Automated Mutation Testing Framework using JDT

Dhrupad Kaneria
Graduate Student, UT Dallas
800 Campbell Rd
Richardson TX 75080
214-299-3874
dhrupad.kaneria@utdallas.edu

Purva Dahake
Graduate Student, UT Dallas
800 Campbell Rd
Richardson TX 75080
469-450-9959
purva.dahake@utdallas.edu

ABSTRACT

This report provides an introduction to Mutation Testing and Eclipse JDT. IT gives an approach to write program to generate mutants for a given project using JDT 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.

1. INTRODUCTION

1.1 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.

1.2 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.

1.3 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.

2. 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.

3. 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.

3.1 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.

3.2 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.

4. ANALYSIS PLAN

We intend to analyze a real world Java project with more than 1000 lines of code and more than 50 JUnit tests. We plan to write a Java program with the help of Eclipse JDT and manipulate the source code of the project under consideration. With the help of Eclipse JDT, we shall change the operators to start with and generate mutants. These mutants will be executed as a project and checked if it is killed by any test. Based on what mutants are killed, the quality of the unit tests will be obtained.

To further create more mutants, the operands will also be changed. This will give a more precise information about the quality and effectiveness of the unit tests. Our aim is to generate mutants by applying commonly used Mutation operators for Java. The more mutant we generate, the more detailed result is obtained.

By generating all the mutants and executing the test cases on them, we store the number of mutants that are alive after the execution. Once the count is noted, we determine and come to a solid conclusion on the quality and correctness of the unit test cases written in the project.

4.1 Mutation Score

The mutation score [5] is defined as the percentage of killed mutants with the total number of mutants.

Test cases are mutation adequate if the score is 100%. Experimental results have shown that mutation testing is an effective approach for the measuring the adequacy of the test cases. But, the main drawback is that the high cost of generating the mutants and executing each test case against that mutant program.

5. EXPERIMENTAL EVALUATION SUBJECTS

The project that will become a part of the analysis will be found from GitHub. We are currently under consideration before finalizing the project for the analyzing. Facts about the input should be considered before deciding the project. We are currently analyzing the facts and size of the project before making a final decision.

6. 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/