Mutation Testing

CSE 731: Software Testing

Guided By

Meenakshi D Souza
Prof. and HOD of CSE Department

International Institute of Information Technology, Bangalore

International Institute of Information Technology, Bangalore

Created By:

Rishabh Teli
MT2023061
Rishabh.Teli@iiitb.ac.in

Nikita Gupta
MT2023132
Nikita.Gupta@iiitb.ac.in

ABSTRACT

This project focuses on designing and implementing a comprehensive testing framework for algorithmic solutions, emphasizing the use of advanced testing techniques to ensure high reliability and robustness. The initial phase involved configuring the development environment, which included installing essential dependencies, integrating necessary plugins, and ensuring seamless compatibility for testing tools. A systematic approach was followed to design unit test cases for various algorithms, ensuring maximum code coverage and addressing edge cases effectively.

The core testing methodology utilized mutation testing, conducted using PIT (Pitest), a state-of-the-art tool for evaluating the effectiveness of test cases. This process involved introducing controlled changes (mutants) into the code and verifying the ability of the test suite to detect these faults. The mutation score provided quantitative insights into the robustness of the test cases. Additionally, third-party tools were integrated into the pipeline to generate detailed statistical reports. These reports offered an in-depth analysis of mutation metrics, such as the percentage of mutants killed and the areas requiring improvement, enabling iterative refinement of the test cases.

The project also explored the integration of automation to streamline the testing workflow, ensuring consistency and reducing manual intervention. By combining unit testing, mutation testing, and detailed statistical analysis, this project demonstrates a scalable and efficient approach to enhancing the quality and reliability of software systems. It serves as a practical demonstration of modern software testing practices, emphasizing precision, automation, and data-driven decision-making to achieve robust software validation.

TABLE OF CONTENTS

Link to Source Code

1. Mutation Testing

Mutation testing is a type of software testing where the system introduces small changes (mutations) to the code to verify if the existing test suite can detect these changes. If a test fails when the code is mutated, it's considered effective.

1.1 Overview

Tests are essential for verifying the correctness of a software system's implementation. However, their creation raises a fundamental question: how can we ensure that the tests themselves are correct and adequately cover the requirements that drove the implementation? This challenge reflects the broader philosophical problem of "Quis custodiet ipsos custodes?" or "Who will guard the guards?"

Mutation testing addresses this issue by introducing small changes, or "mutants," into the code. These mutants simulate bugs, and a well-designed test suite should be able to detect them. If a test suite fails to identify a mutant, it typically indicates a gap in the test suite's ability to catch the types of faults represented by the mutant. However, there are exceptions. Some mutants may represent valid changes that do not introduce faults—for instance, modifications to "dead code" that are never executed.

To achieve effective mutation testing, a large number of mutants are often introduced, resulting in the compilation and execution of numerous program variants. This process can be resource-intensive, historically limiting the practicality of mutation testing. However, the rise of object-oriented programming languages and the widespread adoption of unit testing frameworks have led to the development of mutation testing tools capable of efficiently targeting specific portions of an application. These advancements have made mutation testing a more feasible and valuable method in modern software development.

1.2 Implementation

We have implemented mutation testing across various algorithms within our project. Mutation testing is a highly effective technique for evaluating the robustness of a test suite. It works by introducing small changes (mutations) to the source code and verifying whether the test suite can successfully detect these alterations, simulating potential bugs.

For this purpose, we utilized the **PIT (Pitest)** framework, a widely recognized mutation testing tool for Java. PIT streamlines the process by automating the generation and analysis of code mutations, providing insights into test suite effectiveness and identifying areas for improvement.

Our project consists of over **1,000 lines of code**, encompassing multiple algorithms and functionalities. By applying mutation testing, we aimed to ensure that our test suite not only covers the codebase thoroughly but also reliably detects subtle faults, thereby enhancing the overall quality and resilience of the application.

2. Test Case Strategy

Our test case strategy adopted a comprehensive approach to mutation testing, emphasizing the robustness and reliability of the project's algorithms. The focus was on systematically targeting a diverse set of potential code alterations, including mutations in conditionals, arithmetic operations, and return values.

