# Is Mutation Testing Scalable for Real-World Software Projects?

Simona Nica, Franz Wotawa
*Institute for Software Technology*
*Graz University of Technology*
*Graz, Austria*
*snica,wotawa@ist.tugraz.at*

Rudolf Ramler
*Software Competence Center Hagenberg GmbH*
*Hagenberg, Austria*
*rudolf.ramler@scch.at*

***Abstract*—A significant amount of research has been conducted in the area of mutation testing. It is a fault based technique that has been intensively used, over the last decades, as an efficient method to assess the quality of a given test suite. In the literature different mutation tools are available, corresponding to different programming languages or different types of applications. Although mutation testing is a powerful technique, limitations do exist. The most common problems are represented by the increased computation time, necessary to derive the entire mutation testing process, and the equivalent mutants problem. Therefore a natural question arises: is mutation testing really suitable in real-world environments? Through the research we start here, we aim to come with an accurate answer to this question.**

***Keywords*-mutation testing; mutation tools; coverage tools; eclipse project;**

## I. Introduction

Mutation testing is a test technique that has been used to evaluate the test suite of an application, but also for the test case generation process. It is a fault based technique that makes use of a well determined set of faults for measuring the efficiency of the test suites. The mutation process involves the following steps:

1) Faults are introduced into a program resulting in different faulty versions (mutants) of this program.
2) Each mutant is run against the provided set of test suites. When a mutant fails to pass a test case, it is said that the mutant is killed. Otherwise it is still alive or it could not be detected - e.g., due to dead code or because it is an equivalent mutant.
3) The mutation score (the ratio between the number of killed mutants and the number of all mutants) is computed. The mutation score is an indicator used to evaluate the effectiveness of a test suite, i.e., its capability to detect the faults introduced through mutations, and thus describes the test suite adequacy.

A mutant is said to be equivalent with the original program when there is no way that a test case can detect the modification - since the output will always be the same with the output of the original program. Figure 1 presents an example, the arithmetic operator replacement (AOR) mutation. It is important to detect and avoid equivalent mutants because they cause an artificially low mutation score, as they cannot be killed.

Mutation testing is seen as a good metric for measuring the coverage levels achieved through different test coverage techniques. The authors in [1] prove that in some situations coverage measure techniques do not represent the most adequate measure in discovering all the faults an application is prone to. For example, in the case of test driven development, one first writes the tests and then starts writing the source code. In most of these situations the programmer obtains a good coverage of the code, but only those specific faults may be detected, the ones the programmer had thought of during the development of the tests. In contrast, mutation testing can be taken as a good indicator for measuring the coverage levels achieved through different test coverage techniques.

In 1971, Richard Lipton introduced the concept of mutation testing. The technique was further developed by DeMillo, Lipton and Sayward [2]. The technique can be applied at unit testing level [3], [4], [5], integration testing level[6], [7] or it can be used to validate the specifications [8], [9], [10], [11]. Several mutation testing tools were developed, for different existing programming environments: Fortran [12], [13], Java [14], [15], [16], [17], [18], C# [19], [20], C [21] and SQL [22]

Although the mutation testing technique can be computationally very expensive and also time consuming, it has been shown that mutation testing is stronger than coverage based metrics [4]. Therefore a natural question arises: is mutation testing worth the effort in a real life software project?

This paper is structured as follows. In Section 2 we give a brief description of the working environment and the tools used in our research. In Section 3, we present and discuss the results. In Section 4 we discuss the related research. Finally, in Section 5 we conclude the paper.

## II. Environment Setting

In this paper, we aim to assess the costs of applying mutation testing on a real-life software system. Following aspects have been investigated to answer the questions whether mutation testing worth the effort in a real life software project:

```
if( a == 2 && b == 2)
  c = a * b;
```

$$\xrightarrow{\text{Applying AOR operator}}$$

```
if( a == 2 && b == 2)
  c = a + b;
```

Figure 1. Equivalent Mutant

- The time required for mutation testing,
- The results of mutation testing compared to coverage analysis,
- The issues encountered in setting up and running selected mutation testing tools.

In what follows, we briefly present the working environment configuration.

## A. Environment Configuration

We have chosen to use mutation testing on Eclipse [23], a widely known and large open source project that shows many parallels to commercial and industrial software projects, especially those developed on the basis of the Eclipse application framework. We retrieved the source code Eclipse Release Build 3.0, from the Eclipse repository [23].

