Mutation Testing with PIT

Team:

Akhil Puppala (IMT2021021)
Gowtham Reddy (IMT2021081)

Abstract:

This report details the implementation of mutation testing using PIT (Pitest) on a Java-based project. Mutation testing, a fault-based testing method, is designed to assess the effectiveness of test cases. The project involves utilizing mutation operators, creating mutants, and evaluating whether these mutants survive or are eliminated to enhance the robustness of the test suite. The findings highlight the practical advantages of mutation testing in improving software quality.

Introduction:

Mutation testing is a software testing approach that modifies a program's source code by introducing small changes, known as mutations, to create multiple versions called mutants. The goal is to assess the effectiveness of the test suite in detecting and eliminating these mutants, thereby ensuring software robustness.

PIT (Pitest) is a mutation testing tool for Java that generates mutants and evaluates the adequacy of test cases. This project leverages PIT to analyze and enhance the test suite of a Java application.

Objectives:

The main objectives of this project are:

- To implement mutation testing using PIT.
- To evaluate the strength and coverage of the test suite.
- To analyze the impact of different mutation operators on the test results.

Methodology:

1) **Tools used:**
    - Programming Language: Java
    - Build Tool: Maven
    - Mutation Testing Tool: PIT
    - Testing Framework: JUnit

2) Project Setup:

The mutation testing setup included the following configurations in the `pom.xml` file:

- Adding dependencies for PIT, JUnit, and necessary libraries.
- Defining the target classes and corresponding test classes.
- Configuring mutation operators, including experimental options like `EXPERIMENTAL_ARGUMENT_PR`.

```xml
    <build>
        <plugins>
            <!-- PIT Mutation Testing Plugin -->
            <plugin>
                <groupId>org.pitest</groupId>
                <artifactId>pitest-maven</artifactId>
                <version>1.9.11</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>mutationCoverage</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <targetClasses>
                        <param>org.example.*</param>
                    </targetClasses>
                    <targetTests>
                        <param>org.example.*</param>
                    </targetTests>
                    <outputFormats>
                        <param>HTML</param>
                    </outputFormats>
                    <mutators>
                        <param>ALL</param>
                    </mutators>
                    <features>
                        <feature>+EXPORT</feature>
                    </features>
                    <threads>4</threads>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Mutation Operators

The following mutation operators were utilized in this project, categorized as either unit or integration based on their scope and impact:

Unit Mutators

These operators focus on testing individual code units, such as methods, to verify their handling of edge cases and unexpected inputs:

- CONDITIONALS BOUNDARY: Alters conditional boundary values.
- INCREMENTS: Modifies increment operations, such as `i++`.
- INLINE CONSTS: Substitutes constants with inlined values.
- INVERT NEGS: Reverses the sign of numerical values.
- MATH: Replaces mathematical operators with alternative ones.
- NEGATE CONDITIONALS: Reverses the logic of conditions in `if` statements.
- REMOVE INCREMENTS: Eliminates increment operations.

Integration Mutators

These mutators are designed to evaluate the interaction between components and the overall program logic:

- EMPTY RETURNS: Replaces method return values with an empty return.
- EXPERIMENTAL ARGUMENT PROPAGATION: Propagates arguments to different usages.
- FALSE RETURNS: Forces methods to return `false`.
- NULL RETURNS: Forces methods to return `null`.
- PRIMITIVE RETURNS: Forces methods to return default primitive values.
- TRUE RETURNS: Forces methods to return `true`.
- VOID METHOD CALLS: Eliminates calls to methods that return `void`.
- NON VOID METHOD CALLS: Removes calls to methods that return values.
- REMOVE CONDITIONALS EQUAL ELSE: Removes the `else` block when conditional branches are equal.
- REMOVE CONDITIONALS EQUAL IF: Removes the `if` block when conditional branches are equal.
- REMOVE CONDITIONALS ORDER ELSE: Removes the `else` block when conditionals follow a specific order.

- REMOVE CONDITIONALS ORDER IF: Removes the `if` block when conditionals follow a specific order.

This categorization clarifies the mutators' scope, aiding in understanding their role in effectively testing the software.

Test Cases:

The test cases were designed to cover all possible code paths and edge cases:

