Software Testing Project Report

Karan Raj Pandey(IMT2021014)
Prasad Jore(IMT2021104)

Contents

1. Introduction

Software testing is a critical process used to determine whether a software product meets its intended requirements and to ensure it is free from defects. This involves evaluating software components through manual or automated testing tools to check for various key properties. The primary objective of software testing is to identify errors, discrepancies, or unmet requirements when compared to the expected specifications. Effective software testing is crucial because it allows early detection and correction of bugs or issues before the software product is released.

A thoroughly tested software product guarantees reliability, security, and high performance, resulting in time efficiency, cost savings, and improved customer satisfaction.

Mutation testing is a specialised type of software testing that involves making deliberate modifications (mutations) to the source code to check whether the existing test cases can detect these changes. The purpose of mutation testing is to evaluate the robustness and quality of the test cases, ensuring that they can catch faults introduced by the mutations. A mutation is simply a single, small alteration in the program's code. Each mutant version of the program should differ from the original by only one mutation. The test cases are considered mutation adequate if they achieve a 100% score, meaning they effectively detect all mutations. Research has demonstrated that mutation testing is a powerful method for assessing test case adequacy, though it comes with the drawback of being resource-intensive due to the need to generate and execute numerous mutant versions.

The mutation testing process follows these steps:

1. **Mutant Generation**: Faults are introduced into the source code by creating several mutant versions. Each mutant contains a single fault, and the aim is to cause the mutant to fail, demonstrating the effectiveness of the test cases.
2. **Test Case Execution**: Test cases are run on both the original and mutant versions of the program. Adequate test cases are designed to detect faults in the program.
3. **Comparison**: The outputs of the original and mutant versions are compared.
    - If the outputs differ, the mutant is considered *killed* by the test case, indicating that the test case successfully detected the fault.
    - If the outputs are the same, the mutant *survives*, signalling the need for more refined test cases to catch the fault.

For this project, we developed a suitable codebase and applied mutation testing using the PIT (Program Integration Tester) tool.

1.

2. Codebase Documentation

The class StringLibrary has 37 functions on various aspects of a String in Java. The following
list has a short description of each function along with the inputs and expected outputs.

1. String reverseWords(String s): Returns the sentence with the words reversed in it.
Input: “ hello how are you ”
Expected output: you are how hello

2. String longestPalindrome(String s): Returns the longest palidromic substring in the input string.
Input: “abbaggbasggaggba”
Expected output: “ggagg”

3. int romanToInt(String s): Takes a roman number as string input and returns an integer representing it in decimal.
Input: “CMXCIX”
Expected output: 999

4. int atoi(String s): Takes an integer number string and converts it into an integer.
Input: “ - 25343 ”
Expected output: -25343

5. String longestCommonPrefix(String[] strs): Returns the longest found common pre-fix from an array of strings.
Input: {“pregsg”,“presgsd”,“pregggsss”, “praasg”}
Expected output: “pr”

6. int[] z_function(String s): Takes a string and returns an array representing z-function for that string. z-function, z[i] for a string s at index i, is the maximum length of prefix of s that matches the prefix of s[i:n-1] (s[i:j] denotes the substring of s from indices i to j).
Input: “sdfhgdfdlgadesg”
Expected output: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0}

7. int[] prefix_function(String s): Takes a string and returns an array representing prefix-function for that string. prefix-function, pi[i] for a string s at index i, is the maximum length of prefix of s that matches the suffix of s[0:i].
Input: “sdfhgdfdlgadesg”

Expected output: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0}

8. int firstOccurrence(String haystack, String needle): Returns the first occurrence of the string needle in the string haystack. Returns -1 if not found.
Input: “heyhowarehoware”, “how”
Expected output: 3

9. int editDistance(String word1, String word2): Returns the edit distance between word1 and word2.
Input: “abcasdgsd”, “ahdsfbbcsdsf”
Expected output: 8

