International Institute of Information Technology,
Bangalore

Software Testing : CS731

Mutation Testing on Various DSA problems

Kuldip Bhatale

MT2023087

Aakash Bhardwaj

MT2023143

Github Code Repo [Repo]

Report on Mutation Testing

1. Introduction

Mutation testing is an advanced software testing technique where small modifications, called *mutations*, are introduced into a program's source code to evaluate the effectiveness of its test suite. The primary objective of mutation testing is to assess the quality and robustness of the tests by ensuring they can detect the introduced faults. If a mutation is not detected by the test suite, it indicates potential gaps in the tests. As a white-box testing technique, mutation testing focuses on analyzing the internal structure of the program.

1.1 Why Mutation Testing?

Traditional test coverage metrics, such as line, statement, and branch coverage, only measure the extent of the code executed during testing. While they indicate which parts of the code are exercised, they fail to evaluate the ability of tests to detect errors. Mutation testing addresses this limitation by introducing deliberate faults (mutations) into the code and evaluating whether the test suite catches them. This method provides a more comprehensive measure of test effectiveness and identifies weaknesses in test cases, prompting improvements in quality.

1.2 Tools Used

Java (PIT)

PIT (Pitest) is a leading mutation testing tool for Java and JVM-based applications. It is fast, scalable, and integrates seamlessly with modern testing and build tools like Maven and Gradle.

- **How PIT works**:
  PIT runs the existing unit tests against altered versions of the application code. These altered versions, called mutants, represent small, intentional code changes.
    - If the application code changes cause the test suite to fail, the mutant is “killed.”
    - If the test suite does not detect these changes, the mutant “survives,” signaling an issue with the test suite.

PIT is highly efficient, making it the preferred tool for mutation testing in Java environments.

1.3 Objective of Mutation Testing

The primary objectives of mutation testing are:

- **Identifying inadequately tested code:** It highlights code areas not sufficiently validated by the current test cases.
- **Discovering hidden defects:** It uncovers defects that may be missed by other testing methods.
- **Improving error detection:** By identifying gaps in test coverage, it aids in catching new or unconventional errors.

- **Assessing test case quality:** It evaluates the thoroughness and reliability of the test suite.

1.4 How Mutation Testing Works

The mutation testing process can be broken down into the following steps:

1. **Create Mutants:**
   The source code is altered to generate mutant versions by:
   - Changing arithmetic operators (e.g., `+` to `-`)
   - Flipping conditional logic (e.g., `>` to `<`)
   - Modifying constants and control structures.
2. **Run Tests:**
   The test suite is executed against each mutant version of the code.
3. **Evaluate Test Effectiveness:**
   - If a mutant causes the test to fail, it is considered “killed.”
   - If the mutant does not trigger any test failures, it “survives,” indicating potential gaps in the test suite.
4. **Assess Test Quality:**
   A high mutation kill rate suggests effective tests, while surviving mutants indicate areas for improvement.

1.5 Levels of Mutation Testing

Mutation testing can be applied at different levels:

1.5.1 Unit Mutation

- Focuses on testing individual units or functions in isolation.
- Introduces mutations within single units, such as altering arithmetic operations or modifying conditions.
- Example: Changing a `+` operator to a `-` operator in a method or swapping logical conditions in a function.
- Goal: Validate the correctness of unit-level behaviors and ensure unit tests can detect faults.

1.5.2 Integration Mutation

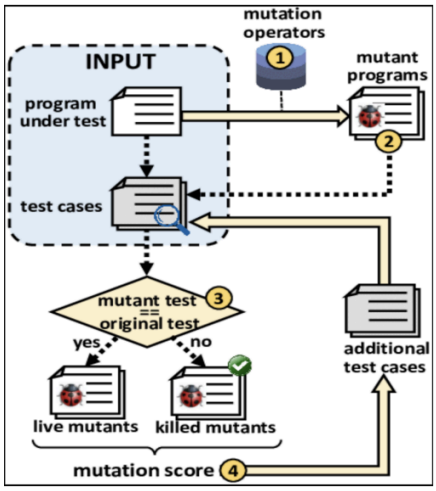
- Focuses on interactions between different components or modules in the system.
- Mutations are introduced at integration points, such as method calls or data exchanges between modules.
- Example: Modifying how data is passed between methods or altering logic in the communication between components.
- Goal: Ensure the correctness of integration-level behaviors and validate that integration tests detect faults in module interactions.

1.6 Mutation Score

The **Mutation Score** is a quantitative metric used to assess the effectiveness of a test suite in identifying faults.

$$\text{Mutation Score} = \left(\frac{\text{Killed Mutants}}{\text{Total Number of Mutants}} \right) \times 100$$

- A higher mutation score reflects a more effective test suite.
- A low mutation score suggests that improvements are needed to detect faults more comprehensively.