```
@Test
public void testSingleWord() {
    countanagrams solution = new countanagrams();
    assertEquals( expected: 6, solution.countAnagrams( s: "abc")); // Single word with distinct characters
}
```

Implementation Details:

The implementation consisted of the following steps:

- Breaking down the source code into smaller, modular functions to enhance readability and maintainability.
- Setting up the `pom.xml` file to integrate PIT.
- Executing PIT to generate mutants and analyzing the outcomes.
- Improving the test suite by addressing the surviving mutants.

Results:

This section provides a summary of the results from the mutation testing process. The accompanying figures showcase the source code, mutation coverage, the impact of active mutators, and the distribution of survived mutants for two of the files.

1)

Sol5.java

```
1  package org.example;
2
3  class Sol5 {
4      public int maxProfit(int[] prices) {
5          int n= prices.length;
6  4       int [][][] dp= new int[n+1][2][3];
7
8  5       for(int i=0; i<n; i++){// base case for i<n put 0's, only for understanding  no need to write bcz array is by default zero
9  6           for(int buy=0; buy<=1; buy++){
10 2               dp[i][buy][0]= 0;
11             }
12         }
13 6       for(int buy=0; buy<=1; buy++){ //base case for maxTransaction<0 put 0's only for understanding  no need to write bcz array is by default zero
14 6           for(int maxTransaction=0; maxTransaction<=2; maxTransaction++){
15 1               dp[n][buy][maxTransaction]= 0;
16             }
17         }
18
19 6       for(int i= n-1; i>=0; i--){
20 1           int profit=0;
21 6           for(int buy=0; buy<=1; buy++){
22 6               for(int maxTransaction=1; maxTransaction<=2; maxTransaction++){
23 4                   if(buy==1){  // buying
24 5                       int take= -prices[i] + dp[i+1][0][maxTransaction]; // buying ith on day and here it -prices[i] becxause we are buying it
25 5                       int notTake= 0 + dp[i+1][1][maxTransaction]; // not buying ith  onday
26 2                       profit= Math.max(take, notTake);
27                         dp[i][buy][maxTransaction]= profit;
28
29 3                   }else if(buy==0){  // selling
30
31 6                       int take= prices[i] + dp[i+1][1][maxTransaction-1];// selling on ith day and here it +prices[i] because we are selling it
32 5                       int notTake= 0 + dp[i+1][0][maxTransaction]; // not selling on ith day
33 2                       profit= Math.max(take, notTake);
34                         dp[i][buy][maxTransaction]= profit;
35                     }
36                 }
37             }
38         }
39 4       return dp[0][1][2];
40     }
41 }
```

Mutations

```
6  1. Substituted 1 with 0 → KILLED
   2. Substituted 2 with 3 → SURVIVED
   3. Substituted 3 with 4 → SURVIVED
   4. Replaced integer addition with subtraction → KILLED
8  1. changed conditional boundary → SURVIVED
   2. Substituted 0 with 1 → SURVIVED
   3. negated conditional → KILLED
```