10. int longestCommonSubsequence(String text1, String text2): Returns the length of the longest common subsequence in the two input words.
Input: “abcasdgsd”, “ahdsfbbcsdsf”
Expected output: 6

11. int palindromePartitioning(String str): Returns the minimum number of cuts needed to partition the string str into palindromic substrings.
Input: “abbcccdgab”
Expected output: 6

12. boolean checkPangram(String str): This function checks if the given string is a pan-gram or not. A pangram is a string which has each alphabet from A to Z at least once.
Input: “AbcdefghijkLmnopqrstUVwxyz”
Expected output: true

13. String toLowercase(String str): Returns the string formed by converting every letter in the input string to lowercase.
Input: “abUkdfASZ”
Expected output: “abukdfasz”

14. String toUppercase(String str): Returns the string formed by converting every letter in the input string to uppercase.
Input: “abUkdfASz”
Expected output: “ABUKDFASZ”

15. boolean areAnagrams(String str1, String str2): Returns true if the two input strings are anagrams of each other and false otherwise. Anagrams are strings that have the same number of occurrences of each character present in them.
Input: “ggadabbsa”, “abbadgags”
Expected output: true

16. int countOccurrences(String str1, String str2): Returns the number of times str2 appeared as a substring in str1.
Input: “abcdabcabcabcabcd”, “abc”

Expected output: 5

17. int countWords(String str): Returns the number of words in a string. A word here is a maximal consecutive substring of characters, not including white space in it.
Input: “gdsf gsdf ffsd sdfds hgdsds gsdf gsdg asdg”
Expected output: 8

18. int countCharacters(String str): Returns the nubmer of non-space characters in a string.
Input: “gdsf gsdf ffsd sdfds hgdsds gsdf gsdg asdg”
Expected output: 35

19. String minLengthWord(String str): Returns the minimum length word in a string. If mutliple occurrences are there, it returns the earliest one.
Input: “jfh yi a sdf hgd”
Expected output: “a”

20. String maxLengthWord(String str)): Returns the maximum length word in a string. If mutliple occurrences are there, it returns the earliest one.
Input: “jfh yi a sdf hgd”
Expected output: “jfh”

21. boolean issubsequence(String s1, String s2): Returns true if the string s1 is a sub-sequence of string s2 and false otherwise.
Input: “ABC”, “abbadgagsc”
Expected output: true

22. String reverseString(String str): Returns the string formed by reversing the input string.
Input: “Udjsfhdc..”
Expected output: “..cdhfsjdU"

23. boolean isCornerPresent(String str, String corner): Find if a string str starts and ends with another given string corner.
Input: "hellorajhello"
Expected output: hello

24. boolean isPresent(String str1, String str2): Given two strings str1 and str2, check if all characters of str2 are present in str1.
Input: "abcdef"
Expected output: acf

25. String mirror(String str, int n): Mirror the characters in str from the N th position up to the length of the string in alphabetical order.
Input: "hellogoodmorning", 4
Expected output: heloltllwnlimrmt

26. boolean canMakeStr2(String str1, String str2): Given two strings str1 and str2, check if str2 can be formed from str1.
Input: "abcdef", "ade"
Expected output: true

27. String getInfix(String exp): Convert a given postfix expression exp to its infix equivalent.
Input: "abc++"
Expected output: (a+(b+c))

28. String postToPre(String exp): Convert a given postfix expression exp to its prefix equivalent.
Input: "AB+CD-*"
Expected output: *+AB-CD

29. String preToPost(String exp): Convert a given prefix expression exp to its postfix equivalent.
Input: "*+AB-CD"
Expected output: AB+CD-*

30. String preToIn(String exp): Convert a given prefix expression exp to its infix equivalent.
Input: "*+AB-CD"
Expected output: ((A+B)*(C-D))

31. int minOps(String str1, String str2): Find the minimum number of operations required to transform str1 to str2.
Input: "ABD", "BAD"
Expected output: 1