2.1 Objectives:

1. **Detection and Resolution of 'KILLED' Mutations**
   The primary goal was to identify and address mutations categorized as *'KILLED.'* These represent successfully detected and resolved code changes, ensuring the stability and correctness of the underlying algorithms.
2. **Improving Coverage for 'SURVIVED' Mutations**
   Mutations classified as *'SURVIVED'* indicated weaknesses in the current test suite's coverage. These areas were prioritized for enhanced test case development to better detect subtle faults and ensure more comprehensive protection against potential defects.
3. **Eliminating 'NO_COVERAGE' Gaps**
   Addressing sections of the codebase marked as *'NO_COVERAGE'* was integral to our strategy. By filling these gaps, we aimed to ensure thorough and consistent evaluation of the entire codebase, leaving no potential issue unexamined.

2.2 Approach:

- **Iterative Refinement:**
  The test suite underwent continuous refinement to adapt to newly introduced code changes. This iterative process ensured that the testing framework evolved alongside the project, maintaining alignment with its growing complexity.
- **Focus on Algorithm Reliability:**
  By targeting critical code areas and introducing diverse mutations, the strategy aimed to solidify the algorithms' resilience against unforeseen issues, ensuring long-term stability and performance.

This approach enabled a robust mutation testing framework, minimizing undetected faults and enhancing the overall quality of the project's codebase.

3. Tools Used

3.1 IntelliJ IDEA for Java

IntelliJ IDEA is a renowned integrated development environment (IDE) tailored for Java development. It is celebrated for its:

- **Robust Features**: Includes intelligent code completion, in-depth analysis, and error detection.
- **User-Friendly Interface**: Offers an intuitive and customizable workspace for developers.
- **Plugin Support**: Provides access to a vast library of plugins, enabling integration with various tools and frameworks.

These features make IntelliJ IDEA a productivity powerhouse for efficient coding, debugging, and testing, ensuring high-quality development workflows.

3.2 PIT Mutation Testing Tool

PIT is a powerful and easy-to-use mutation testing tool designed for Java applications. It operates by introducing small code modifications (mutations) to evaluate the efficiency of the test suite in identifying these changes. Key benefits include:

- **Automation**: Automatically generates mutated versions of the source code.
- **Insightful Analysis**: Measures the effectiveness of the existing test suite by detecting undetected mutants.
- **Seamless Integration**: PIT can be easily integrated into development workflows using plugins like **PITclipse** for Eclipse or **Maven** for build management.

These tools collectively streamline development and testing processes, ensuring comprehensive validation and high code quality.

3.3 Mutators in PIT:

```
<mutators>
    <mutator>CONDITIONALS_BOUNDARY</mutator>
    <mutator>EMPTY_RETURNS</mutator>
    <mutator>FALSE_RETURNS</mutator>
    <mutator>INCREMENTS</mutator>
    <mutator>INVERT_NEGS</mutator>
    <mutator>MATH</mutator>
    <mutator>NEGATE_CONDITIONALS</mutator>
    <mutator>NULL_RETURNS</mutator>
    <mutator>PRIMITIVE_RETURNS</mutator>
    <mutator>TRUE_RETURNS</mutator>
    <mutator>VOID_METHOD_CALLS</mutator>
    <mutator>NON_VOID_METHOD_CALLS</mutator>
    <mutator>EXPERIMENTAL_ARGUMENT_PROPAGATION</mutator>
    <mutator>EXPERIMENTAL_NAKED_RECEIVER</mutator>
</mutators>
```

Each **mutator** represents a type of change (mutation) that PIT can introduce to your code during testing. Here's what the specific mutators in your configuration mean:

1. **CONDITIONALS_BOUNDARY**: Modifies conditional statements like > to >=.
2. **EMPTY_RETURNS**: Replaces return statements in void methods with empty returns.
3. **FALSE_RETURNS**: Forces methods to always return false.
4. **INCREMENTS**: Modifies increments/decrements, e.g., i++ to i--.
5. **INVERT_NEGS**: Inverts negative numbers to positive.
6. **MATH**: Changes basic mathematical operations, like + to -.
7. **NEGATE_CONDITIONALS**: Negates conditionals, turning if (x) into if (!x).
8. **NULL_RETURNS**: Forces methods to return null.
9. **PRIMITIVE_RETURNS**: Changes return values of primitive data types.
10. **TRUE_RETURNS**: Forces methods to always return true.
11. **VOID_METHOD_CALLS**: Removes calls to void methods.
12. **NON_VOID_METHOD_CALLS**: Removes calls to methods that return a value.
13. **EXPERIMENTAL_ARGUMENT_PROPAGATION**: Modifies how arguments are passed to methods.
14. **EXPERIMENTAL_NAKED_RECEIVER**: Experiments with replacing method receivers in a context-sensitive way.