```
   6. removed conditional - replaced comparison check with true → KILLED
14 1. changed conditional boundary → SURVIVED
   2. Substituted 0 with 1 → SURVIVED
   3. Substituted 2 with 3 → KILLED
   4. negated conditional → SURVIVED
   5. removed conditional - replaced comparison check with false → SURVIVED
   6. removed conditional - replaced comparison check with true → KILLED
15 1. Substituted 0 with 1 → KILLED
19 1. changed conditional boundary → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Replaced integer subtraction with addition → KILLED
   4. negated conditional → KILLED
   5. removed conditional - replaced comparison check with false → KILLED
   6. removed conditional - replaced comparison check with true → KILLED
20 1. Substituted 0 with 1 → SURVIVED
21 1. changed conditional boundary → KILLED
   2. Substituted 0 with 1 → KILLED
   3. Substituted 1 with 0 → KILLED
   4. negated conditional → KILLED
   5. removed conditional - replaced comparison check with false → KILLED
   6. removed conditional - replaced comparison check with true → TIMED_OUT
22 1. changed conditional boundary → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 2 with 3 → KILLED
   4. negated conditional → KILLED
   5. removed conditional - replaced comparison check with false → KILLED
   6. removed conditional - replaced comparison check with true → KILLED
23 1. Substituted 1 with 0 → KILLED
   2. negated conditional → KILLED
   3. removed conditional - replaced equality check with false → KILLED
   4. removed conditional - replaced equality check with true → KILLED
24 1. Substituted 1 with 0 → SURVIVED
   2. Substituted 0 with 1 → KILLED
   3. removed negation → KILLED
   4. Replaced integer addition with subtraction → KILLED
   5. Replaced integer addition with subtraction → KILLED
25 1. Substituted 0 with 1 → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 1 with 0 → KILLED
   4. Replaced integer addition with subtraction → KILLED
   5. Replaced integer addition with subtraction → KILLED
26 1. removed call to java/lang/Math::max → KILLED
   2. replaced call to java/lang/Math::max with argument → KILLED
29 1. negated conditional → KILLED
   2. removed conditional - replaced equality check with false → KILLED
   3. removed conditional - replaced equality check with true → SURVIVED
31 1. Substituted 1 with 0 → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 1 with 0 → KILLED
   4. Replaced integer addition with subtraction → KILLED
   5. Replaced integer subtraction with addition → KILLED
   6. Replaced integer addition with subtraction → KILLED
32 1. Substituted 0 with 1 → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 0 with 1 → KILLED
   4. Replaced integer addition with subtraction → KILLED
   5. Replaced integer addition with subtraction → KILLED
33 1. removed call to java/lang/Math::max → KILLED
   2. replaced call to java/lang/Math::max with argument → KILLED
39 1. Substituted 0 with 1 → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 2 with 3 → KILLED
   4. replaced int return with 0 for org/example/Sol5::maxProfit → KILLED
```

Summary of Killed and Survived Mutants for the maxProfit:

Method

Killed Mutants

The following types of mutants were effectively killed, showcasing the robustness of the test suite:

1. Control Flow and Conditional Logic:
    - Negating conditionals (e.g., replacing `buy == 1` with `buy != 1`).
    - Changing conditional boundaries (e.g., replacing `maxTransaction <= 2` with `maxTransaction < 2`).
    - Removing conditionals and replacing equality checks with constants (`true` or `false`).
2. Arithmetic and Logical Operations:
    - Substituting values (e.g., replacing `1` with `0` or `2` with `3` in logic).
    - Replacing integer addition (`+`) with subtraction (`-`) or vice versa.
    - Replacing integer return values with constants (`0`, `1`, etc.).
3. Function Calls:
    - Removing or modifying calls to utility methods, such as `Math.max`.
    - Replacing `Math.max(a, b)` with one of the arguments (`a` or `b`).

Survived Mutants

The surviving mutants indicate areas where the test suite did not fully cover potential edge cases or variations:

1. Boundary and Initialization Issues:
    - Subtle changes in initialization values (e.g., replacing `2` with `3` in conditions).
    - Changes to loop or conditional boundaries not triggering test failures (e.g., `maxTransaction` boundary conditions).
2. Conditionals:
    - Some mutations to conditionals, such as replacing `buy == 1` with `buy == 0`, survived due to insufficient coverage of these specific cases.

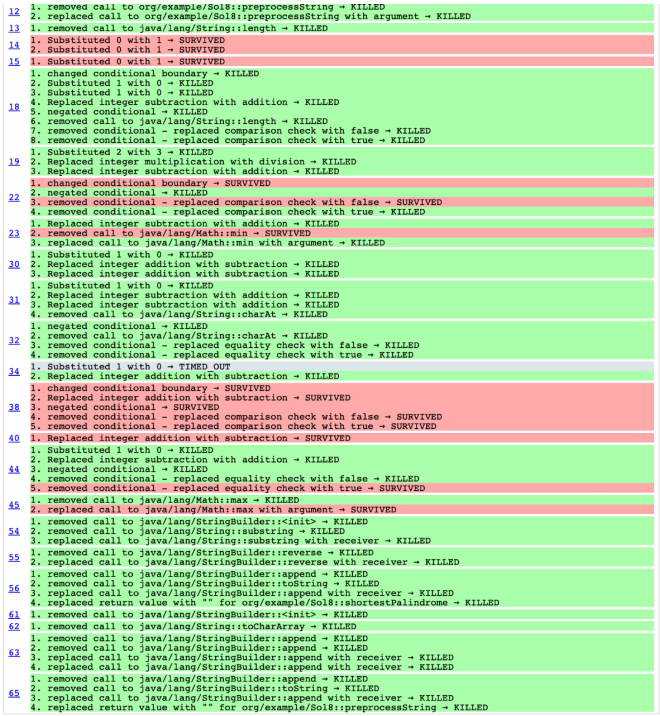
2)

Sol8.java

