COEN 6761 Software Testing and Validation

White-Box and Black box Testing QA Report

Student Name	Student ID
Harshilsinh Solanki	40298679
Vraj Shah	40311967
Tanisha Fonseca	40266436
Teja Sayila	40321469

Submitted To:

Dr. Yan Liu

Table of Content:

1. Introduction:

This document presents a detailed QA report on the Robot Movement Simulator application. The primary focus is on automated test cases, coverage analysis, mutation testing, and quality assessment.

1.2 Objective:

The main objective is to ensure that the application meets its functional requirements and demonstrates a high level of code coverage using unit tests, including mutation and dataflow testing. The results will guide improvements and highlight any existing gaps in test quality.

1.3 GitHub Url:

https://github.com/tanishaf28/Coverage.git

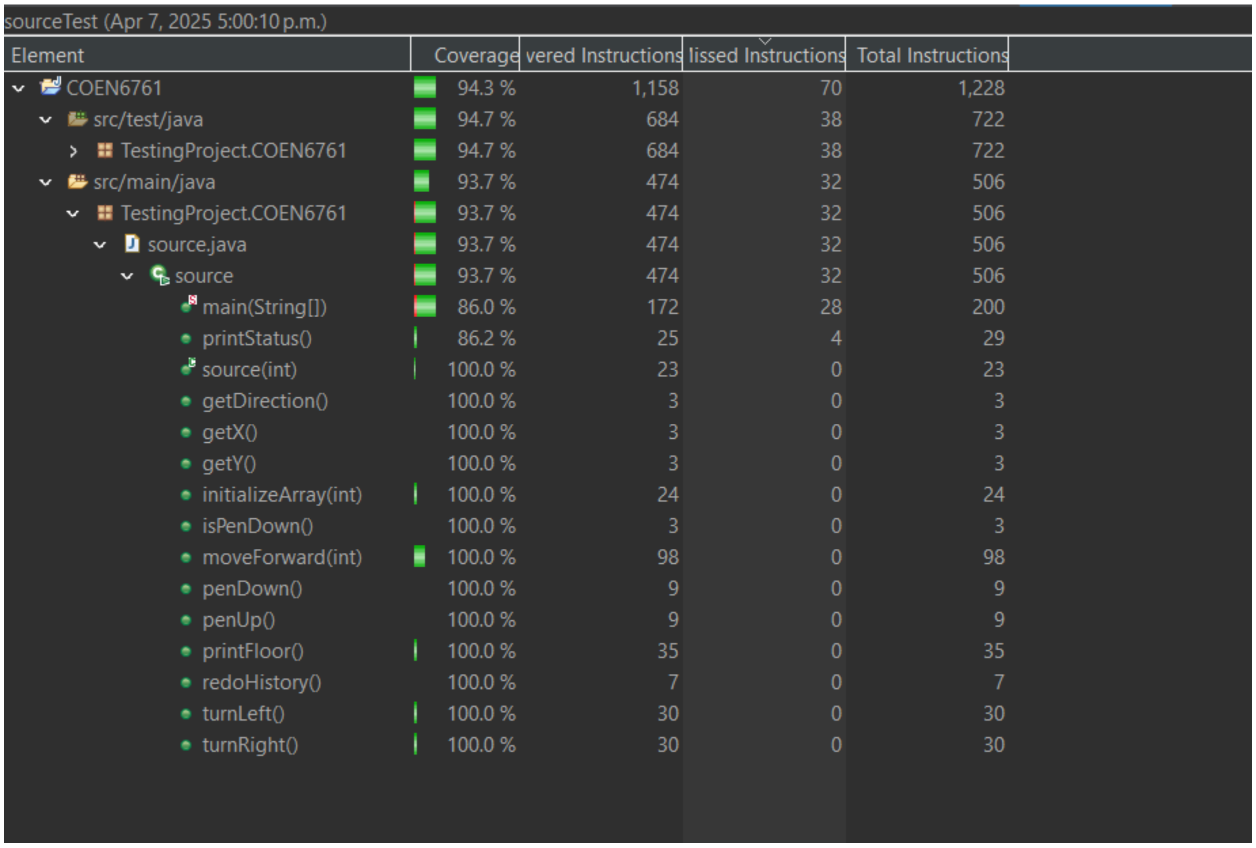
1.4 Tools and Technology Used:

- **JUnit 5.3.1** for unit testing
- **Jacoco 0.8.8** for code coverage
- **PIT Mutation Testing** for mutation analysis
- Java 23, Maven, IntelliJ IDEA, Ubuntu 24.04
- JUnit Platform Console Launcher for CLI testing
- **Eclipse** / **IntelliJ** screenshots for visual verification

2. Test Cases and Modifications

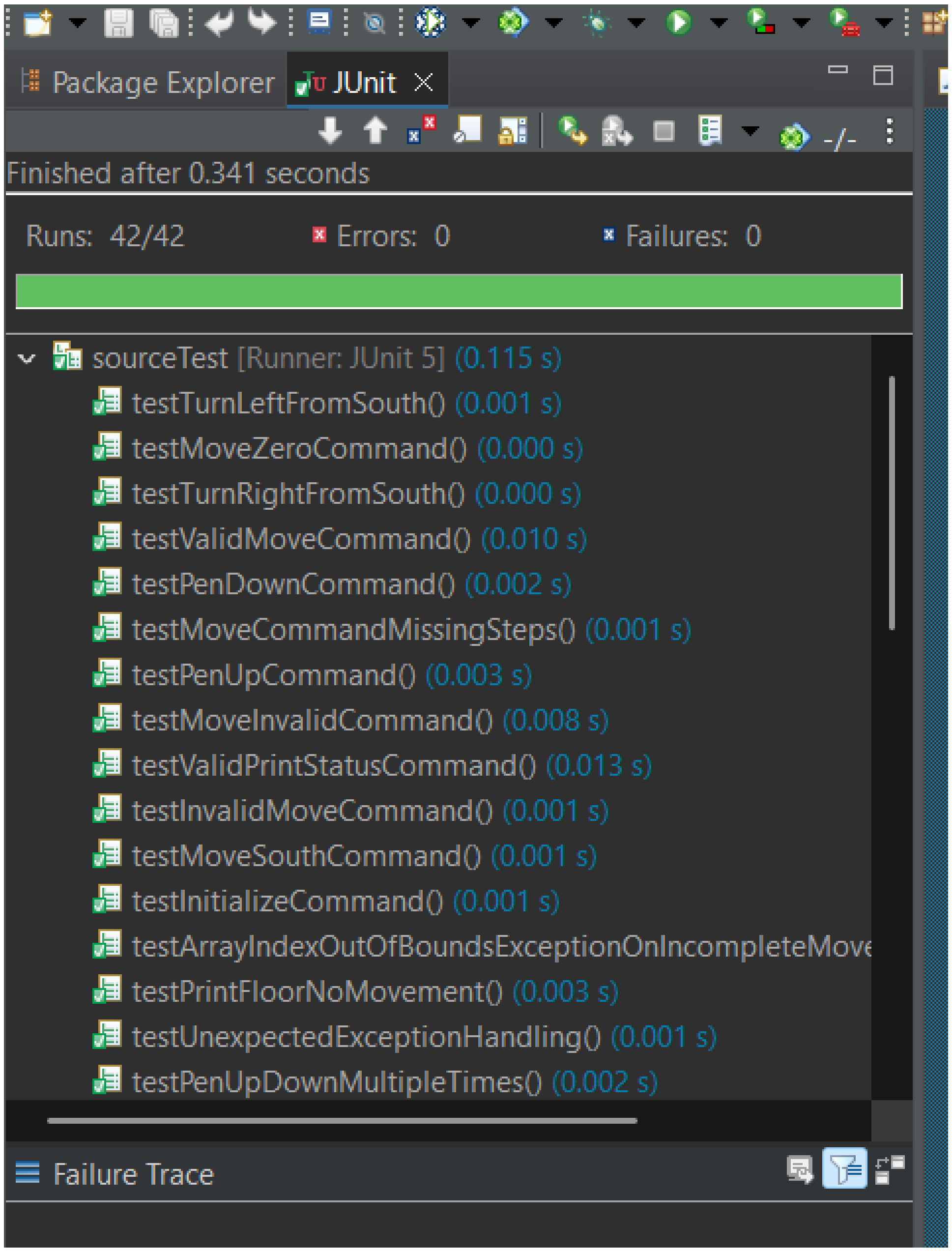
Source Test Coverage

All functions have 100% coverage, print status has 86.2%.Given below is a comparison of the old and new test cases.



sourceTest (Apr 7, 2025 5:00:10 p.m.)