32. int countTransformation(String str1, String str2): Find the number of ways of transforming one string str1 to another str2 by removing 0 or more characters.
Input: "abcccdf", "abccdf"
Expected output: 3

33. String transformString(String str): Transform a given string str by deleting vowels, changing the case and inserting special symbols.
Input: "hello123!!!;;;;"
Expected output: #H#L#L123!!!;;;;
34. boolean checkTransform(String str1, String str2): Check if a string str1 can be converted to another string str2.
Input: "ABcd", "BCD"
Expected output: false

35. String printRoman(int number): Convert the decimal value number to its Roman numeral equivalent.
Input: 521

Expected output: DXXI

36. double binaryToDecimal(String binary): Convert the binary number represented as String binary to its Decimal equivalent.
Input: "110.101"
Expected output: 6.625

37. String decimalToBinary(double num, int k_prec): Convert the decimal number num to its Decimal equivalent upto k precision.
Input: 6.986, 8
Expected output: 110.11111100

3. Mutation Testing using PIT

There are several mutation testing tools for Java, such as µJava, Jester, and Jumble, but they are not widely adopted. These tools are often slow, difficult to use, and primarily developed to meet academic research needs rather than practical applications for development teams. In contrast, PIT was selected for its speed, user-friendliness, and ongoing development and support.

PIT is a cutting-edge mutation testing tool that delivers gold-standard test coverage for Java and JVM-based applications. It is fast, scalable, and seamlessly integrates with modern testing and build systems. PIT works by running unit tests against automatically modified versions of the application's code. If changes in the application code don't cause the unit tests to fail, it may indicate weaknesses in the test suite.

PIT produces reports that are easy to understand, combining both line coverage and mutation coverage metrics.

The tool uses a customizable set of mutation operators (or mutators) applied to the compiled bytecode rather than the source files. This method is generally faster and simpler to integrate into a build process. PIT defines various mutation operations, such as removing method calls, flipping logical conditions, modifying return values, and more. Applying these mutation operators generates numerous mutants—Java classes altered by one mutation—which are expected to behave differently from their original counterparts.

While mutation testing can be computationally intensive and time-consuming, especially for larger codebases or complex test suites, it provides valuable insights into test suite effectiveness.