```
1  package org.example;
2
3  class Sol8 {
4
5      public String shortestPalindrome(String s) {
6          // Return early if the string is null or empty
7  7       if (s == null || s.length() == 0) {
8  1           return s;
9          }
10
11         // Preprocess the string to handle palindromes uniformly
12 2       String modifiedString = preprocessString(s);
13 1       int[] palindromeRadiusArray = new int[modifiedString.length()];
14 2       int center = 0, rightBoundary = 0;
15 1       int maxPalindromeLength = 0;
16
17         // Iterate through each character in the modified string
18 8       for (int i = 1; i < modifiedString.length() - 1; i++) {
19 1           int mirrorIndex = 2 * center - i;
20
21             // Use previously computed values to avoid redundant calculations
22 4           if (rightBoundary > i) {
23 3               palindromeRadiusArray[i] = Math.min(
24                                 rightBoundary - i,
25                                 palindromeRadiusArray[mirrorIndex]
26                     );
27             }
28
29             // Expand around the current center while characters match
30 3           while (
31 4                         modifiedString.charAt(i + 1 + palindromeRadiusArray[i]) ==
32 4                                 modifiedString.charAt(i - 1 - palindromeRadiusArray[i])
33             ) {
34 2               palindromeRadiusArray[i]++;
35             }
36
37             // Update the center and right boundary if the palindrome extends beyond the current boundary
38 8           if (i + palindromeRadiusArray[i] > rightBoundary) {
39                 center = i;
40 1               rightBoundary = i + palindromeRadiusArray[i];
41             }
42
43             // Update the maximum length of palindrome starting at the beginning
44 5           if (i - palindromeRadiusArray[i] == 1) {
45 2               maxPalindromeLength = Math.max(
46                                 maxPalindromeLength,
47                                 palindromeRadiusArray[i]
48                     );
49             }
50         }
51
52         // Construct the shortest palindrome by reversing the suffix and prepending it to the original string
53         StringBuilder suffix = new StringBuilder(
54 3               s.substring(maxPalindromeLength)
55 2           ).reverse();
56 4         return suffix.append(s).toString();
57     }
58
59     private String preprocessString(String s) {
60         // Add boundaries and separators to handle palindromes uniformly
61 1       StringBuilder sb = new StringBuilder("^");
62 1       for (char c : s.toCharArray()) {
63 4           sb.append("#").append(c);
64           }
65 4        return sb.append("#$").toString();
66     }
67 }
```

Mutations

