Mutation Testing on Programming Problems

Tarun Sharma
MT2023043

Farman Ahmed
MT2023129

Instructor: **Prof. Meenakshi D Souza**

Date - 26 November 2024

Contents

1 Introduction to Mutation Testing

Mutation testing [1] is a powerful software testing technique aimed at evaluating the quality of test cases by introducing small changes, called **mutations** [2], into the source code of a program. These changes simulate potential defects, and the goal is to determine whether the existing test suite can detect these mutations. If the test suite fails to detect a mutation, it indicates a potential weakness in the tests.

1.1 Key Concepts

- **Mutants**: These are the altered versions of the program created by applying specific mutation operators. Each mutant introduces a single change to the program's logic.
- **Mutation Operators**: These are predefined rules or patterns used to create mutants. Common examples include replacing arithmetic operators (‘+‘ with ‘-‘), logical operators (‘`&&`‘ with ‘`||`‘) , or altering conditional boundaries.
- **Kill a Mutant**: A mutant is considered killed if the test suite fails when executed on the mutated version of the code. If a mutant is not killed, it suggests either a missing or ineffective test case.

1.2 Purpose of Mutation Testing

The primary purpose of mutation testing is to measure the **effectiveness of the test suite**. It helps answer questions like:

- Are the test cases comprehensive enough to catch subtle errors?
- Which parts of the code are poorly tested?

1.3 Advantages and Challenges

Advantages:

- Helps in identifying weak test cases.
- Ensures high test coverage and robust software.

Challenges:

- Computationally expensive, especially for large codebases.
- May produce equivalent mutants (mutations that do not alter program behavior), which require manual intervention to detect.

1.4 Illustration

Below is a visual representation of the mutation testing process:

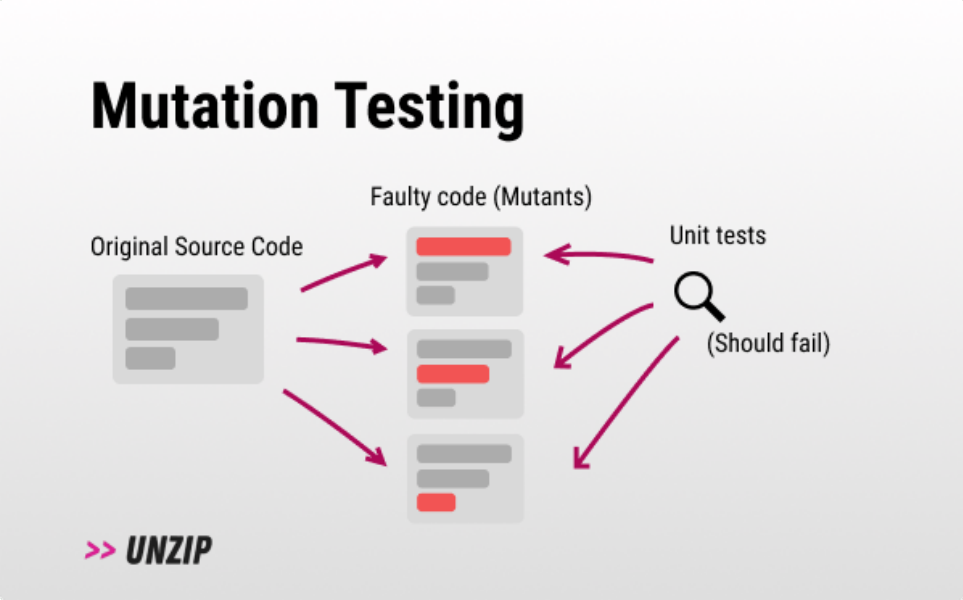


Figure 1: The Mutation Testing Process: Mutants are created from the source code and tested against the test suite.

1.5 Levels of Mutation Testing

Mutation testing operates at different levels depending on the scope and granularity of the mutations:

- **Unit Level:** [3] Mutations are introduced within individual functions or methods. This is the most granular level and is often used for testing specific components of the program.
- **Class/Module Level:** Mutations are applied to classes or modules, testing interactions within these components.
- **System Level:** Mutations are introduced across the entire system, including interactions between modules, external APIs, and the database.