Equivalent Mutants

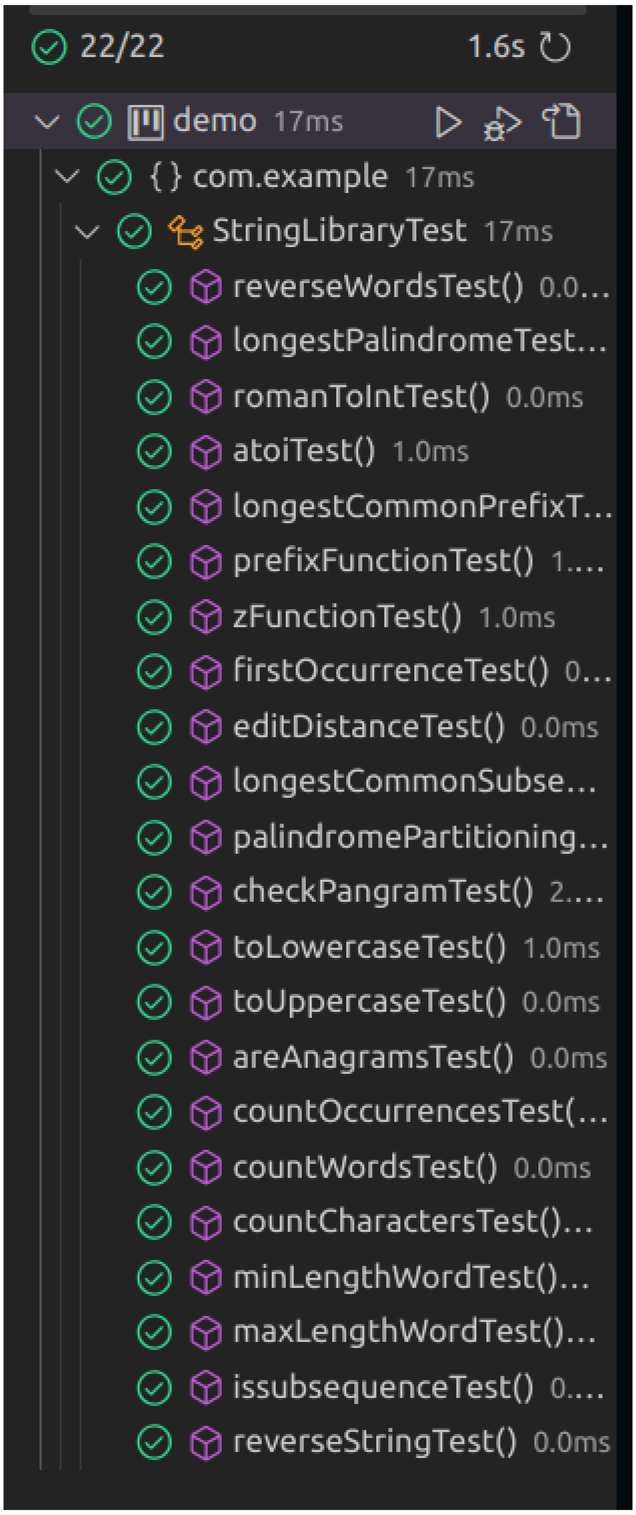
Some mutants may not exhibit any behavioural differences from the original code. These are known as *equivalent mutants*, and they cannot be detected or "killed" by any test case.

Common Errors

- **Timeout Errors**: These can occur if a mutation leads to an infinite loop, such as removing the increment statement from a `for` loop.
- **Memory Errors**: These may arise if a mutation increases memory consumption or from the additional overhead of running multiple tests with mutations.
- **Run Errors**: These indicate that an issue occurred while testing a mutation. Certain non-viable mutations can currently result in run errors.

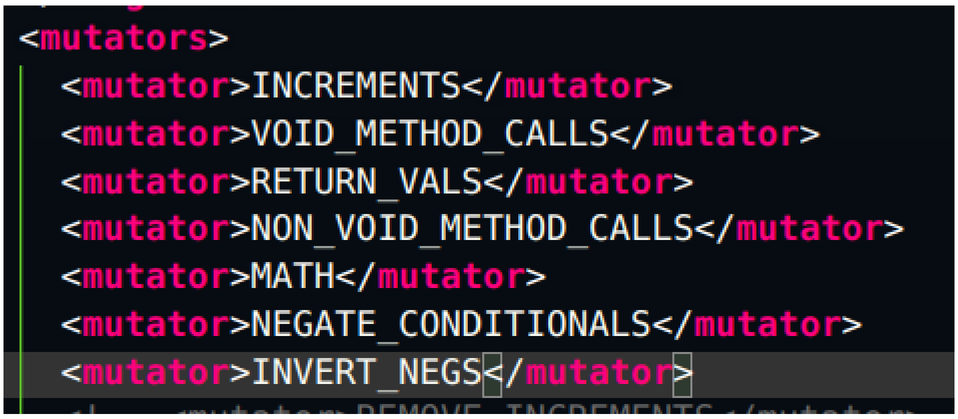
4. Observations And Results

These are the unit tests performed.



```
22/22                          1.6s
demo 17ms
  { } com.example 17ms
    StringLibraryTest 17ms
      reverseWordsTest() 0.0...
      longestPalindromeTest...
      romanToIntTest() 0.0ms
      atoiTest() 1.0ms
      longestCommonPrefixT...
      prefixFunctionTest() 1....
      zFunctionTest() 1.0ms
      firstOccurrenceTest() 0...
      editDistanceTest() 0.0ms
      longestCommonSubse...
      palindromePartitioning...
      checkPangramTest() 2....
      toLowercaseTest() 1.0ms
      toUppercaseTest() 0.0ms
      areAnagramsTest() 0.0ms
      countOccurrencesTest(...
      countWordsTest() 0.0ms
      countCharactersTest()...
      minLengthWordTest()...
      maxLengthWordTest()...
      issubsequenceTest() 0....
      reverseStringTest() 0.0ms
```