In PIT (Pitest), **mutators** represent the types of changes that can be introduced into the code during mutation testing. These mutators allow developers to test the effectiveness of their test suites by simulating different kinds of code alterations. PIT provides three primary predefined sets of mutators: **ALL**, **DEFAULTS**, and **STRONGER**.

3.3.1 ALL Mutators

- **Description**:
  The ALL configuration includes every mutator that PIT provides. This means that every possible mutation type will be applied to the code. While comprehensive, this can lead to a significant increase in processing time and may generate mutations that are not always relevant to your project.
- **Use Case**:
  Use the ALL mutators when you need the most exhaustive mutation testing, such as when performing a full audit of test suite effectiveness for a critical or legacy system.

- **Example in Configuration**:

```
<!--                    All    -->
                <mutators>
                    <mutator>ALL</mutator>
                </mutators>
```

3.3.3 DEFAULTS Mutators

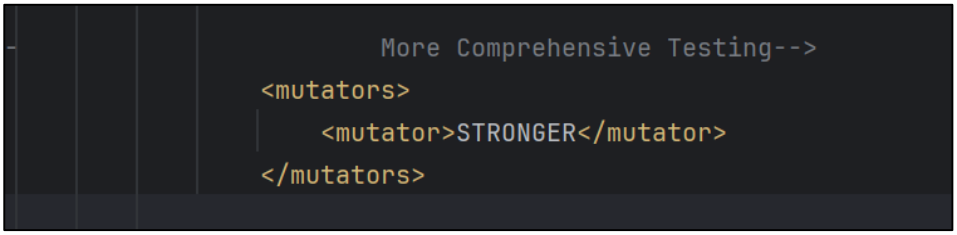
- **Description**:
  The DEFAULTS configuration includes a carefully curated subset of mutators that strike a balance between thoroughness and performance. These mutators are chosen to represent the most common code changes that could occur in real-world scenarios, making it suitable for most use cases.
- **Includes**:
  - **CONDITIONALS_BOUNDARY**: Changes conditionals (e.g., < to <=).
  - **INCREMENTS**: Modifies increment and decrement operators (e.g., i++ to i--).
  - **RETURN_VALS**: Alters return values.
  - **VOID_METHOD_CALLS**: Removes calls to void methods.
  - **MATH**: Modifies mathematical operations (e.g., + to -).
  - **NEGATE_CONDITIONALS**: Negates boolean conditions.
- **Use Case**:
  Suitable for most projects where mutation testing needs to focus on key areas without introducing unnecessary overhead.
- **Example in Configuration**:

```
<!--                    Default-->
                <mutators>
                    <mutator>DEFAULTS</mutator>
                </mutators>
```

3.3.4 STRONGER Mutators

- **Description**:
  The STRONGER configuration includes a stricter set of mutators for advanced testing. This set not only contains the **DEFAULTS** mutators but also introduces additional mutation types to challenge the robustness of the test suite further.

- **Includes**:
    - All **DEFAULTS** mutators.
    - Additional experimental or less common mutators, such as those affecting argument propagation and deeper logical operations.
- **Use Case**:
  Ideal for teams aiming for highly resilient test suites, such as those developing critical software systems where faults are unacceptable.
- **Example in Configuration**:



Key Considerations

- **Performance**:
    - Using **ALL** mutators will significantly increase testing time and may introduce irrelevant mutations.
    - **DEFAULTS** is generally the most balanced option in terms of performance and relevance.
    - **STRONGER** may also require additional computational resources but provides more robust testing.
- **Relevance**:
  Consider the project's requirements when choosing a set of mutators. For most teams, **DEFAULTS** is sufficient, while **STRONGER** and **ALL** are better suited for in-depth testing of mission-critical systems.