1.6 Tools Used in Mutation Testing

1.6.1 PITest for Java

PITest (also known as PIT) [4] is a leading mutation testing tool for Java applications. It integrates seamlessly with modern development workflows and helps developers identify weaknesses in their test suites.

- **Features:**
  - Automates the process of introducing and testing mutations in Java code.
  - Generates a detailed mutation report with information on mutants killed, survived, or untested.
  - Supports integration with popular build tools like Maven and Gradle.

- **How It Works:** PITest modifies bytecode to introduce mutants and reruns existing test cases to observe their effectiveness. Surviving mutants indicate that specific test cases did not detect the changes, suggesting a potential gap in the test coverage.
- **Advantages:**
  - Easy to configure and use.
  - Provides insights into the robustness of unit tests.
  - Helps maintain high code quality by uncovering missed edge cases.
- **Limitations:** While highly effective, PITest might produce false positives or struggle with equivalent mutants that do not alter the program's observable behavior.

1.7 Mutation Score

The effectiveness of mutation testing is quantified using the **Mutation Score**, calculated as follows:

$$\text{Mutation Score} = \frac{\text{Number of Mutants Killed}}{\text{Total Number of Mutants Introduced}} \times 100$$

Where:

- **Mutant Killed:** A test case fails due to the mutation, indicating that the test detected the fault.
- **Total Mutants Introduced:** The total number of modifications (mutants) made to the program during the testing process.

A high mutation score indicates that the test suite is effective at detecting introduced faults. However, achieving a 100% mutation score may not always be feasible due to equivalent mutants (mutations that do not change the program's observable behavior).

2 Source Code

[5] We have divided our tasks into two one is Unit testing and second one is Integration testing.

2.1 Unit

This subsection describes the core functionalities implemented in individual units of the program. The code is designed to solve specific problems effectively, ensuring each unit is independently testable.

Building Teams

The 'BuildingTeams' class solves a bipartite graph coloring problem, where the goal is to divide graph nodes into two teams such that no two connected nodes belong to the same team. It uses a breadth-first search (BFS) approach to traverse the graph and assigns teams alternately. If a conflict is detected (i.e., two connected nodes have the same team), the method returns "IMPOSSIBLE". Otherwise, it outputs the valid team assignments for all nodes.

Coin Change

The ‘CoinChange‘ class implements a recursive solution with memoization for the classic coin change problem. The method ‘solve‘ calculates the minimum number of coins required to achieve a given target sum (‘k‘) using a list of coin denominations (‘arr‘). If the target sum is zero, the result is zero. If the target is negative, it returns an arbitrarily large value to denote infeasibility. Memoization is achieved through a ‘dp‘ array to store results of previously computed subproblems, reducing redundant calculations. This approach efficiently handles overlapping subproblems and minimizes the overall computation time for larger inputs.

Dice Combinations

The ‘DiceCombinations‘ class solves a problem where the task is to determine the number of ways to achieve a given target sum (‘n‘) using dice rolls. It uses dynamic programming with an array ‘dp‘, where ‘dp[i]‘ stores the number of ways to achieve the sum ‘i‘. The solution iterates through all possible dice outcomes (1 to 6) for each sum and aggregates the results while applying a modulo operation (‘MOD = 10^9+ 7‘) to handle large numbers efficiently. The method returns the total combinations for the target sum, ensuring a time-efficient and scalable solution.

Grid Paths

The ‘GridPaths‘ class solves the problem of finding the number of distinct paths in a grid from the top-left corner to the bottom-right corner, while avoiding blocked cells (represented by ‘*‘). It employs dynamic programming to calculate the number of paths at each grid cell (‘dp[i][j]‘). The method ‘solve‘ uses recursion with memoization to traverse the grid. It calculates paths moving either down or right, ensuring that only valid moves are made (i.e., moves that don’t go outside the grid or through blocked cells). The result is computed modulo $10^9 + 7$ to manage large numbers. The function returns the total number of paths from the starting point to the destination.

