.0	–
mutationtest/mocksupport/JavassistInterceptorTest.java	1.0	–
mutationtest/report/csv/CSVReportListenerTest.java	0.66	–
process/JavaProcessTest.java	0.23	–
execute/ExitingResultCollectorTest.java	0.66	–
functional/FArrayTest.java	0.997	–
junit/JUnit4SuiteFinderTest.java	0.09	–
mutationtest/engine/gregor/AvoidAssertsMethodAdapterTest.java	0.79	–
mutationtest/engine/gregor/blocks/BlockTrackingMethodDecoratorTest.java	0.99	–
mutationtest/execute/MutationTimeoutDecoratorTest.java	0.25	–
mutationtest/execute/TimeOutDecoratedTestSourceTest.java	1.0	–
mutationtest/execute/CheckTestHasFailedResultListenerTest.java	1.0	–
mutationtest/MutationResultTest.java	0.0	–
bytecode/FrameOptionsTest.java	0.98	–
TestResultTest.java	0.0	–
mutationtest/config/ReportOptionsTest.java	0.08	–
classinfo/TestToClassMapperTest.java	0.794	–
coverage/CoverageDataTest.java	0.77	–
execute/TestPitest.java	0.51	–
execute/MultipleTestGroupTest.java	0.0	–
functional/FCollectionTest.java	1.0	–
junit/adapter/AdaptedJUnitTestUnitTest.java	0.18	–
junit/adapter/AdaptingRunListenerTest.java	1.0	–
mutationtest/MutationStatusMapTest.java	0.48	–
mutationtest/build/MutationTestUnitTest.java	0.22	–
mutationtest/build/DefaultTestPrioritiserTest.java	0.49	–
mutationtest/build/PercentAndConstantTimeoutStrategyTest.java	0.22	–
mutationtest/build/DefaultGrouperTest.java	1.0	–
mutationtest/tooling/JarCreatingJarFinderTest.java	0.88	–
mutationtest/tooling/SpinnerListenerTest.java	0.77	–
mutationtest/execute/MutationTestWorkerTest.java	0.07	–
mutationtest/incremental/IncrementalAnalyserTest.java	0.53	–
mutationtest/incremental/DefaultCodeHistoryTest.java	0.5	–
util/StreamUtilTest.java	0.07	–
util/StringUtilTest.java	0.0	–
util/TimeSpanTest.java	0.85	–

* *pitest/src/test/java/org/pitest/*

©2014 Mario Gonzalez Macedo. All rights are reserved

Table 2: Mutation scores for sampled test suites (continued)

Test Suite *	Mutation Score	Fails?
util/InputStreamLineIterableTest.java	0.55	–
util/FunctionsTest.java	0.0	–
util/IsolationUtilsTest.java	0.62	–
util/GlobTest.java	1.0	–
util/SafeDataInputStreamTest.java	0.14	–
TestGroupingStrategies.java	0.59	–
TestJUnitConfiguration.java	0.59	–
junit/adapter/ForeignClassLoaderAdaptingRunListenerTest.java	0.96	–
util/SocketFinderTest.java	0.024	–
util/ComputeClassWriterTest.java	0.71	–
coverage/execute/ReceiveTest.java	0.795	–
mutationtest/engine/gregor/MethodInfoTest.java	0.52	–
junit/JUnitTestClassIdentifierTest.java	1.0	–
classinfo/ClassInfoTest.java	–	yes
classpath/ArchiveClassPathRootTest.java	–	yes
functional/MutableListTest.java	–	yes
classinfo/ClassInfoVisitorTest.java	–	yes
classpath/DirectoryClassPathRootTest.java	–	yes
coverage/codeassist/CoverageMethodVisitorTest.java	–	yes
dependency/DependencyExtractorTest.java	–	yes
mutationtest/verify/DefaultBuildVerifierTest.java	–	yes
util/ServiceLoaderTest.java	–	yes

** pitest/src/test/java/org/pitest/*

Table 4: Meaningful non-concurrency mutation operators for Java

Operator	PITest	Mutator
AOR_B: Binary Arithmetic Operator Replacement	Yes	Yes
AOR_U: Unary Arithmetic Operator Replacement	Yes	Yes
AOR_S: Short-cut Arithmetic Operator Replacement	Yes	Yes
ROR: Relational Operator Replacement	Yes	Yes
COR: Conditional Operator Replacement	–	Yes
SOR: Shift Operator Replacement	Yes	Yes
LOR: Logical Operator Replacement	Yes	Yes
ASR: Assignment Operator Replacement	Partial	Yes

Table 5: Meaningful concurrency mutation operators for Java

Operator	PITest	Mutator
$RKSN$: Remove synchronized keyword	–	Yes
$RMWN_1$: Remove call to wait	–	Yes
$RMWN_2$: Remove call to notify	–	Yes
$RMWN_3$: Remove call to notifyAll	–	Yes
RMV: Remove volatile keyword	–	Yes
RMS: Remove static keyword	–	Yes

©2014 Mario Gonzalez Macedo. All rights are reserved