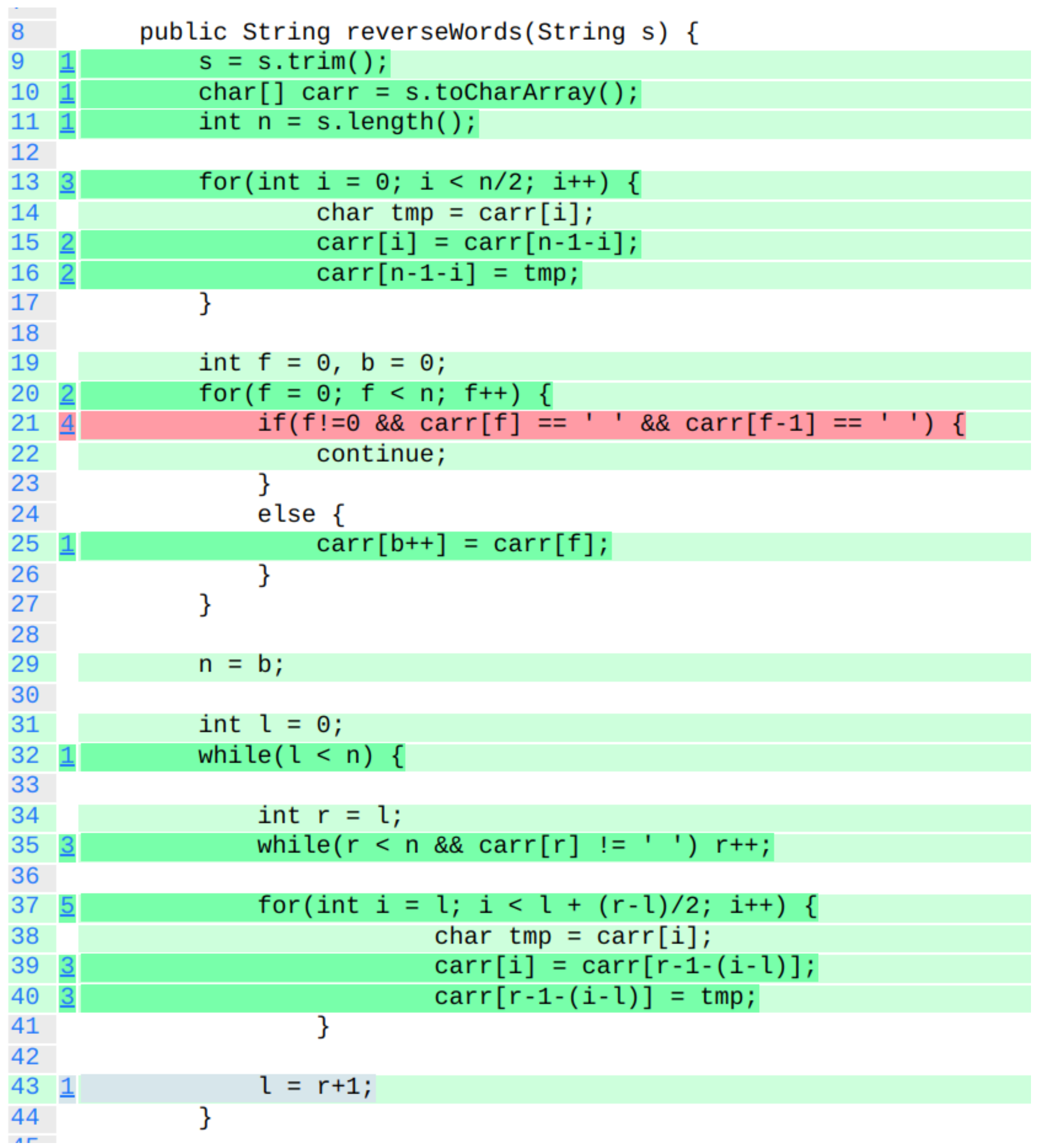
These are the list of mutators used in the project.



```
<mutators>
  <mutator>INCREMENTS</mutator>
  <mutator>VOID_METHOD_CALLS</mutator>
  <mutator>RETURN_VALS</mutator>
  <mutator>NON_VOID_METHOD_CALLS</mutator>
  <mutator>MATH</mutator>
  <mutator>NEGATE_CONDITIONALS</mutator>
  <mutator>INVERT_NEGS</mutator>
  <!--    <mutator>REMOVE_INCREMENTS</mutator>
```