Message Route

The ‘MessageRoute‘ class finds the shortest path for a message to travel from node 1 to node n in an undirected graph. The graph is represented as an adjacency list, and the class uses Breadth-First Search (BFS) to explore the graph. The method ‘findMessageRoute‘ performs BFS, tracking the parent of each node to reconstruct the path once the destination is reached. The result includes the length of the path followed by the nodes in the path. If no path exists, the method returns ”IMPOSSIBLE”. This approach ensures that the shortest route is found due to the properties of BFS, where the first time a node is reached, it is via the shortest path.

Round Trip

The ‘RoundTrip‘ class is designed to determine if a graph is bipartite and assign nodes to two groups accordingly. The graph is represented as an adjacency list, and the method ‘determineBipartite‘ uses a BFS (Breadth-First Search) approach to traverse the graph. It attempts to color each node in two colors (1 or 2) to ensure that no two adjacent nodes have the same color. If such a coloring is possible, the graph is bipartite, and the method

returns the group assignments for each node. If a conflict is detected (i.e., adjacent nodes have the same color), it returns "IMPOSSIBLE". The BFS ensures that all components are processed independently, and the graph is checked for bipartiteness by alternating colors.

2.2 Integration

This section contains code implementations that integrate multiple components or modules to verify their combined behavior. Integration testing ensures that units like 'BuildingRoads' interact correctly with other dependent modules, such as input-output handling or application workflows. Further details will be covered in subsequent code summaries.

Building Roads

The 'BuildingRoads' class is designed to work with undirected graphs, identifying connected components and suggesting minimal connections to ensure full connectivity. It uses an adjacency list for graph representation and depth-first search (DFS) to traverse and explore nodes. This approach helps count isolated components and provides a way to generate a set of edges that can connect these components efficiently.

Grid Components

The 'GridComponents' class is designed to count the number of connected components in a 2D grid. The grid is represented as a matrix where each cell can either be empty ('.') or blocked ('#'). The method 'countComponents' iterates through the grid and uses Depth-First Search (DFS) to explore and mark all cells that are part of the same connected component (i.e., all reachable empty cells). It tracks the number of components by incrementing a counter each time a new DFS is initiated. The directions for movement (up, down, left, right) are defined in the 'dr' and 'dc' lists. This approach ensures that all connected empty cells are grouped together, and the number of such components is returned.

Labyrinth

The 'Labyrinth' class is used to find the shortest path in a grid from the start point ('A') to the end point ('B') using the Breadth-First Search (BFS) algorithm. The grid is represented as a 2D array where walls are marked with '#' and empty spaces are navigable. The class tracks visited cells to avoid revisiting and uses a direction tracker array to store the direction taken to reach each cell. The 'GridPathFinder' method initializes the grid and necessary data structures, while the 'bfs' method performs the BFS to find the shortest path from 'A' to 'B'. If a path is found, the method reconstructs the path by backtracking through the direction tracker. The 'getPath' method returns the reconstructed path in the correct order by reversing the path list. If no path is found, the method returns 'false'. This approach ensures that the shortest path is found in an efficient manner.

LineCounter

The 'LineCounter' class is designed to count the total number of lines of code in '.java' files within a specified directory and its subdirectories. The class contains two main

methods: ‘countLinesInDirectory‘ and ‘countLinesInFile‘. The ‘countLinesInDirectory‘ method recursively traverses the directory and its subdirectories, calling ‘countLinesInFile‘ for each ‘.java‘ file it encounters. The ‘countLinesInFile‘ method reads each line of a ‘.java‘ file using a ‘BufferedReader‘ and counts the total number of lines. The result is the cumulative line count of all ‘.java‘ files in the directory and its subdirectories. If an error occurs while reading a file, an ‘IOException‘ is caught and printed to the console.

StaticRangeMinimumQueries

The ‘StaticRangeMinimumQueries‘ class implements a segment tree to efficiently perform range minimum queries (RMQ) on a static array. The class supports three main operations: building the segment tree, querying the minimum value in a specified range, and updating an element in the array. The segment tree is built during initialization, with each node storing the minimum value of a corresponding segment of the array. The ‘query‘ method efficiently retrieves the minimum value in a given range using the segment tree. The ‘update‘ method updates a specific element in both the array and the segment tree. This approach ensures fast querying and updating, making it suitable for applications that require frequent range minimum queries on static data.