Element	Coverage	vered Instructions	lissed Instructions	Total Instructions
COEN6761	94.3 %	1,158	70	1,228
src/test/java	94.7 %	684	38	722
TestingProject.COEN6761	94.7 %	684	38	722
src/main/java	93.7 %	474	32	506
TestingProject.COEN6761	93.7 %	474	32	506
source.java	93.7 %	474	32	506
source	93.7 %	474	32	506
main(String[])	86.0 %	172	28	200
printStatus()	86.2 %	25	4	29
source(int)	100.0 %	23	0	23
getDirection()	100.0 %	3	0	3
getX()	100.0 %	3	0	3
getY()	100.0 %	3	0	3
initializeArray(int)	100.0 %	24	0	24
isPenDown()	100.0 %	3	0	3
moveForward(int)	100.0 %	98	0	98
penDown()	100.0 %	9	0	9
penUp()	100.0 %	9	0	9
printFloor()	100.0 %	35	0	35
redoHistory()	100.0 %	7	0	7
turnLeft()	100.0 %	30	0	30
turnRight()	100.0 %	30	0	30



```
Finished after 0.341 seconds
Runs:  42/42          Errors:  0          Failures:  0

sourceTest [Runner: JUnit 5] (0.115 s)
    testTurnLeftFromSouth() (0.001 s)
    testMoveZeroCommand() (0.000 s)
    testTurnRightFromSouth() (0.000 s)
    testValidMoveCommand() (0.010 s)
    testPenDownCommand() (0.002 s)
    testMoveCommandMissingSteps() (0.001 s)
    testPenUpCommand() (0.003 s)
    testMoveInvalidCommand() (0.008 s)
    testValidPrintStatusCommand() (0.013 s)
    testInvalidMoveCommand() (0.001 s)
    testMoveSouthCommand() (0.001 s)
    testInitializeCommand() (0.001 s)
    testArrayIndexOutOfBoundsExceptionOnIncompleteMove
    testPrintFloorNoMovement() (0.003 s)
    testUnexpectedExceptionHandling() (0.001 s)
    testPenUpDownMultipleTimes() (0.002 s)
```