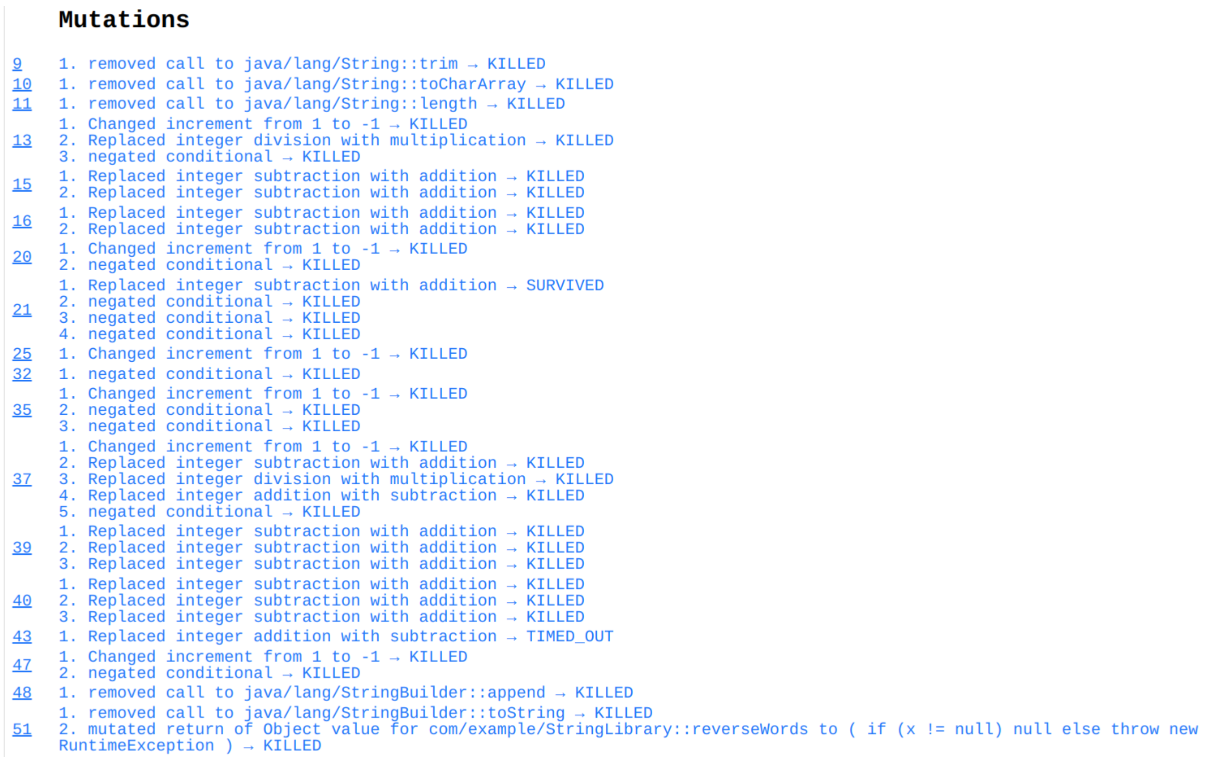
This is a sample from the report generated by PI.