```
   1. negated conditional → KILLED
   2. negated conditional → KILLED
   3. removed call to java/lang/String::length → KILLED
7  4. removed conditional - replaced equality check with false → KILLED
   5. removed conditional - replaced equality check with true → SURVIVED
   6. removed conditional - replaced equality check with true → SURVIVED
   7. removed conditional - replaced equality check with true → KILLED
8  1. replaced return value with "" for org/example/Sol8::shortestPalindrome → SURVIVED
12 1. removed call to org/example/Sol8::preprocessString → KILLED
   2. replaced call to org/example/Sol8::preprocessString with argument → KILLED
13 1. removed call to java/lang/String::length → KILLED
14 1. Substituted 0 with 1 → SURVIVED
   2. Substituted 0 with 1 → SURVIVED
15 1. Substituted 0 with 1 → SURVIVED
   1. changed conditional boundary → KILLED
   2. Substituted 1 with 0 → KILLED
   3. Substituted 1 with 0 → KILLED
   4. Replaced integer subtraction with addition → KILLED
18 5. negated conditional → KILLED
   6. removed call to java/lang/String::length → KILLED
   7. removed conditional - replaced comparison check with false → KILLED
   8. removed conditional - replaced comparison check with true → KILLED
   1. Substituted 2 with 3 → KILLED
19 2. Replaced integer multiplication with division → KILLED
   3. Replaced integer subtraction with addition → KILLED
   1. changed conditional boundary → SURVIVED
   2. negated conditional → KILLED
22 3. removed conditional - replaced comparison check with false → SURVIVED
   4. removed conditional - replaced comparison check with true → KILLED
   1. Replaced integer subtraction with addition → KILLED
23 2. removed call to java/lang/Math::min → SURVIVED
   3. replaced call to java/lang/Math::min with argument → KILLED
   1. Substituted 1 with 0 → KILLED
30 2. Replaced integer addition with subtraction → KILLED
   3. Replaced integer addition with subtraction → KILLED
   1. Substituted 1 with 0 → KILLED
31 2. Replaced integer subtraction with addition → KILLED
   3. Replaced integer subtraction with addition → KILLED
   4. removed call to java/lang/String::charAt → KILLED
   1. negated conditional → KILLED
32 2. removed call to java/lang/String::charAt → KILLED
   3. removed conditional - replaced equality check with false → KILLED
   4. removed conditional - replaced equality check with true → KILLED
34 1. Substituted 1 with 0 → TIMED_OUT
   2. Replaced integer addition with subtraction → KILLED
   1. changed conditional boundary → SURVIVED
   2. Replaced integer addition with subtraction → SURVIVED
38 3. negated conditional → SURVIVED
   4. removed conditional - replaced comparison check with false → SURVIVED
   5. removed conditional - replaced comparison check with true → SURVIVED
40 1. Replaced integer addition with subtraction → SURVIVED
   1. Substituted 1 with 0 → KILLED
   2. Replaced integer subtraction with addition → KILLED
44 3. negated conditional → KILLED
   4. removed conditional - replaced equality check with false → KILLED
   5. removed conditional - replaced equality check with true → SURVIVED
45 1. removed call to java/lang/Math::max → KILLED
   2. replaced call to java/lang/Math::max with argument → SURVIVED
   1. removed call to java/lang/StringBuilder::<init> → KILLED
54 2. removed call to java/lang/String::substring → KILLED
   3. replaced call to java/lang/String::substring with receiver → KILLED
55 1. removed call to java/lang/StringBuilder::reverse → KILLED
   2. replaced call to java/lang/StringBuilder::reverse with receiver → KILLED
   1. removed call to java/lang/StringBuilder::append → KILLED
```



Summary of Killed and Survived Mutants for the shortestPalindrome Method