4. Why Configure Mutators?

Customizing mutators allows developers to target specific types of logic in their code that they suspect might be prone to errors. Using relevant mutators ensures the mutation testing process aligns with your project's needs and helps improve your test suite effectively.

In our case, the configuration specifies a comprehensive set of mutators, including some experimental ones, for thorough mutation testing of the specified classes.

Run PIT runner

It can be run directly from the command-line:

mvn test-compile org.pitest:pitest-maven:mutationCoverage

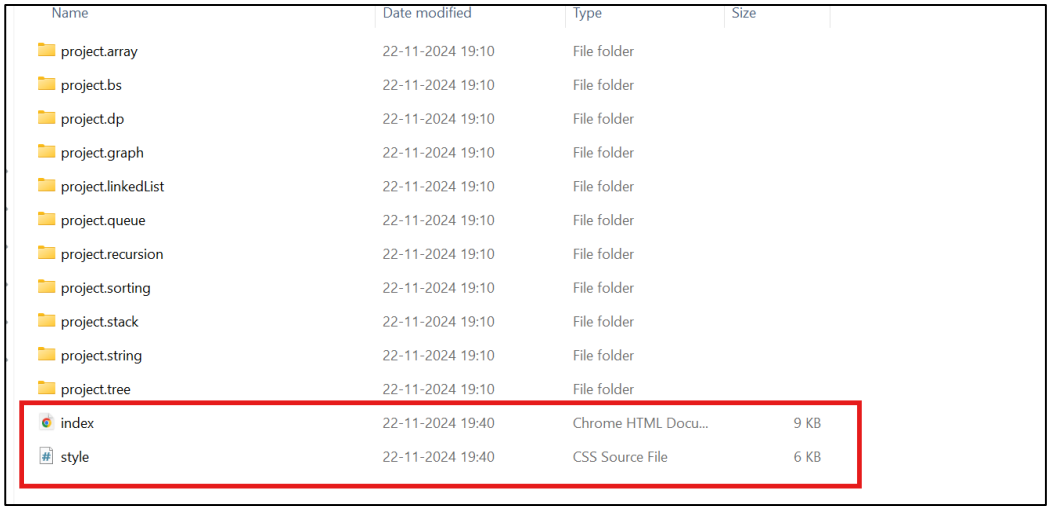
this command will create mutants and run the mutation testing on the codebase, the resultant report will be generated in the target folder as an HTML document.

```
PS E:\IIITB\_Sem_3\ST\END\Project\software_testing> mvn org.pitest:pitest-maven:mutationCoverage
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< project:software_testing >----------------------
[INFO] Building software_testing 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- pitest:1.15.2:mutationCoverage (default-cli) @ software_testing ---
[INFO] Root dir is : E:\IIITB\_Sem_3\ST\END\Project\software_testing
[INFO] Found plugin : Default csv report plugin
[INFO] Found plugin : Default xml report plugin
[INFO] Found plugin : Default html report plugin
[INFO] Found plugin : Static initializer code detector plugin
```

```
- Statistics
================================================================================
>> Line Coverage (for mutated classes only): 1022/1038 (98%)
>> Generated 1020 mutations Killed 902 (88%)
>> Mutations with no coverage 9. Test strength 89%
>> Ran 1067 tests (1.05 tests per mutation)
Enhanced functionality available at https://www.arcmutate.com/
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:35 min
[INFO] Finished at: 2024-11-22T19:40:23+05:30
[INFO] ------------------------------------------------------------------------
PS E:\IIITB\_Sem_3\ST\END\Project\software_testing>
```