In Table I, the versions and configuration parameters of the tools and test objects used throughout our research are described. All of the presented work was conducted using the virtual environment Oracle VM Virtual Box. The virtual machine is configured to run on Windows XP SP2 operating system, on an Intel Core 1.73 GHz with 2 GB of RAM. For the Java Virtual Machine, we compile and run all the files involved in the research with version 1.6, update 24. We have chosen to work within a virtual environment, in order to offer a fast portability and also an easy management for our research. We aim at a fully automatized process, for all the Eclipse plug-ins, which will run over a predefined period of time, on different architectures.

We apply three of the most widely used mutation tools: MuJava [3], Jumble [16] and Javalanche [15]. We run the mutation testing technique and then compare the results with the code coverage information provided by Clover [27] and EclEmma [28].

## B. Applied Mutation Tools

For computing the mutation score metric, we take into account, throughout the research, the following mutation tools:

| Tool / System | Version | Location/Comment |
|---|---|---|
| Eclipse | 3.0 | [23] |
| MuJava | 3 | [24] |
| Jumble | 1.1.0 | [25] |
| Javalanche | 0.3.6 | [26] |
| Clover | 3.0.2 | [27] |
| EclEmma | 1.5.1 | [28] |

Table I
OVERVIEW ENVIRONMENT CONFIGURATION

1) **MuJava**: MuJava is a Java based mutation tool, which was originally developed by Offut, Ma, and Kwon [14]. Its main three characteristics are:
   - Generation of mutants for a given program.
   - Analysis of the generated mutants.
   - Running of provided test cases.

   Due to the newly implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration into the testing or debugging process. Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. It works with both types of mutation operators:
   - Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
   - Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

   MuJava was not designed to work with JUnit test cases, nor to compile with Java versions greater than 1.4; i.e., Java development kit 1.5 or 1.6. Due to the fact that for most of the applications we use throughout the research, we work with MuJava as the mutation testing tool, we have implemented different add-ons to support JUnit test cases and partial mutation of Java source files compiled with JDK 1.5 or greater. We take into account both the traditional mutation operators, i.e., the method level, and the class level ones. MuJava comes with a graphical user interface.
2) **Jumble** It is a class level mutation tool. Moreover, this tool supports JUnit 3 and, recently, it was updated to work with JUnit 4. Similar to MuJava, just one mutation is possible at a time, over the source code under test. First, the tool runs all the tests on the original, unmodified, source file and checks whether they pass or not, recording the time necessary for each test. Then, it mutates the file according to different mutations operators and runs the tests again. The process is done when all the mutations have been tested. Unlike MuJava, Jumble is able to mutate constants.
3) **Javalanche** This mutation testing tool should resolve two major problems in mutation testing: efficiency and equivalent mutants problem. Javalanche works on byte code and can mutate very large programs. The

authors resolve the problem of equivalent mutants by assessing the impact of mutations over the dynamic invariants [29]. According to the authors of the tool, Javalanche has an unique feature. The tool is able to rate the mutations in accordance with their impact on the behavior of program functions, i.e., the greater the impact of an undetected mutation is, the lower the possibility of an equivalent mutant.

We have chosen to conduct the research using the above described tools, taken into account their usage inside the experiments conducted in the mutation testing area.

## III. Results

In this section, we present the first results of our research, by taking into consideration the three aspects that we follow in our research work: time, mutation testing results, using the JUnit tests provided on the Eclipse repository, and finally we describe the issues encountered in setting up and running the different mutation testing tools.

### A. Research Procedure

In our research we follow the next steps:

1) Check-out the project from the Eclipse repository;
2) Run the plug-in test cases associated to the checked out project;
3) Download and install the coverage and mutation tools;
4) Set all the necessary class paths for each tool;
5) Run the tools over the original project and record the results. This step is the one that consumes most of the time, i.e., approximately 1 month and a half in case of our chosen plug-in project. This is mainly due to the different compilation exceptions encountered; for the compilation and tools running tasks one human resource was allocated.

As the research procedure is the same for each of the Eclipse plug-ins, we conduct the first research steps with the Eclipse Java development tools Core project. The JDT [30] provides the tool plug-ins that implement the Java IDE, which supports the development of any Java application, including Eclipse plug-ins.

The JDT Core project, *org.eclipse.jdt.core*, has associated three test projects:

1) org.eclipse.jdt.core.tests.builder
2) org.eclipse.jdt.core.tests.compiler
3) org.eclipse.jdt.core.tests.model

### B. Time

Concerning the time necessary to derive this research, we have to take into account:

- The time necessary to configure the tools; the effort estimated was of approximately one week;
- The mutants generation time; for the selected plug-in, it took us between 6 to 8 hours, i.e., a full working day;
- The time needed to run the test cases against the set of mutants. This is the most significant one, as we have a huge number of mutants.

