Automated Mutation Testing using JDT and AST

Dhrupad Kaneria

Graduate Student, UT Dallas  
800 W Campbell Rd  
Richardson TX 75080

dhrupad.kaneria@utdallas.edu

Purva Dahake

Graduate Student, UT Dallas  
800 W Campbell Rd  
Richardson TX 75080

purva.dahake@utdallas.edu

**ABSTRACT**

This report provides an introduction to Mutation Testing, Eclipse JDT and AST. It gives an approach to write a program to generate mutants for a given project using JDT while taking the help of AST without making any changes to the actual code. One real word java program (>1000 likes of code) with JUnit tests (>50 tests) is taken into consideration and mutation testing is carried out on this project. The process will generate mutants and the mutants will be executed for the project to determine the quality and correctness of the unit test cases for the project. Depending on the number of test cases passed, the mutants are killed and the mutation score is calculated.

# INTRODUCTION

## Mutation Testing

**Mutation testing** [1] (or *Mutation analysis* or *Program mutation*) is used to design new software tests and evaluate the quality of existing software tests. Mutation Testing is a fault-based technique that measures the effectiveness of the test suites for fault localization, based on seeded faults. Mutation testing involves modifying a program in small ways. Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant.

## Eclipse JDT

JDT [2] is a framework that provides provision to manipulate the java source code. Since it is generally attached to Eclipse, it is often called an Eclipse JDT. The JDT is broken down into components. Each component operates like a project unto its own.

## JUnit Tests

JUnit [3] is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit. A research survey across 10,000 java projects hosted on GitHub found that JUnit was most commonly used. It is was used by 30.7% of the projects.

# PROBLEM

Tests are created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. In order to answer this problem, we tend to undergo the verification of test cases. This verification is done by Mutation Testing.

Each test case is executed on all the mutants that are generated. The outcome of the execution (number of mutants that are still alive) will determine more about the quality and correctness of the test cases written.

# BASICS OF EXISTING TECHNIQUE

The important part of mutation testing is to induce defects by making small changes in the current program. This is mostly done by changing the operands, or by changing the operands. Assuming the average size of the project is 1000 lines, changing all the operators and then running the program manually is practically impossible. Hence there are certain tools available in the market that generates mutants and gives the result. The widely used mutation testing tools are described and introduced in the below sections.

## Pit

Pit [4] is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It is fast, scalable and integrates with modern test and build tooling. Pit runs the unit tests against automatically modified versions of the application code. Pit is actively developed and supported. As a result, it is one of the widely used tool for mutation testing.

## µJava

µJava is a mutation system for Java programs. µJava uses two types of mutation operators, class level and method level. It creates object-oriented mutants for Java classes according to 24 operators that are specialized to object-oriented faults. Method level (traditional) mutants are based on the selective operator set. After creating mutants, µJava allows the tester to enter and run tests, and evaluates the mutation coverage of the tests.

# ANALYSIS PLAN

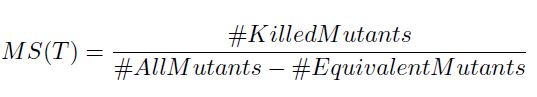
We intend to analyze a real world Java project with more than 1000 lines of code and more than 50 JUnit tests.

## Mutation Score

The mutants are divided into equivalent and non-equivalent mutants or live and killed mutants. The relation between the four types of mutants is given by the following diagram.



The relation between these mutants is given by the mutation score. The mutation score [5] is defined as the percentage of killed mutants with the total number of mutants.



Since it is very difficult to find the number of equivalent mutants, we have made a silent change in the formula and is given by:

*MS (T) =*

Though this is not the exact estimation, this helps in providing a decent knowledge about the quality of the test cases written.

## Mutation Strategy

The mutants can be generated based on different changes. The original program code is changed only at one line. If more than one change is induced, it is called Higher Order Mutation Testing. After going through many articles and papers on mutation testing, we have come up with the following strategies.

|  |  |
| --- | --- |
| **Old Operator** | **New Operator** |
| < | <= |
| <= | < |
| > | >= |
| >= | > |
| != | = |
| = | != |
| || | && |
| && | || |
| \* | - |
| - | \* |

# EXPERIMENTAL EVALUATION SUBJECTS

The experimental subject on which the mutation testing is being tested out is selected form the list of the projects provided. The project is called parse4j. It consists of 92 test cases and over 1000 lines of code spread across different packages and files. Out of the 92 test cases, 9 test cases had failed the execution and 11 test cases are errored. After understanding the implementation of the test cases and scenario being tested, we learnt that these are the test cases for testing the negative impact i.e. these test cases were written to test the exception handling for invalid test inputs.

# PSEUDO CODE

This section will have the pseudo code and the algorithm used in implementing the project. The entire project is divided into mainly three different stages namely generating mutant, running the test cases and calculating mutation score.

## Approach

## Stage 1:

The first step in the project was to select the appropriate specimen for mutation testing. The criteria for selecting the examination specimen are:

* it should be sufficiently large (>1000 lines of code)
* it should have more than 50 unit test cases.
* the examination specimen should be able to be executed programmatically.

After selecting the experimental specimen (the master project – parse4j), it was examined for the given specifications. The test cases were executed for the given project using maven. The observations returned were noted down for future analysis. We then started writing the java code to access this subject taking the help of Eclipse JDT with the AST.

## Stage 2:

The second step in the project was to access the specimen and store it as an Abstract Syntax Tree (AST). After successfully generating the AST, the next step was to run the unit test cases on the actual program (the AST format). The result of the execution (the number of tests passed, failed) is noted down. This execution was carried on the actual java program before generating any mutants.

## Stage 3:

The third step was to create a mutant. In this step, the copies of actual project were made for the mutants. In each copy, the changes were made and mutants were generated. By using ASTVisitor class provided by the AST, the ifStatement and whileStatements were visited and their expressions were changed. From the criteria discussed previously, the operators were changed and saved. These changes were also stored and verified against before making the change. If the change was already applied, the next line was changed.

Once all the mutants are generated, the next step was to execute the junit test cases on these mutants. At the end of each execution, the tests that passed and failed are noted down. Depending on the result of the execution, the mutants are either classified as dead or alive. This statistics will help us to determine how strong and accurate the unit tests are.

C:\Users\user\Desktop\STVV - 1.png

# OBERVATIONS

This section will have the observations that are noticed while implementing the project.

# CONCLUSION

This section will have the conclusion, key finding and learnings from the project.

# REFERENCES

1. Mutation Testing, <https://en.wikipedia.org/wiki/Mutation_testing>
2. Eclipse JDT, <https://eclipse.org/jdt/>
3. JUnit, <https://en.wikipedia.org/wiki/JUnit>
4. PIT, <http://pitest.org/>
5. Mutation Score, <http://www.guru99.com/mutation-testing.html>
6. µJava, <http://cs.gmu.edu/~offutt/mujava/>