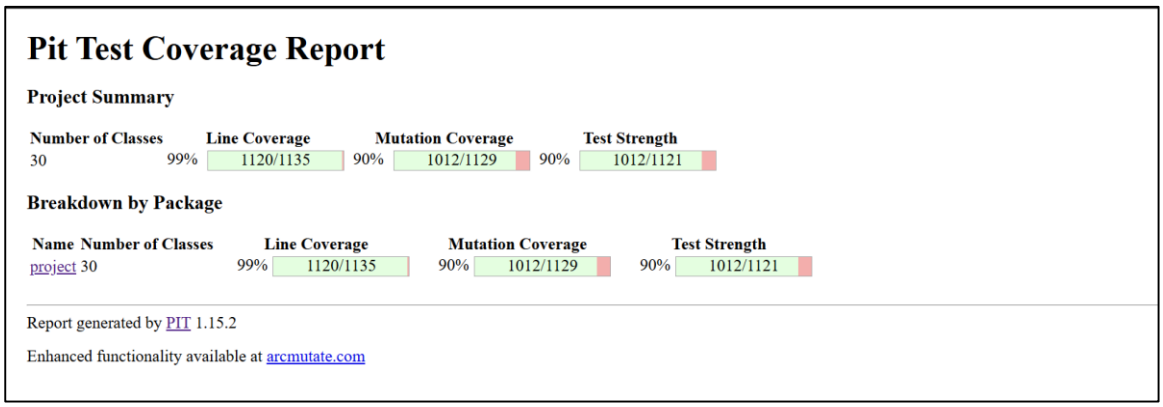
5. Reports

View the generated reports:
After the execution is complete, we can find the HTML report in the target/pit-reports directory. Open the index.html file in a browser to view the detailed mutation testing report.



5.1 Pit Test Coverage Report:

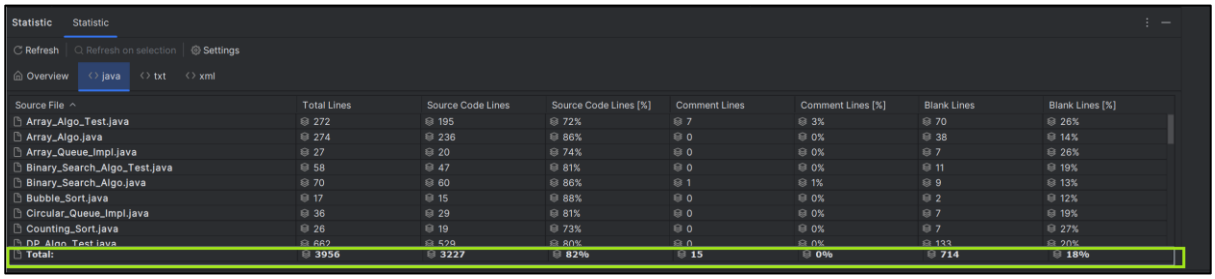
The **Pit Test Coverage Report** provides detailed insights into the effectiveness of the test suite when subjected to mutation testing. It identifies the total mutants generated, the percentage of mutants killed, and highlights areas of the codebase with insufficient test coverage. The key metrics used in the report are:
Mutants Generated: The total number of mutations introduced by Pitest.
Mutants Killed: The number of mutants successfully detected by the test suite.
Survived Mutants: The mutants that were not detected by the test suite.
Mutation Coverage Score: A percentage representing the effectiveness of the test suite in killing mutants.