---

Killed Mutants

The following mutants were effectively detected and killed, indicating strong coverage by the test suite:

1. Control Flow and Conditional Logic:
    - Negated conditionals (e.g., replacing `s == null` with `s != null`).
    - Removed conditionals and replaced equality checks with constants (`true` or `false`).
    - Changed conditional boundaries (e.g., altering loop boundaries or condition checks).
2. Arithmetic and Logical Operations:

- Replaced integer addition with subtraction and vice versa.
- Substituted integer values (e.g., `0` with `1`, `2` with `3`).
- Replaced integer subtraction with addition.

3. Function Calls:
    - Removed calls to critical methods:
        - `String::length`, `String::charAt`, `String::toCharArray`, and `String::substring`.
        - `Math::max` and `Math::min`.
        - `StringBuilder::<init>`, `StringBuilder::reverse`, and `StringBuilder::append`.
    - Replaced calls to utility methods (e.g., `Math::max` and `Math::min`) with one of their arguments.
4. Object Construction and Initialization:
    - Removed object initialization, such as `StringBuilder::<init>`.
5. Other Modifications:
    - Replaced return values (e.g., `""` or an empty string for `shortestPalindrome`).
    - Replaced calls to helper methods, such as `preprocessString`, with direct arguments.

Survived Mutants

These mutants were not detected by the test suite, revealing gaps in test coverage:

1. Boundary and Initialization:
    - Substituting constants, such as `0` with `1` and `1` with `0`, survived in some conditional checks and loop boundaries.
    - Modifications to boundaries in conditional checks (e.g., `if` or `while`) were not adequately tested.
2. Function Calls:
    - Replaced calls to utility methods, such as `Math::min`, with arguments were not detected in some cases.
3. Return Value Changes:
    - Replacing the return value of `shortestPalindrome` with `""` (empty string) in some instances survived.
4. Control Flow and Logical Operations:
    - Negating conditionals in certain situations did not trigger test failures.

- Removing conditionals and replacing them with constants (`true` or `false`) survived in a few cases.

Pit report:

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
MaxSumAfterPartitioning.java	93%	13/14	93%	28/30	93%	28/30
Sol10.java	100%	24/24	85%	88/104	85%	88/104
Sol3.java	94%	16/17	88%	66/75	90%	66/73
Sol4.java	100%	19/19	85%	28/33	85%	28/33
Sol5.java	100%	24/24	73%	62/85	73%	62/85
Sol6.java	94%	16/17	90%	87/97	96%	87/91
Sol7.java	97%	32/33	81%	50/62	82%	50/61
Sol8.java	100%	28/28	78%	60/77	78%	60/77
Sol9.java	100%	25/25	99%	71/72	99%	71/72
constructGridLayout.java	97%	106/109	74%	179/241	75%	179/239
countPrefix.java	100%	18/18	96%	48/50	96%	48/50
countanagrams.java	100%	20/20	100%	50/50	100%	50/50
eggDrop.java	100%	27/27	87%	72/83	88%	72/82
intervals.java	92%	11/12	98%	55/56	98%	55/56
maxvalue.java	100%	67/67	79%	177/223	79%	177/223
mergeOverlappingIntervals.java	92%	11/12	98%	55/56	98%	55/56
minswaps.java	100%	22/22	74%	55/74	74%	55/74
overlapping.java	100%	26/26	78%	67/86	78%	67/86
rectangle.java	100%	36/36	98%	63/64	98%	63/64
smallestnumber.java	100%	42/42	74%	139/188	83%	139/168

Conclusion:

This project successfully implemented mutation testing using PIT, demonstrating its ability to evaluate and improve the robustness of test cases. The use of both standard and experimental mutation operators provided valuable insights into the strengths and weaknesses of the test suite.

References:

- PIT Documentation: https://pitest.org/
- JUnit Documentation: https://junit.org/junit4/
- Maven Documentation: https://maven.apache.org/