StaticRangeSumQueries

The ‘StaticRangeSumQueries‘ class implements a segment tree to efficiently perform range sum queries on a static array. It supports building the segment tree, querying the sum of elements in a specified range, and updating elements. The segment tree is initialized during construction, with each node storing the sum of a segment of the array. The ‘query‘ method efficiently retrieves the sum of elements in a specified range by leveraging the segment tree’s structure. This approach allows for fast sum queries on static data, making it ideal for applications that require frequent range sum computations.

2.3 LOC (Lines Of Code excluding documentation and inferences)

- Programs: 620 lines
- Tests: 1821 lines

2.4 Mutation Operators Used as per Class Notes

We will only cover some of the Mutations performed since there are around 200 mutation and covering them wont be possible in this Report.

MessageRoute.java

```
1   package org.example;
2   import java.util.*;
3
4   public class MessageRoute {
5
6       public String findMessageRoute(int n, int m, List<int[]> edges) {
7           // Adjacency list
8   1       List<List<Integer>> adj = new ArrayList<>();
9   5       for (int i = 0; i <= n; i++) {
10  2           adj.add(new ArrayList<>());
11          }
12
13          // Add edges to adjacency list
14          for (int[] edge : edges) {
15  2           int a = edge[0], b = edge[1];
16  3           adj.get(a).add(b);
17  3           adj.get(b).add(a);
18          }
19
20          // BFS variables
21  1       Queue<Integer> queue = new LinkedList<>();
22  2       boolean[] vis = new boolean[n + 1];
23  2       int[] parent = new int[n + 1];
24  2       Arrays.fill(parent, -1);
25
26  2       vis[1] = true;
27  3       queue.add(1);
28  1       int level = 1;
29
30          // BFS to find the shortest path
31  4       while (!queue.isEmpty()) {
32  1           int size = queue.size();
33  6           while (size-- > 0) {
34  2               int u = queue.poll();
35
36                  // If we reach node n, reconstruct the path
37  3               if (u == n) {
38  1                   StringBuilder result = new StringBuilder();
39  4                   result.append(level).append("\n");
40
41  1                   List<Integer> path = new ArrayList<>();
42  4                   while (u != -1) {
43  2                       path.add(u);
44                          u = parent[u];
45                      }
46
```

Figure 2: Code Structure in HTML Report in PIT(java)