The light green shows line coverage, dark green shows mutation coverage whereas light pink show lack of line coverage and dark pink shows lack of mutation coverage.



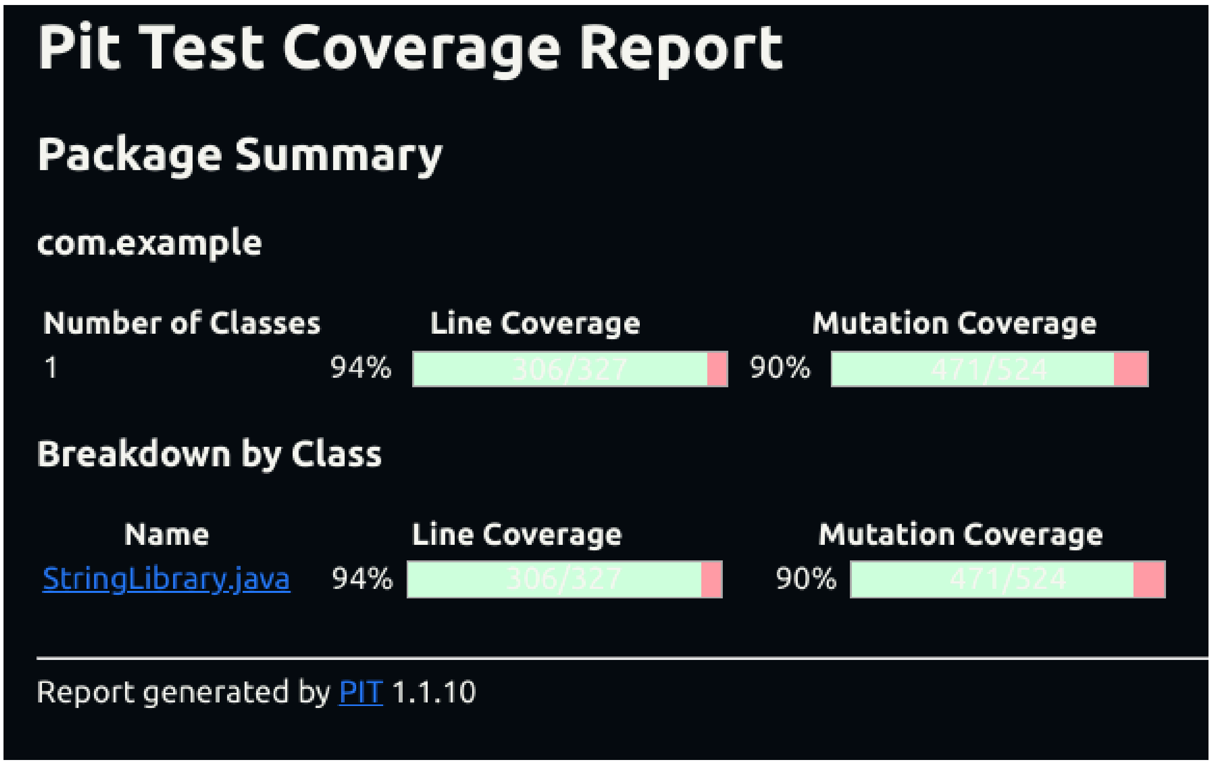
```
      public String reverseWords(String s) {
1         s = s.trim();
1         char[] carr = s.toCharArray();
1         int n = s.length();

3         for(int i = 0; i < n/2; i++) {
              char tmp = carr[i];
2             carr[i] = carr[n-1-i];
2             carr[n-1-i] = tmp;
          }

          int f = 0, b = 0;
2         for(f = 0; f < n; f++) {
4             if(f!=0 && carr[f] == ' ' && carr[f-1] == ' ') {
                  continue;
              }
              else {
1                 carr[b++] = carr[f];
              }
          }

          n = b;

          int l = 0;
1         while(l < n) {

              int r = l;
3             while(r < n && carr[r] != ' ') r++;

5             for(int i = l; i < l + (r-l)/2; i++) {
                      char tmp = carr[i];
3                     carr[i] = carr[r-1-(i-l)];
3                     carr[r-1-(i-l)] = tmp;
                  }

1             l = r+1;
          }
```