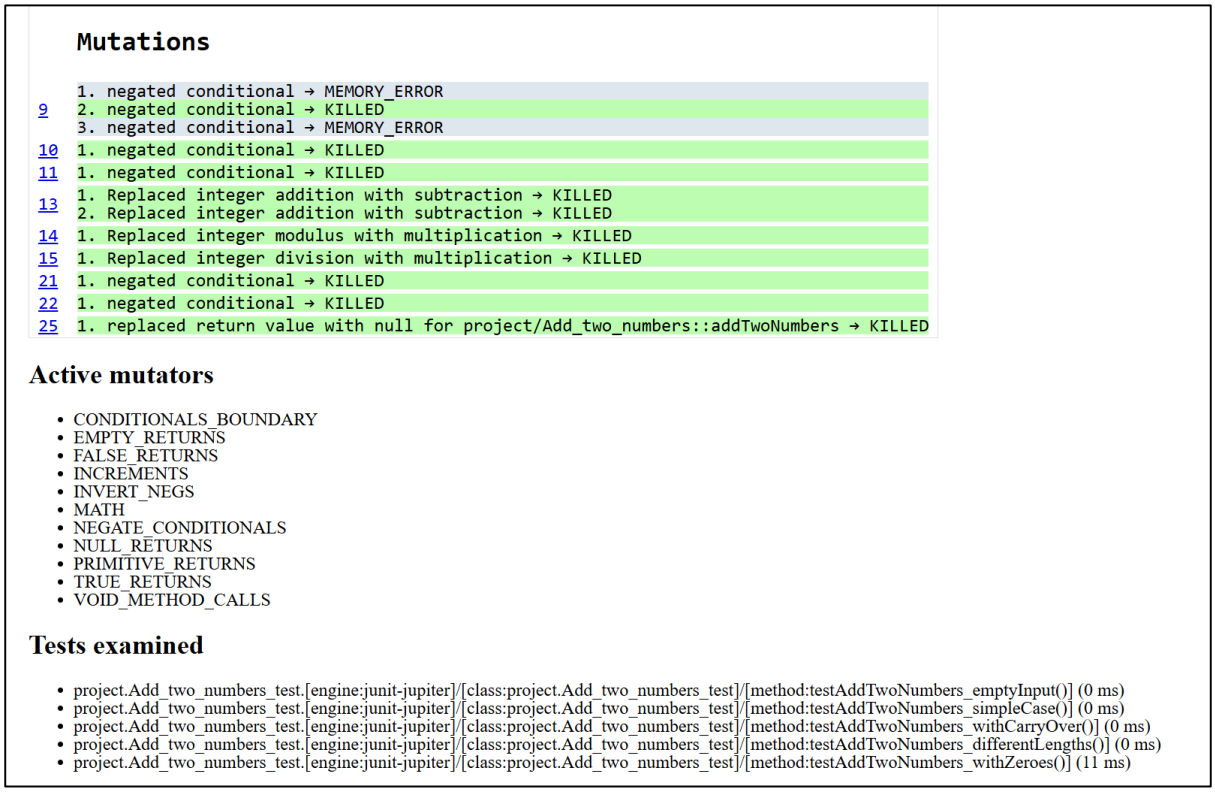
Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Add_two_numbers.java	100%	17/17	100%	12/12	100%	12/12
Array_Algo.java	99%	161/163	82%	132/161	83%	132/159
Array_Queue_Impl.java	100%	8/8	100%	4/4	100%	4/4
Beautiful_Arrangement.java	100%	17/17	97%	28/29	97%	28/29
Binary_Search_Algo.java	98%	45/46	84%	51/61	88%	51/58
Bubble_Sort.java	89%	8/9	75%	9/12	75%	9/12
Circular_Queue_Impl.java	100%	16/16	100%	12/12	100%	12/12
Counting_Sort.java	92%	11/12	100%	9/9	100%	9/9
DP_Algo.java	99%	267/268	92%	314/341	92%	314/340
Generate_Parenthesis.java	100%	12/12	100%	12/12	100%	12/12
Graph_Algo.java	100%	48/48	86%	12/14	86%	12/14
Heap_Algo.java	100%	32/32	94%	30/32	94%	30/32
Insertion_Sort.java	90%	9/10	90%	9/10	90%	9/10
KMP_Algo.java	100%	12/12	100%	13/13	100%	13/13
Linked_List.java	100%	35/35	100%	18/18	100%	18/18
Merge.java	96%	26/27	89%	24/27	89%	24/27
Merge_Sort.java	86%	6/7	100%	9/9	100%	9/9
Next_permutation.java	96%	22/23	81%	22/27	88%	22/25
PalindromeOrNot.java	83%	5/6	86%	6/7	86%	6/7
PalindromePartitioning.java	100%	13/13	100%	12/12	100%	12/12
Partition.java	92%	12/13	80%	8/10	80%	8/10
Quick_Sort.java	83%	5/6	67%	4/6	67%	4/6
Radix_Sort.java	95%	21/22	95%	20/21	95%	20/21
Recursion_Algo.java	100%	27/27	81%	25/31	81%	25/31
Selection_Sort.java	91%	10/11	75%	6/8	75%	6/8
Sliding_Window_Algo.java	100%	51/51	92%	57/62	92%	57/62
Stack_Algo.java	100%	83/83	86%	60/70	86%	60/70
String_Algo.java	100%	34/34	90%	26/29	90%	26/29
Tree_Algo.java	100%	100/100	100%	60/60	100%	60/60
Two_sum.java	100%	7/7	80%	8/10	80%	8/10

These are the statistics from the IntelliJ Plugin called Statistic. It's a third party plugin that can be installed through IntelliJ



As you can see in the detailed report of the individual class you can find the mutations and mutators used for the mutations and the status of the mutators (SURVIVED / KILLED).