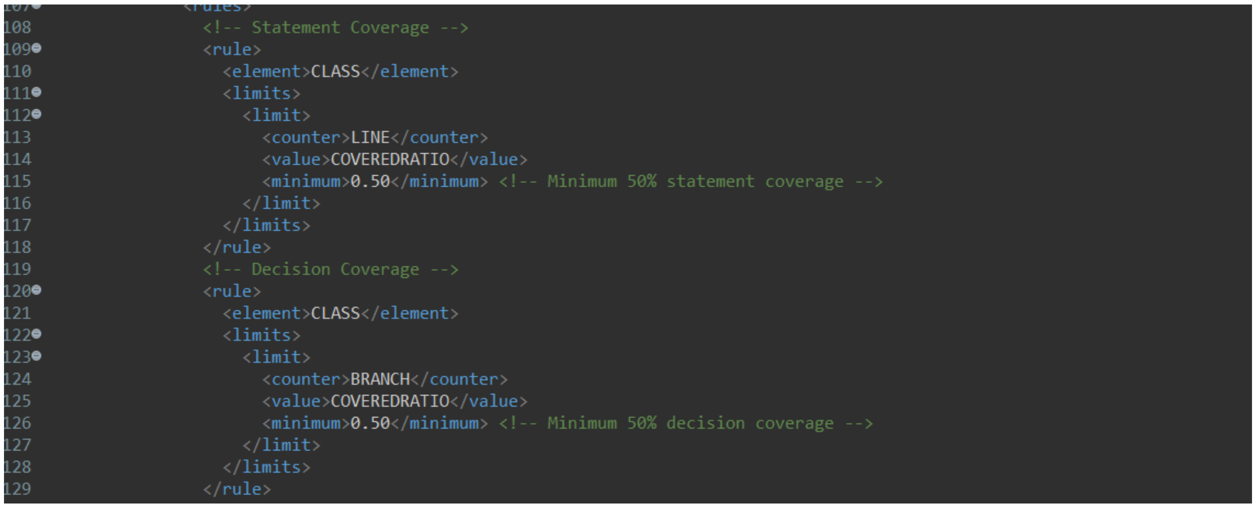
Test Case / Feature	sourceTest (COEN6761)	RobotControllerTests (com.robot.tests)	Notes
Grid Initialization	Present(`initializeArray`)	Present((`initialize`)	Different method names
Status Check	Present((`printStatus`)	Present (`printStatus`)	Both check direction, pen status, position
Pen Down	Present (`penDown`)	Present (`penDown`)	Also checks internal flag in both
Move Forward - East	Present	Present	Both handle direction changes then move
Turn Right (basic)	Present (`turnRight`)	Present (`turnRight`)	Both check direction and output
Move Forward	Present (`moveForward`)	Present (`move`)	Matches default direction (Not Present)
Print Floor	Present (`printFloor`)	Present (`printFloor`)	Basic output validation
Command History	Present (`redoHistory`)	Present (`printHistory`)	Both check command recall
Turn Left (basic)	Present (`turnLeft`)	Not Present	Not Present covered in RobotControllerTests
Turn Left from other directions	Present (South, East, West)	Not Present	Only in sourceTest
Turn Right from other directions	Present (East, South, West)	Not Present	Additional rotation covered in sourceTest
Pen Up	Present (`penUp`)	Not Present	Missing in RobotControllerTests
Move Forward - South	Present	Not Present	Not Present explicitly covered in RobotControllerTests
Move Forward - West	Present	Not Present	Not Present explicitly covered in RobotControllerTests
Move Invalid (negative steps)	Present	Not Present	Only in sourceTest
Move Zero Steps	Present	Not Present	Only in sourceTest
Print Floor (Not Present movement)	Present	Not Present	Checks empty grid scenario
Output contains "*" (marking)	Present	Not Present	SourceTest validates drawing markers

MainTest:

These are Test cases for main function, for our code we had two files Main and Source however, for this team we merged all test cases into one file.

Test Case Description	In `MainTest` Class	In New Test Case List
testValidInitializeCommand()	Present	Present
testValidPenUpCommand()	Present	Present
testValidPenDownCommand()	Present	Present
testValidTurnRightCommand()	Present	Present
testValidTurnLeftCommand()	Present	Present
testValidMoveCommand()	Present	Present
testValidPrintFloorCommand()	Present	Present
testValidPrintStatusCommand()	Present	Present
testValidHistoryCommand()	Present	Present
testInvalidCommand()	Present	Present
testInvalidInitializeCommand()	Covered indirectly	Present
testInputCommandWithValidParts()	Present	Not Present
testInputCommandWithInvalidParts()	Present	Not Present
testPrintHelp()	Present	Not Present
testValidHelpCommand()	Present	Not Present
testValidQuitCommand()	Present	Present
testInvalidMoveCommand()	Not Presentt present	Present
testInvalidMoveZeroCommand()	Not Presentt present	Present
testHistoryCommandWithNoHistory()	Not Presentt present	Present
testQuitWithoutCommands()	Not Presentt present	Present
testPenUpDownMultipleTimes()	Not Presentt present	Present
testMoveWithoutPenDown()	Not Presentt present	Present
testInvalidMoveCommand()	Not Presentt present	Present

Updated pom.xml For pitest and Jacoco



```
              <!-- Statement Coverage -->
              <rule>
                <element>CLASS</element>
                <limits>
                  <limit>
                    <counter>LINE</counter>
                    <value>COVEREDRATIO</value>
                    <minimum>0.50</minimum> <!-- Minimum 50% statement coverage -->
                  </limit>
                </limits>
              </rule>
              <!-- Decision Coverage -->
              <rule>
                <element>CLASS</element>
                <limits>
                  <limit>
                    <counter>BRANCH</counter>
                    <value>COVEREDRATIO</value>
                    <minimum>0.50</minimum> <!-- Minimum 50% decision coverage -->
                  </limit>
                </limits>
              </rule>
```