Here is the list of mutations applied. Refer Pitest report for detailed analysis.



```
     Mutations

9    1. removed call to java/lang/String::trim → KILLED
10   1. removed call to java/lang/String::toCharArray → KILLED
11   1. removed call to java/lang/String::length → KILLED
     1. Changed increment from 1 to -1 → KILLED
13   2. Replaced integer division with multiplication → KILLED
     3. negated conditional → KILLED
15   1. Replaced integer subtraction with addition → KILLED
     2. Replaced integer subtraction with addition → KILLED
16   1. Replaced integer subtraction with addition → KILLED
     2. Replaced integer subtraction with addition → KILLED
20   1. Changed increment from 1 to -1 → KILLED
     2. negated conditional → KILLED
     1. Replaced integer subtraction with addition → SURVIVED
21   2. negated conditional → KILLED
     3. negated conditional → KILLED
     4. negated conditional → KILLED
25   1. Changed increment from 1 to -1 → KILLED
32   1. negated conditional → KILLED
     1. Changed increment from 1 to -1 → KILLED
35   2. negated conditional → KILLED
     3. negated conditional → KILLED
     1. Changed increment from 1 to -1 → KILLED
     2. Replaced integer subtraction with addition → KILLED
37   3. Replaced integer division with multiplication → KILLED
     4. Replaced integer addition with subtraction → KILLED
     5. negated conditional → KILLED
     1. Replaced integer subtraction with addition → KILLED
39   2. Replaced integer subtraction with addition → KILLED
     3. Replaced integer subtraction with addition → KILLED
     1. Replaced integer subtraction with addition → KILLED
40   2. Replaced integer subtraction with addition → KILLED
     3. Replaced integer subtraction with addition → KILLED
43   1. Replaced integer addition with subtraction → TIMED_OUT
47   1. Changed increment from 1 to -1 → KILLED
     2. negated conditional → KILLED
48   1. removed call to java/lang/StringBuilder::append → KILLED
     1. removed call to java/lang/StringBuilder::toString → KILLED
51   2. mutated return of Object value for com/example/StringLibrary::reverseWords to ( if (x != null) null else throw new
     RuntimeException ) → KILLED
```

This shows the line coverage and mutation coverage metrics for the codebase under test. It can be observed that we were able to obtain a mutation coverage of 90%.



Pit Test Coverage Report

Package Summary

com.example

Number of Classes	Line Coverage		Mutation Coverage	
1	94%	306/327	90%	471/524

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
StringLibrary.java	94%	306/327	90%	471/524

Report generated by PIT 1.1.10

5. References

• PIT Mutation Testing. http://pitest.org/
• Google. (2021, April 12). Mutation Testing. Google Testing Blog. https://testing.googleblog.com/2021/04/mutation-testing.html
• Pacheco, D. (2021, December 15). Java Mutation Testing with Pitest - Diego Pacheco. Medium. https://diego-pacheco.medium.com/java-mutation-testing-with-pitest-1ab12f3564a7
• JavaPoint Mutation Testing. https://www.javatpoint.com/mutation-testing

6. Contributions

Equal contribution by both teammates, as the project was done in meets.

github link