Mutations

```
    1. negated conditional → MEMORY_ERROR
9   2. negated conditional → KILLED
    3. negated conditional → MEMORY_ERROR
10  1. negated conditional → KILLED
11  1. negated conditional → KILLED
13  1. Replaced integer addition with subtraction → KILLED
    2. Replaced integer addition with subtraction → KILLED
14  1. Replaced integer modulus with multiplication → KILLED
15  1. Replaced integer division with multiplication → KILLED
21  1. negated conditional → KILLED
22  1. negated conditional → KILLED
25  1. replaced return value with null for project/Add_two_numbers::addTwoNumbers → KILLED
```

Active mutators

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

Tests examined

- project.Add_two_numbers_test.[engine:junit-jupiter]/[class:project.Add_two_numbers_test]/[method:testAddTwoNumbers_emptyInput()] (0 ms)
- project.Add_two_numbers_test.[engine:junit-jupiter]/[class:project.Add_two_numbers_test]/[method:testAddTwoNumbers_simpleCase()] (0 ms)
- project.Add_two_numbers_test.[engine:junit-jupiter]/[class:project.Add_two_numbers_test]/[method:testAddTwoNumbers_withCarryOver()] (0 ms)
- project.Add_two_numbers_test.[engine:junit-jupiter]/[class:project.Add_two_numbers_test]/[method:testAddTwoNumbers_differentLengths()] (0 ms)
- project.Add_two_numbers_test.[engine:junit-jupiter]/[class:project.Add_two_numbers_test]/[method:testAddTwoNumbers_withZeroes()] (11 ms)

Active Mutators applied in our codebase:

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

6. SURVIVED and KILLED Mutants

Mutation testing is a software testing technique designed to evaluate the quality of test cases by introducing small modifications (mutations) into the source code. These mutations simulate potential errors that a developer might introduce. The goal is to check if the existing test cases can detect and fail due to the introduced changes. The effectiveness of mutation testing is measured by how many introduced mutants are detected ("KILLED") versus those that remain undetected ("SURVIVED").

6.1 SURVIVED Mutants:

Mutants are said to have *SURVIVED* when the test suite fails to detect the mutation in the code.
Let's take an example of twoSum function of class Array_Algo. It suggests that sorting was removed during mutation testing, and the test cases did not fail. This indicates that the twoSum implementation does not rely on sorting the array after the mutation. If sorting was expected to be a crucial step, this must be a testcase that proves that. So, in order to kill that mutant, we can add new testcase as mentioned in the comment in the below screenshot.

```
public class Array_Algo_Test {

    @Test
    public void towSumTest(){

        Array_Algo array_algo1 = new Array_Algo();
//        //removed call to java/util/Arrays::sort → SURVIVED
//        int[] arr1 = {1,9,3,7,8};
//        Assertions.assertTrue(array_algo1.twoSum(arr1,9));
        int[] arr1 = {9,1,3,7,1};
        Assertions.assertTrue(array_algo1.twoSum(arr1,target:12));

        Array_Algo array_algo2 = new Array_Algo();
        int[] arr2 = {3,7,8,9,1};
        Assertions.assertTrue(array_algo2.twoSum(arr2,target:9));

        Array_Algo array_algo3 = new Array_Algo();
        int[] arr3 = {1,9,3,7,8};
        Assertions.assertFalse(array_algo3.twoSum(arr3,target:3));

        Array_Algo array_algo4 = new Array_Algo();
        int[] arr4 = {3,7,8,9,1};
        Assertions.assertFalse(array_algo4.twoSum(arr4,target:3));

//        //changed conditional boundary → SURVIVED
        Array_Algo array_algo5 = new Array_Algo();
        int[] arr5 = {5,2,3,4,5};
        Assertions.assertFalse(array_algo5.twoSum(arr5,target:4));

    }
```

6.2 KILLED Mutants:

Definition: Mutants are said to be *KILLED* when the test suite detects the mutation, causing the relevant test cases to fail.

Example: If a mutation changes x = x + 1 to x = x - 1, and a test case fails due to incorrect output caused by the mutation, the mutant is killed.
Reason: The test suite includes specific cases to validate the correctness of the increment logic.
Example: Mutating a loop condition i < n to i <= n, and the test fails due to an infinite loop or an off-by-one error in the output.
Reason: Comprehensive test coverage and proper assertions ensure such logic changes are caught.

