Software Testing Report

Mutation Testing

Atul Tripathi

MT2023084

Project Objective

This project focuses on understanding and applying the concept of Mutation testing by integrating a specialized tool into a personal project. We chose the PIT mutation Testing tool due to its ease of use in Java applications.

The broader aim of the project is to explore how Mutation Testing enhances the quality of test cases and improves software reliability.

Source Code Overview

The project leverages a custom Java-based algebra library as its source code. This library, hosted on GitHub (https://github.com/danhales/linearalgebra) includes to core classes, Vector and Matrix. These classes implement a range of linear algebra operations and are fundamental in domains like data analysis and machine learning, For those familiar with Python's NumPy, this library provides an analogous interface tailored for Java, with around 1,800 lines of code offering significant functionality.

Exploring Mutation Testing

Mutation Testing evaluates the effectiveness of a test suite by introducing deliberate changes (mutations) into the code to simulate common programming errors. These mutations might include logical errors, misused operators, or alterations in conditional statements.

The testing process involves:

1. Generating mutated versions of the original source code.
2. Running the test suite against both the original and mutated code.
3. Identifying "killed" mutants (where the test suite detects mutations) versus "surviving" mutants (undetected errors).

This methodology helps uncover weaknesses in the test suite, such as untested code paths or insufficient edge-case handling. The insights gained allow developers to refine their tests for improved defect detection and better coverage.

Importance of Mutation Testing

Unlike traditional code coverage metrics, which merely indicate which lines of code are executed during testing, Mutation Testing provides a deeper evaluation of a test suite's ability to identify faults. This makes it particularly useful in ensuring the reliability of mission-critical software.

Types of Mutant Killing

In Mutation Testing, a mutant can be "killed" in one of two ways:

1. **Weak Killing**: The program’s internal state changes due to a mutation, but the final output remains unaffected.
2. **Strong Killing**: The mutation causes a noticeable difference in the program’s output compared to the original version.

Our focus in this project is to design robust test cases capable of **strongly killing** all mutants, ensuring that potential bugs do not go unnoticed.

Unique Insights from This Project

Through this project, we not only validated the functionality of a linear algebra library but also gained insights into the practical applications of Mutation Testing in real-world scenarios. By analyzing the outcomes, we identified ways to improve test coverage, reduce false positives, and strengthen test suites for future projects.

Mutation Coverage Report

The coverage report is shown below:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	78%	298/381	70%	675/969	89%	675/757

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example	2	78%	298/381	70%	675/969	89%	675/757

Report generated by PIT 1.17.1

Enhanced functionality available at arcmutate.com

Pit Test Coverage Report

Package Summary

com.example

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	78%	298/381	70%	675/969	89%	675/757

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Matrix.java	79%	207/263	69%	472/682	87%	472/542
Vector.java	77%	91/118	71%	203/287	94%	203/215

Report generated by PIT 1.17.1

References:

1. This project's git repository – https://github.com/atul55tripathi/software-testing
2. Maven - https://maven.apache.org/guides/getting-started/
3. PIT in maven - https://pitest.org/quickstart/maven/
4. Junit - https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html