Mutations

8	1. removed call to java/util/ArrayList::<init> → KILLED
9	1. changed conditional boundary → KILLED 2. Substituted 0 with 1 → KILLED 3. removed conditional - replaced comparison check with false → KILLED 4. negated conditional → KILLED 5. removed conditional - replaced comparison check with true → TIMED_OUT
10	1. removed call to java/util/ArrayList::<init> → KILLED 2. removed call to java/util/List::add → KILLED
15	1. Substituted 0 with 1 → SURVIVED Covering tests 2. Substituted 1 with 0 → SURVIVED Covering tests
16	1. removed call to java/lang/Integer::valueOf → KILLED 2. removed call to java/util/List::add → SURVIVED Covering tests 3. removed call to java/util/List::get → KILLED
17	1. removed call to java/lang/Integer::valueOf → KILLED 2. removed call to java/util/List::get → KILLED 3. removed call to java/util/List::add → SURVIVED Covering tests
21	1. removed call to java/util/LinkedList::<init> → KILLED
22	1. Substituted 1 with 0 → SURVIVED Covering tests 2. Replaced integer addition with subtraction → KILLED
23	1. Substituted 1 with 0 → SURVIVED Covering tests 2. Replaced integer addition with subtraction → KILLED
24	1. removed call to java/util/Arrays::fill → SURVIVED Covering tests 2. Substituted -1 with 0 → SURVIVED Covering tests
26	1. Substituted 1 with 0 → SURVIVED Covering tests 2. Substituted 1 with 0 → SURVIVED Covering tests
27	1. Substituted 1 with 0 → SURVIVED Covering tests 2. removed call to java/lang/Integer::valueOf → KILLED 3. removed call to java/util/Queue::add → SURVIVED Covering tests
28	1. Substituted 1 with 0 → SURVIVED Covering tests
31	1. removed call to java/util/Queue::isEmpty → TIMED_OUT 2. removed conditional - replaced equality check with false → SURVIVED Covering tests 3. negated conditional → SURVIVED Covering tests 4. removed conditional - replaced equality check with true → TIMED_OUT
32	1. removed call to java/util/Queue::size → TIMED_OUT
33	1. negated conditional → TIMED_OUT 2. Changed increment from -1 to 1 → KILLED 3. removed conditional - replaced comparison check with false → TIMED_OUT 4. changed conditional boundary → KILLED 5. removed conditional - replaced comparison check with true → KILLED 6. Removed increment -1 → KILLED
34	1. removed call to java/util/Queue::poll → KILLED 2. removed call to java/lang/Integer::intValue → SURVIVED Covering tests
37	1. removed conditional - replaced equality check with false → SURVIVED Covering tests 2. negated conditional → KILLED 3. removed conditional - replaced equality check with true → KILLED
38	1. removed call to java/lang/StringBuilder::<init> → NO_COVERAGE
39	1. replaced call to java/lang/StringBuilder::append with receiver → NO_COVERAGE 2. removed call to java/lang/StringBuilder::append → NO_COVERAGE 3. replaced call to java/lang/StringBuilder::append with receiver → NO_COVERAGE 4. removed call to java/lang/StringBuilder::append → NO_COVERAGE
41	1. removed call to java/util/ArrayList::<init> → NO_COVERAGE
	1. removed conditional - replaced equality check with false → NO_COVERAGE 2. Substituted -1 with 0 → NO_COVERAGE

Figure 3: Mutation Operators in our Code in PIT (Java)

Active mutators

- CONDITIONALS_BOUNDARY
- CONSTRUCTOR_CALLS
- EMPTY_RETURNS
- EXPERIMENTAL_ARGUMENT_PROPAGATION
- EXPERIMENTAL_BIG_DECIMAL
- EXPERIMENTAL_BIG_INTEGER
- EXPERIMENTAL_MEMBER_VARIABLE
- EXPERIMENTAL_NAKED_RECEIVER
- EXPERIMENTAL_REMOVE_SWITCH_MUTATOR_[0-99]
- EXPERIMENTAL_SWITCH
- FALSE_RETURNS
- INCREMENTS
- INLINE_CONSTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NON_VOID_METHOD_CALLS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- REMOVE_CONDITIONALS_EQUAL_ELSE
- REMOVE_CONDITIONALS_EQUAL_IF
- REMOVE_CONDITIONALS_ORDER_ELSE
- REMOVE_CONDITIONALS_ORDER_IF
- REMOVE_INCREMENTS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Figure 4: Active Mutation Operators in our Code

Pit Test Coverage Report

Package Summary

org.example

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
6	89%	115/129	82%	306/372	87%	306/350