7. How to Improve Mutation Testing

Contribution to Test Suite

- Increase Code Coverage: Add test cases to cover untested branches, conditions, and edge cases.Use tools like JaCoCo or Pitest to identify uncovered code.
- Refine Assertions: Ensure test cases include meaningful assertions to verify specific behaviors, outputs, or state changes.Avoid superficial assertions that do not validate critical parts of the logic.
- Optimize Test Case Design:Use boundary value analysis, equivalence partitioning, and decision table testing to create diverse test cases.Write test cases for all possible input combinations, including edge cases.
- Regular Mutation Testing Execution:Integrate mutation testing into the CI/CD pipeline to continuously assess and improve the quality of the test suite.
- Analyze mutation reports to identify and fix weak areas.
- Collaboration and Feedback:Encourage developers and testers to review mutation testing results and propose targeted improvements to the test cases.Use mutation testing outcomes as a learning tool to improve coding and testing practices.

8. Mutation testing at integration level

PalindromePartitioning.java

```
1   package project;
2
3   import java.util.Arrays;
4
5   public class PalindromePartitioning {
6       public int palindromePartitioning(String str) {
7           int n = str.length();
8           int[] dp = new int[n];
9   1       Arrays.fill(dp, -1);
10  2       return helper(0, n, str, dp) - 1;
11      }
12
13      static int helper(int i, int n, String str, int[] dp) {
14  1       if (i == n) return 0;
15
16  2       if (dp[i] != -1) return dp[i];
17          int minCost = Integer.MAX_VALUE;
18  2       for (int j = i; j < n; j++) {
19  1           if (PalindromeOrNot.isPalindrome(i, j, str)) {
20  2               int cost = 1 + helper(j + 1, n, str, dp);
21                  minCost = Math.min(minCost, cost);
22              }
23          }
24  1       return dp[i] = minCost;
25      }
26  }
```

Mutations

```
9   1. removed call to java/util/Arrays::fill → KILLED
10  1. replaced int return with 0 for project/PalindromePartitioning::palindromePartitioning → KILLED
    2. Replaced integer subtraction with addition → KILLED
14  1. negated conditional → KILLED
16  1. negated conditional → KILLED
    2. replaced int return with 0 for project/PalindromePartitioning::helper → KILLED
18  1. negated conditional → KILLED
    2. changed conditional boundary → KILLED
19  1. negated conditional → KILLED
20  1. Replaced integer addition with subtraction → KILLED
    2. Replaced integer addition with subtraction → KILLED
24  1. replaced int return with 0 for project/PalindromePartitioning::helper → KILLED
```

Integration testing focus on how two or more class interacts with each other and tests that the integrated system behaves as expected or not when mutations are introduced.

Conclusion

We utilized the PIT tool for mutation testing and achieved a **mutation coverage of 90%.** The detailed reports generated by PIT provided insights into the survived mutants, enabling us to refine and enhance our test cases further. By modifying the test suite based on these reports, we can effectively target the surviving mutants and improve the overall mutation coverage, ensuring a more robust and reliable testing framework.

Contribution:

Rishabh Teli:
I handled the initial project setup, including configuring the environment and installing necessary dependencies and plugins. I developed unit test cases for multiple algorithms, focusing on achieving high code coverage and testing edge cases. I executed mutation testing using PIT (Pitest) to evaluate the effectiveness of the test cases and identify weak areas. Additionally, I integrated third-party tools to generate statistical reports, ensuring a detailed analysis of test results.

Nikita Gupta:
My teammate contributed to configuring the project environment and ensuring proper integration of dependencies and plugins. They collaborated in designing and implementing unit test cases for the remaining algorithms, emphasizing robust test design. They conducted mutation testing using PIT on specific sections and validated the generated results. They also optimized and refined the statistical reports produced by third-party tools for clarity and precision.

References:

- https://youtu.be/wZeZMtqVmck?si=bBOwZdIXRbbK97F0
- https://medium.com/geekculture/mutation-testing-for-maven-project-using-pitest-f9b8fef03a05
- https://www.baeldung.com/java-mutation-testing-with-pitest
- www.leetcode.com
- www.geeksforgeeks.org