### C. Mutation Results

For each test project from Table II, we computed the total number of initial test cases $\mathbf{No_{TC}}$, the initial time $\mathbf{T_{orig}}$, in minutes, needed to run the tests, and the success rate $\mathbf{S_{rate}}$ which tells us the percentage of tests that initially passed.

In Table III, we show the detailed mutation testing information for one of the three test projects, *org.eclipse.jdt.core.tests.compiler.regression*. We record the number of generated mutations $\mathbf{No_{Mut}}$, the necessary time for generating all the mutations, $\mathbf{T_{Mut}}$, the mutation score **MS** and the total time for running the tests over the mutants, i.e., $\mathbf{T_{TC_{Mut}}}$. MuJava generated 123 class mutants and almost 31 000 method mutants, in approximately 360 minute, i.e., 6 hours. We estimated the total time for running all the generated mutants; we did not run all the method level mutants, due to the increased time complexity. The average mutation score recorded was around 65%. The computed mutation score, for MuJava, is the average of all the mutation scores computed for each run of the plug-in, in accordance with the selected mutants.

As it can be observed from Table III, we were not able to obtain any mutation points for Jumble and Javalanche. By $\mathbf{T_{No_{TC}}}$ we denote the total number of test cases from a specific test project. In Table IV, we record, the success rate for the three plug-in projects, after running all the test cases from each project, using the coverage tools. Concerning the types of code coverage recorded by the tools we have selected for our research, we know that:

- *Clover* measures statement, branch and method coverage;
- *EclEMMA* computes class, method, statement and basic block code coverage.

In the research conducted so far, we have reported the mutation score to the statement coverage level. Further code coverage measures will be taken into account for the mature stages of our research.

### D. Encountered Issues

Up to now we were not able to generate mutants, for the JTD Core project, with Jumble or Javalanche; this part of

| Test Project | $\mathbf{No_{TC}}$ | $\mathbf{T_{orig}}$ | $\mathbf{S_{rate}}$ |
|---|---|---|---|
| org.eclipse.jdt.core.tests.builder | 79 | 161.312 | 100.00 % |
| org.eclipse.jdt.core.tests.compiler | 2542 | 16.203 | 100.00% |
| org.eclipse.jdt.core.tests.regression | 2622 | 387.735 | 100.00% |
| org.eclipse.jdt.core.tests.eval | 350 | 65.562 | 100.00% |
| org.eclipse.jdt.core.tests.dom | 1584 | 136.437 | 100.00% |
| org.eclipse.jdt.core.tests.formatter | 486 | 21.109 | 100.00% |
| org.eclipse.jdt.core.tests.model | 2084 | 293.782 | 100.00% |

Table II
Eclipse JUnit Test Results

| Tool | $\mathbf{No_{Mut}}$ | $\mathbf{T_{Mut}}$ | MS | $\mathbf{T_{TC_{Mut}}}$ |
|---|---|---|---|---|
| MuJava | 123/30947 | 174.69 min/185.7min | app.65% | est. 2 months |
| Jumble | - | - | - | - |
| Javalanche | - | - | - | - |

Table III
MUTATION TESTING INFORMATION PER MUTATION TESTING TOOL

| Project | $\mathbf{T_{No_{TC}}}$ | Clover | EclEMMA |
|---|---|---|---|
| org.eclipse.jdt.core.tests.builder | 79 | 100.00% | 100.00% |
| org.eclipse.jdt.core.tests.compiler | 16287 | 100.00% | 100.00% |
| org.eclipse.jdt.core.tests.model | 8639 | 99.97% | 100.00% |

Table IV
SUCCES RATE

our work is still in progress. The main problem we have encountered was to run the test cases as plug-ins test, using the different mutation tools. Besides time consuming, the generation of mutants proved to be also very complex.

Concerning the first mutation tool, MuJava, there are some limitations we have to take into consideration:

- MuJava is not able to generate any mutants in case of constants (it does not mutate constant values);
- Also, missing statements are another limitation of the tool. We are not able to generate mutants, by statement deletion nor insertion;
- In case of multiple bugs in one statement, the MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program (this limitation is however easy to overcome);
- In order to support execution of JUnit tests, the nullary constructor has to be added to each test class file. Also, the private methods *setUp()* and *tearDown()* must have public access;
- The last problem regarding mutation is that sometimes equivalent mutants are generated.