---

2. Source Code

Our code base contains approximately 1,250 lines and is organized into various functional modules:

1. Unit Testing in JAVA(DSA)

Files:

- `Main.java`: 465 lines
- `AppTest.java`: 495 lines

Unit and Integration Testing in Java (Food-delivery)

Files:

- **App.java**: 165 lines
- **AppTest.java**: 125 lines

Total = 1250(approx.)

```
1   package org.example;
2
3   class MathUtils {
4
5       // Method to check if a number is prime
6       public boolean isPrime(int n) {
7   3       if (n <= 1) return false; // 0, 1, and negatives are not prime
8   2       for (int i = 2; i <= Math.sqrt(n); i++) {
9   3           if (n % i == 0) return false;
10          }
11  1       return true;
12      }
13
14      // Method to check if a number is a Fibonacci number
15      public boolean isFibonacci(int n) {
16  3       if (n < 0) return false;
17  3       int x1 = 5 * n * n + 4;
18  3       int x2 = 5 * n * n - 4;
19  3       return isPerfectSquare(x1) || isPerfectSquare(x2);
20      }
21
22      private boolean isPerfectSquare(int x) {
23          int s = (int) Math.sqrt(x);
24  3       return s * s == x;
25      }
26
27      // Method to calculate the factorial of a number
28      public int factorial(int n) {
29  3       if (n < 0) return -1; // Undefined for negative numbers
30          int result = 1;
31  2       for (int i = 1; i <= n; i++) {
32  1           result *= i;
33          }
```

	Mutations
7	1. replaced boolean return with true for org/example/MathUtils::isPrime → KILLED 2. negated conditional → KILLED 3. changed conditional boundary → KILLED
8	1. negated conditional → KILLED 2. changed conditional boundary → KILLED
9	1. replaced boolean return with true for org/example/MathUtils::isPrime → KILLED 2. Replaced integer modulus with multiplication → KILLED 3. negated conditional → KILLED
11	1. replaced boolean return with false for org/example/MathUtils::isPrime → KILLED
16	1. negated conditional → KILLED 2. replaced boolean return with true for org/example/MathUtils::isFibonacci → KILLED 3. changed conditional boundary → KILLED
17	1. Replaced integer multiplication with division → KILLED 2. Replaced integer addition with subtraction → KILLED 3. Replaced integer multiplication with division → KILLED
18	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer multiplication with division → KILLED 3. Replaced integer multiplication with division → KILLED
19	1. replaced boolean return with true for org/example/MathUtils::isFibonacci → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
24	1. replaced boolean return with true for org/example/MathUtils::isPerfectSquare → KILLED 2. negated conditional → KILLED 3. Replaced integer multiplication with division → KILLED
29	1. replaced int return with 0 for org/example/MathUtils::factorial → KILLED 2. changed conditional boundary → KILLED 3. negated conditional → KILLED
31	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
32	1. Replaced integer multiplication with division → KILLED
34	1. replaced int return with 0 for org/example/MathUtils::factorial → KILLED
39	1. negated conditional → KILLED 2. changed conditional boundary → KILLED 3. replaced boolean return with true for org/example/MathUtils::isArmstrong → KILLED

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Mutation operators applied

1. Arithmetic Operator Mutation

- **Description:** Replace arithmetic operators (`+`, `-`, `*`, `/`, `%`) with others.
- **Example:**
  - `a + b → a * b` (addition replaced with multiplication).
  - `x / y → x - y` (division replaced with subtraction).

---

2. Relational Operator Mutation

- **Description:** Replace relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) with others.
- **Example:**
  - `a != b → a == b` (not-equal replaced with equal).
  - `x > y → x <= y` (greater-than replaced with less-than-or-equal).

---

3. Logical Operator Mutation

- **Description:** Replace logical operators (`&&`, `||`) with others.
- **Example:**
  - `a && b → a || b` (AND replaced with OR).
  - `x || y → x && y` (OR replaced with AND).

---

4. Conditional Operator Mutation

- **Description:** Alter conditions in control structures (e.g., `if`, `while`, `for`).

- **Example:**
  - `if (a > b)` → `if (a == b)` (greater-than replaced with equality).
  - `while (x < y)` → `while (x >= y)` (condition reversed).

---

5. Negation Mutation

- **Description:** Negate expressions or boolean conditions.
- **Example:**
  - `if (a > b)` → `if (!(a > b))` (positive condition replaced with negation).
  - `x && y` → `!(x && y)` (negated logical AND).

---

6. Constant Replacement Mutation

- **Description:** Replace constants or literals with other values.
- **Example:**
  - `return 42` → `return 0` (numeric constant replaced).
  - `limit = 10` → `limit = -1` (positive constant replaced with a negative value).

---

7. Variable Replacement Mutation

- **Description:** Replace one variable with another of a compatible type.

- **Example:**
  - `a = b` → `a = c` (variable `b` replaced with `c`).
  - `x = y` → `x = z` (similar replacement in assignment).

---

8. Bitwise Operator Mutation

- **Description:** Replace bitwise operators (`&`, `|`, `^`, `~`) with others.
- **Example:**
  - `a & b` → `a | b` (AND replaced with OR).
  - `~a` → `a ^ b` (NOT replaced with XOR).