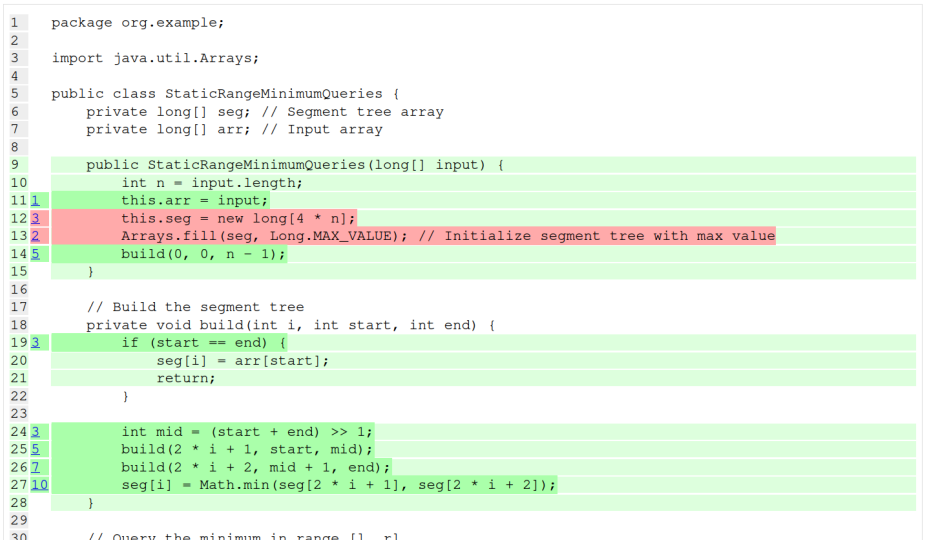
Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BuildingTeams.java	100%	30/30	97%	69/71	97%	69/71
CoinChange.java	89%	8/9	94%	29/31	94%	29/31
DiceCombinations.java	88%	7/8	83%	25/30	83%	25/30
GridPaths.java	90%	9/10	90%	82/91	90%	82/91
MessageRoute.java	74%	31/42	42%	33/78	59%	33/56
RoundTrip.java	100%	30/30	96%	68/71	96%	68/71

Report generated by PIT 1.17.1

Figure 5: Overall Mutation Testing Report generated by PIT (Java)

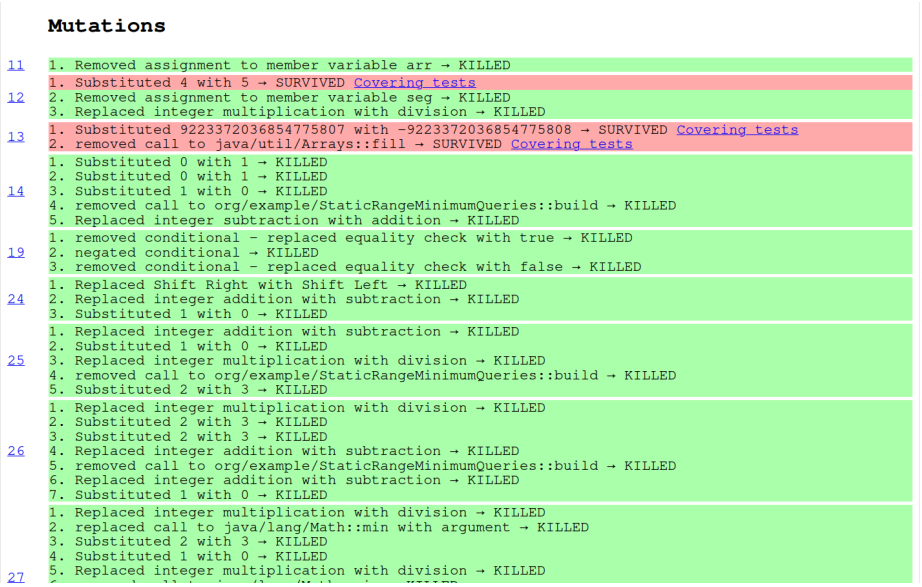
StaticRangeMinimumQueries.java



```
1   package org.example;
2
3   import java.util.Arrays;
4
5   public class StaticRangeMinimumQueries {
6       private long[] seg; // Segment tree array
7       private long[] arr; // Input array
8
9       public StaticRangeMinimumQueries(long[] input) {
10          int n = input.length;
11 1        this.arr = input;
12 3        this.seg = new long[4 * n];
13 2        Arrays.fill(seg, Long.MAX_VALUE); // Initialize segment tree with max value
14 5        build(0, 0, n - 1);
15      }
16
17      // Build the segment tree
18      private void build(int i, int start, int end) {
19 3          if (start == end) {
20              seg[i] = arr[start];
21              return;
22          }
23
24 3        int mid = (start + end) >> 1;
25 5        build(2 * i + 1, start, mid);
26 7        build(2 * i + 2, mid + 1, end);
27 10       seg[i] = Math.min(seg[2 * i + 1], seg[2 * i + 2]);
28      }
29
30      // Query the minimum in range [l, r]
```