Regarding MuJava, as it can be already observed from Table III, the majority of mutants was represented by the method ones. From this large pool of traditional mutants, the three most commonly encountered were:

1) AOIS, i.e., Arithmetic Operator Insertion, with 13654 mutants,
2) LOI, i.e., Logical Operator Insertion, with 4698 mutants, and
3) ROR, i.e., Relational Operator Replacement, with 3980 mutants.

Javalanche is of real interest in our approach, as it should deal with the equivalent mutant problem. This would allow us to reduce the high number of generated mutants and thus reduce the effort. Therefore, we further try to run the research and, together with the people involved in Javalanche development, come with a solution.

We have to mention that on small and simple projects, i.e., no more than 200 lines of code and which have the test sets in the same project as the mutated classes, we were able to configure and successfully use with success Jumble and Javalanche.

In what follows, we briefly describe the experience recorded for configuring and running the mutation tools we have used in this paper. We denote by $\mathbf{Tool_{Config}}$, i.e., tool configuration, the knowledge accumulated when configuring all the paths; by $\mathbf{Mut_{Gen}}$ we present the mutants generation step and by $\mathbf{Running_{TC}}$ the observations when running the mutants.

1) ***MuJava***
   - $\mathbf{Tool_{Config}}$: The graphical user interface, but also the command line version, are intuitive and easy to use.
   - $\mathbf{Mut_{Gen}}$: The tool must have access to the class files corresponding to each file to be mutated; also, the user can select which mutation operators to apply, both from the set of traditional mutants and also from the class level ones.
   - $\mathbf{Running_{TC}}$: MuJava requires the tests to have the nullary constructor. Also, the methods setUp() and tearDown() must have public access (default is protected). This was time consuming, as we had to update all the test classes with the nullary constructor and the public access for the two methods.
     Both for generating the mutants and then running them it takes a lot of time. A solution may be the integration into an ant script, which is to be run on a monthly basis without any user interaction.
2) ***Jumble***
   - $\mathbf{Tool_{Config}}$: A readme.txt file is available, where the steps to take are quite easy to follow. Nevertheless, after following the instructions and setting the classpath, we were not able to derive a running configuration.
   - $\mathbf{Mut_{Gen}}$: We were not able to get any mutants, due to execution errors.
   - $\mathbf{Running_{TC}}$: Not reachable.
3) ***Javalanche***
   - $\mathbf{Tool_{Config}}$: The javalanche.xml file has to be copied to the current user directory, where it is located the project to be mutated. Then an easy configuration follows, i.e., change the paths to the installation folder and for the working project into the xml file.
   - $\mathbf{Mut_{Gen}}$: Javalanche instruments the byte code and then needs to take control over the test execution. For test execution Javlanche relies on JUnit test suites. If a test suite is not supplied, Javalanche just mutates the code, but it can not

take over the control of the test execution.

- **Running$_{\mathbf{TC}}$**: We did not manage to reach this step.

Regarding the two coverage tools we have used, we found it easy to setup and integrate them into a daily ant script, but also as an Eclipse plug-in (both Clover and EclEmma can be used as Eclipse plug-ins).

## IV. Related research

As mutation testing has proven to be an efficient technique in assessing the quality of the test pool, the attention was focused on whether or not mutation can be used in large scale software applications. In [4], the authors try to answer this problem by running mutation testing over a set of software programs, written in C language, each the size of more than 200 lines of code, and with a large pool of test cases. All the programs had available a pool of faults. The authors were able to show that, when carefully used, the mutation testing technique can provide good results in fault detection.

In [31], the authors proposed a new mutation testing tool, developed in Java and AspectJ for Java programs. They run a research study on real-world open source Java projects, randomly selected, and compare the results with Jumble and MuJava.

What distinguishes our work from the previous ones, is the fact that we take a huge, well known and widely used software project, i.e., Eclipse, and start to record different software metrics. The most important of them is the mutation score metric. For Eclipse we can track the faults database and therefore derive a realistic and practical report of the mutation testing technique, together with other quality software metrics, in order to depict real software bugs. One of the work we report to is the research conducted by Zeller [32].

## V. Conclusion

Mutation testing is an efficient method to detect errors inside the software projects. Unfortunately, the available open source mutation testing tools we have used so far in our research work, have proven to take a lot of time in order to derive all the configuration settings. Although mutation testing can assist in revealing many errors, not all of them represent real actual software failures. The problems mostly encountered with this technique are the complexity to derive the process (as higher the number of generated mutants is, as higher the computation time) and also the equivalent mutant problem.