---

9. Unary Operator Mutation

- **Description:** Modify unary operators like increment (`++`), decrement (`--`), and negation (`-`).
- **Example:**
  - `i++` → `i--` (post-increment replaced with post-decrement).
  - `-a` → `+a` (negative sign replaced with positive).

- **MathUtility:** A collection of mathematical functions.
- **Employee Management:** Functions related to employee data handling and processing.
- **Bank Management:** Modules managing customer accounts, transactions, and banking operations.

- **E-commerce Services:** Handles product catalogs, customer orders, and payment processes.

Testing Details:

- **Manual Unit Tests:** Written for `MathUtility`, `Employee`, and `Bank Management` functions.
- **Integration Tests:** Developed for the e-commerce services module to validate inter-module functionality.

---

3. Results

Code Analysis:

- Total Lines of Code: **1,200**
- Modules Tested: **MathUtility, Employee, Bank Management, E-commerce Services**

Mutation Testing Summary:

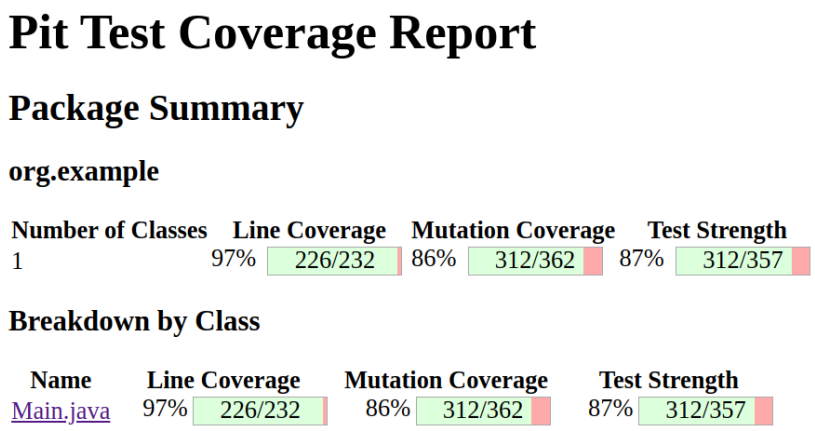
- **Total Mutants Generated:** 411
- **Killed Mutants:** 354
- **Surviving Mutants:** 58

Mutation Score: 86%

Key Observations:

- **Strengths:**
  - Unit tests for `MathUtils`, `BitwiseUtils and UnaryOperator` achieved a kill rate of 86%, showcasing robust test coverage.
  - Integration tests for the food-delivery module demonstrated a kill rate of 86%, reflecting moderate effectiveness.
- **Areas for Improvement:**
  - Surviving mutants in the food-delivery module suggest the need for additional test cases to cover edge cases in data interactions.
  - Enhanced testing strategies are needed for boundary conditions in the food-delivery.

Figures:



Pit Test Coverage Report

Package Summary

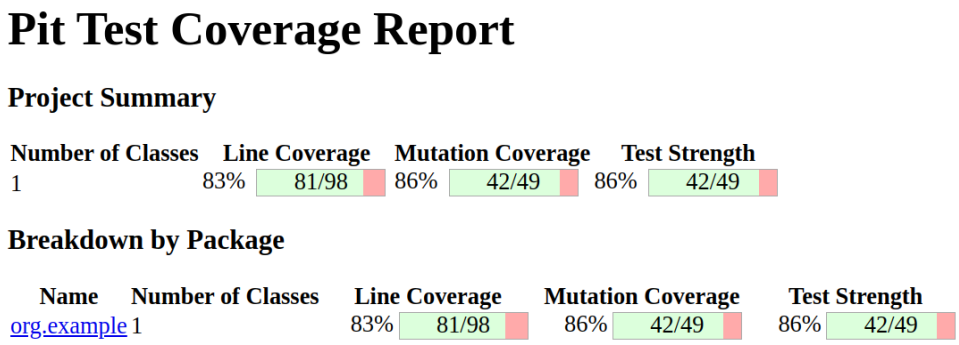
org.example

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	97%	226/232	86%	312/362	87%	312/357

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Main.java	97%	226/232	86%	312/362	87%	312/357

Figure 1: Visual representation of mutation results of unit testing.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	83%	81/98	86%	42/49	86%	42/49

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example	1	83%	81/98	86%	42/49	86%	42/49

Figure 2: Visual representation of mutation results of Integration testing.

---

4. Conclusion

Mutation testing is a powerful method to evaluate and enhance the quality of software tests. By deliberately introducing faults and observing test responses, it provides a rigorous measure of test suite effectiveness.

- **Effectiveness:** With a mutation score of 85%, our test suite demonstrates substantial robustness, though room for improvement remains in integration testing.
- **Action Plan:** Focus on refining integration tests and expanding coverage for edge cases to achieve a higher mutation score.

Mutation testing not only highlights weaknesses in tests but also ensures the reliability and maintainability of the codebase, ultimately contributing to a more robust software system.

Team Contributions:

We collaboratively worked on the mutation testing process for both Java and food-delivery applications, ensuring comprehensive test coverage and improving the robustness of the test suites.