Figure 6: Code Structure in HTML Report in PIT(java) integration

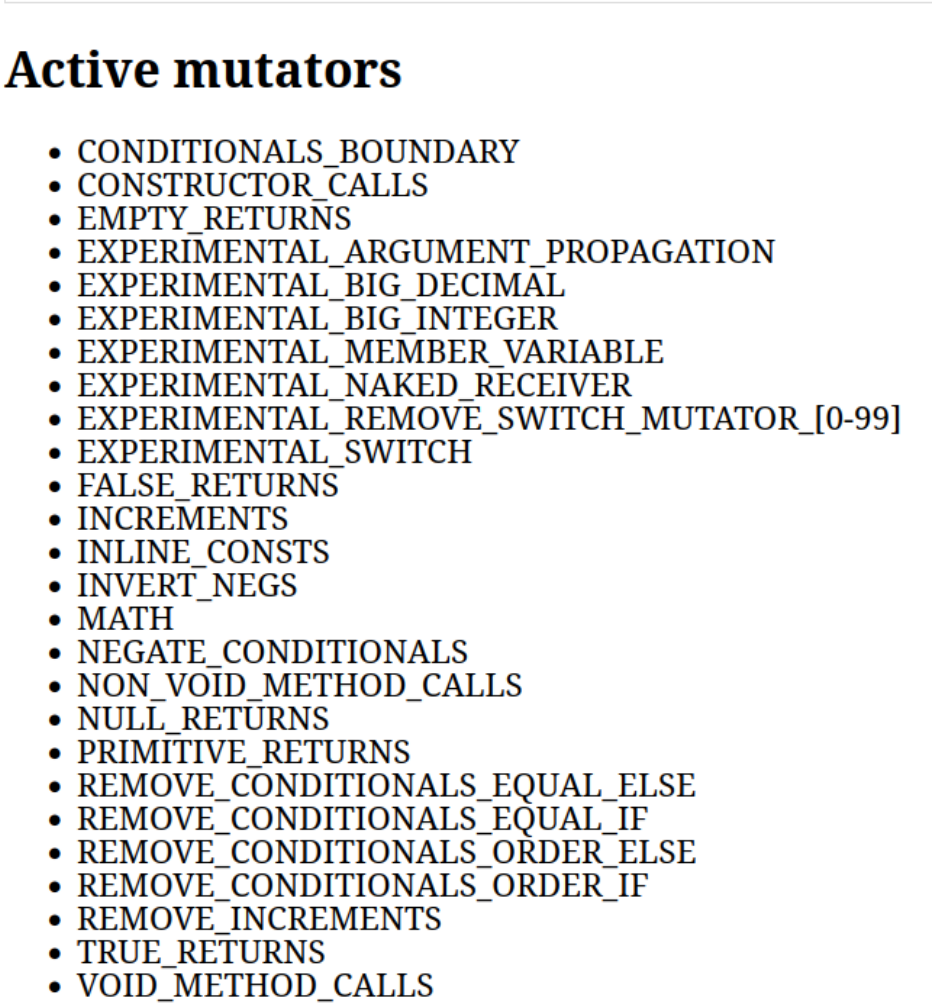
2.4.1 Mutation Operators Generated by PIT for Integration testing