Each of the above described mutation tools requires different configuration settings. The time effort we have invested just in configuring each tool and then deriving the entire mutation testing technique is now of several months. From the results obtained during the research work, we state that mutation can be regarded as a good software quality metric, but special attention should be given to the drawbacks presented above and, also, to the total amount of time. Meanwhile, setting up the configuration and then running the code coverage tools has proven to be easy to conduct. Based on other previous works, we compare the results given by code coverage with the ones obtained from the mutation testing process.

Through this current research work we start a study, trying to answer the title question: *is Mutation Testing Scalable for Real-World Software Projects?*. We aim to further develop this work, trying also to benefit from the advantage offered by Javalanche: equivalent mutant detection.

## Acknowledgment

The research herein is partially conducted within the competence network Softnet Austria II (www.soft-net.at, COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsfrderungsge-sellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

## References

[1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” vol. 32, no. 8, August 2006, pp. 608–624.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Program Mutation: A New Approach to Program Testing,” in *Infotech State of the Art Report, Software Testing*, 1979, pp. 107–126.

[3] Y.S.Ma, J. Offutt, and Y. R. Kwon, “MuJava : An Automated Class Mutation System,” vol. 15, 2005, pp. 97–133.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?” in *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*, St Louis, Missouri, 15-21 May 2005, pp. 402–411.

[5] A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer,” in *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*, 2-6 October 1994, pp. 824–830.

[6] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Interface Mutation: An Approach for Integration Testing,” vol. 27, no. 3, 2001, pp. 228–247.

[7] U. Praphamontripong and A. J. Offutt, “Applying Mutation Testing to Web Applications,” in *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION’10)*, Paris, France, 6 April 2010, pp. 132–141.

[8] W. Krenn and B. Aichernig, “Test Case Generation by Contract Mutation in Spec#,” in *Proceedings of Fifth Workshop on Model Based Testing (MBT’09)*, York, UK, March 2009, pp. 71–86.

[9] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, Florida, 1-4 November 1999, pp. 210 –219.

[10] V. Okun, "Specification Mutation for Test Generation and Analysis," PhD Thesis, University of Maryland Baltimore County, Baltimore, Maryland, 2004.

[11] W. Ding, "Using Mutation to Generate Tests from Specifications," Master Thesis, George Mason University, Fairfax, VA, 2000.

[12] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford, "The Mothra Tool Set," in *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences (HICSS'22)*, 3-6 January 1989, pp. 275–284.

[13] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," vol. 21, no. 7, October 1991, pp. 685–718.

[14] Y. Ma, A. J. Offutt, and Y. Kwon, "MuJava: a Mutation System for Java," in *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, Shanghai, China, 20-28 May 2006, pp. 827–830.

[15] D. Schuler and A. Zeller, "Javalanche: Efficient Mutation Testing for Java," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, Amsterdam, Netherlands, 24-28 August 2009, pp. 297–298.

[16] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. J. Inglis, and M. Utting, "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests," in *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, Windsor, UK, 10-14 September 2007, pp. 169–175.

[17] I. Moore, "Jester - a JUnit test tester," in *Proceeding of eXtreme Programming Conference (XP'01)*, 2001.

[18] PIT Mutation Testing, "http://pitest.org/," 2011.

[19] A. Derezinska and A. Szustek, "CREAM- A System for Object-Oriented Mutation of C# Programs," Warsaw University of Technology, Warszawa, Poland, Technical Report, 2007.

[20] Nester, "http://nester.sourceforge.net/," 2011.

[21] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, Windsor, UK, 29-31 August 2008, pp. 94–98.

[22] J. Tuya, M. J. S. Cabal, and C. de la Riva, "SQLMutation: A Tool to Generate Mutants of SQL Database Queries," in *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, Raleigh, North Carolina, November 2006, p. 1.

[23] Eclipse, ":pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse," 2011.

[24] M. D. Site, "http://cs.gmu.edu/~offutt/mujava/," 2011.

[25] Jumble, "http://jumble.sourceforge.net/," 2011.

[26] Javalanche, "http://www.st.cs.uni-saarland.de/~schuler/javalanche/download.html," 2011.

[27] Clover, "http://www.atlassian.com/software/clover/," 2011.

[28] EclEmma, "http://www.eclemma.org/," 2011.

[29] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, April 2010, pp. 45–54.

[30] JDT, "http://www.eclipse.org/jdt/," 2011.

[31] L. Madeyski and N. Radyk, "Judy a mutation testing tool for java."

[32] R. Premraj and A. Zeller, "Predicting Defects for Eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07, 2007, p. 9.