Mutations

11	1. Removed assignment to member variable arr → KILLED
12	1. Substituted 4 with 5 → SURVIVED Covering tests 2. Removed assignment to member variable seq → KILLED 3. Replaced integer multiplication with division → KILLED
13	1. Substituted 9223372036854775807 with -9223372036854775808 → SURVIVED Covering tests 2. removed call to java/util/Arrays::fill → SURVIVED Covering tests
14	1. Substituted 0 with 1 → KILLED 2. Substituted 0 with 1 → KILLED 3. Substituted 1 with 0 → KILLED 4. removed call to org/example/StaticRangeMinimumQueries::build → KILLED 5. Replaced integer subtraction with addition → KILLED
19	1. removed conditional - replaced equality check with true → KILLED 2. negated conditional → KILLED 3. removed conditional - replaced equality check with false → KILLED
24	1. Replaced Shift Right with Shift Left → KILLED 2. Replaced integer addition with subtraction → KILLED 3. Substituted 1 with 0 → KILLED
25	1. Replaced integer addition with subtraction → KILLED 2. Substituted 1 with 0 → KILLED 3. Replaced integer multiplication with division → KILLED 4. removed call to org/example/StaticRangeMinimumQueries::build → KILLED 5. Substituted 2 with 3 → KILLED
26	1. Replaced integer multiplication with division → KILLED 2. Substituted 2 with 3 → KILLED 3. Substituted 2 with 3 → KILLED 4. Replaced integer addition with subtraction → KILLED 5. removed call to org/example/StaticRangeMinimumQueries::build → KILLED 6. Replaced integer addition with subtraction → KILLED 7. Substituted 1 with 0 → KILLED
27	1. Replaced integer multiplication with division → KILLED 2. replaced call to java/lang/Math::min with argument → KILLED 3. Substituted 2 with 3 → KILLED 4. Substituted 1 with 0 → KILLED 5. Replaced integer multiplication with division → KILLED

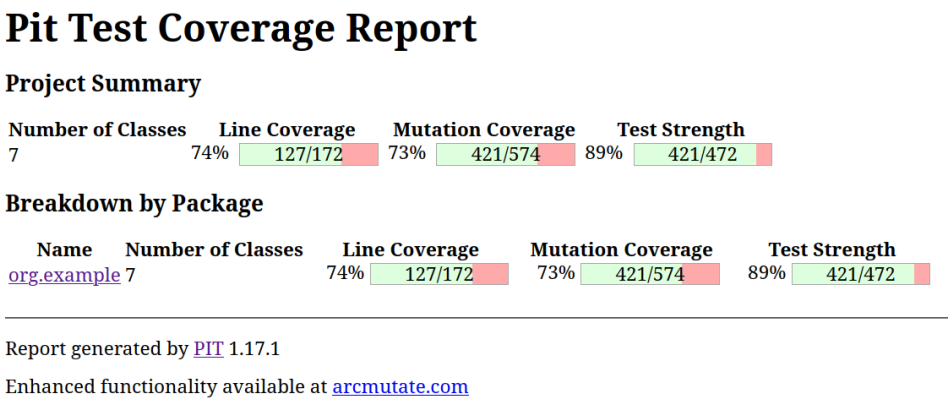
Figure 7: Mutation Operators in our Code in PIT (Java) integration



Active mutators

- CONDITIONALS_BOUNDARY
- CONSTRUCTOR_CALLS
- EMPTY_RETURNS
- EXPERIMENTAL_ARGUMENT_PROPAGATION
- EXPERIMENTAL_BIG_DECIMAL
- EXPERIMENTAL_BIG_INTEGER
- EXPERIMENTAL_MEMBER_VARIABLE
- EXPERIMENTAL_NAKED_RECEIVER
- EXPERIMENTAL_REMOVE_SWITCH_MUTATOR_[0-99]
- EXPERIMENTAL_SWITCH
- FALSE_RETURNS
- INCREMENTS
- INLINE_CONSTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NON_VOID_METHOD_CALLS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- REMOVE_CONDITIONALS_EQUAL_ELSE
- REMOVE_CONDITIONALS_EQUAL_IF
- REMOVE_CONDITIONALS_ORDER_ELSE
- REMOVE_CONDITIONALS_ORDER_IF
- REMOVE_INCREMENTS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Figure 8: Active Mutation Operators in our Code integration



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
7	74% 127/172	73% 421/574	89% 421/472

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.example	7	74% 127/172	73% 421/574	89% 421/472

Report generated by PIT 1.17.1

Enhanced functionality available at arcmutate.com

Figure 9: Overall Mutation Testing Report generated by PIT (Java) integration

References

[1] What is Mutation Testing? Definition from TechTarget.

[2] Mutation Testing on Wikipedia.

[3] How to Test Your Unit Tests.

[4] Documentation of PIT (Java tool to do Mutation Testing).

[5] CSES.