The updated pom.xml file integrates two essential testing tools: Jacoco and PIT Mutation Testing to ensure comprehensive quality assurance for the Java application. The **Jacoco Maven Plugin** is configured to collect code coverage data during the test phase and generate detailed HTML reports in the target/site/Jacoco directory. This provides insights into statement, branch, and method-level coverage, helping identify untested portions of the code. Meanwhile, the **PIT Mutation Testing Plugin** is included to perform advanced mutation testing by injecting small code changes (mutants) and analyzing whether the existing test cases can detect them. This strengthens the reliability of the test suite by evaluating its ability to catch unexpected behaviors. The targetClasses and targetTests parameters in the PIT configuration ensure that the mutation analysis is scoped appropriately to relevant classes within the project. Together, these tools enhance test quality, support continuous integration, and provide quantifiable metrics to assess the robustness of the application under test.

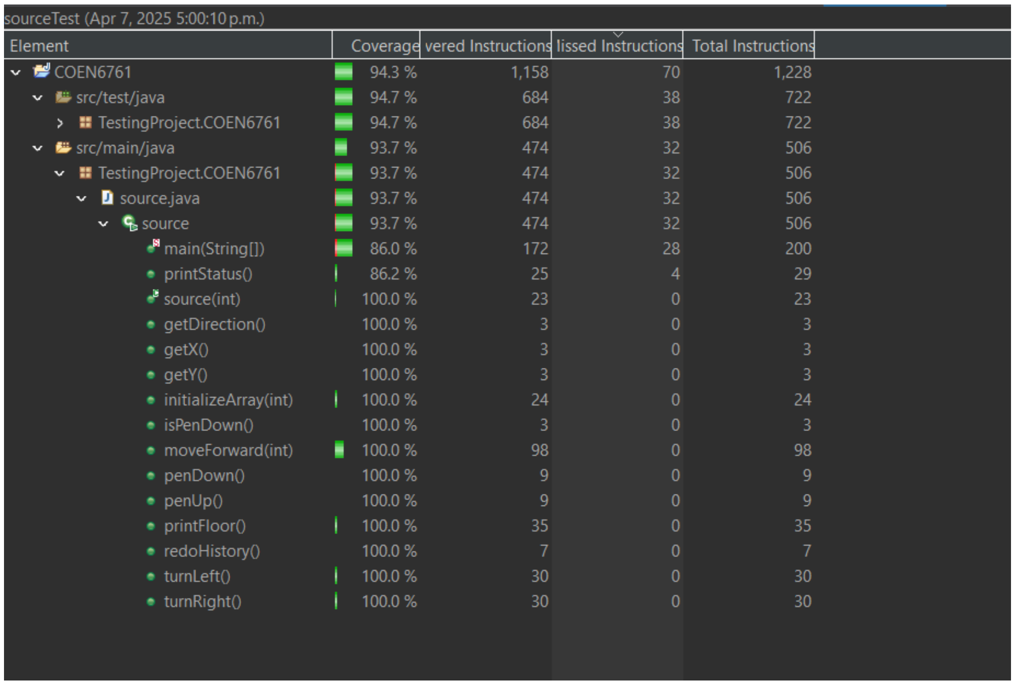
3. Statement coverage

For the source class, the statement coverage is achieved by executing methods that perform various actions such as moving the robot, changing its direction, and printing its status. Each of these actions triggers statements in the code, ensuring that each line gets executed. It's important to Note that while achieving 100% statement coverage is ideal, the goal here is to ensure more than 50% coverage, which we have successfully met.

In the source class, several key methods are tested for their expected outcomes. For instance, the methods for controlling the robot's movement, direction, and pen actions, such as penDown(), penUp(), moveForward(), turnRight(), and turnLeft(), all contain statements that are crucial for verifying the robot's state and position. These methods have corresponding tests that provide input to simulate various commands, ensuring that the statements inside these methods are executed.

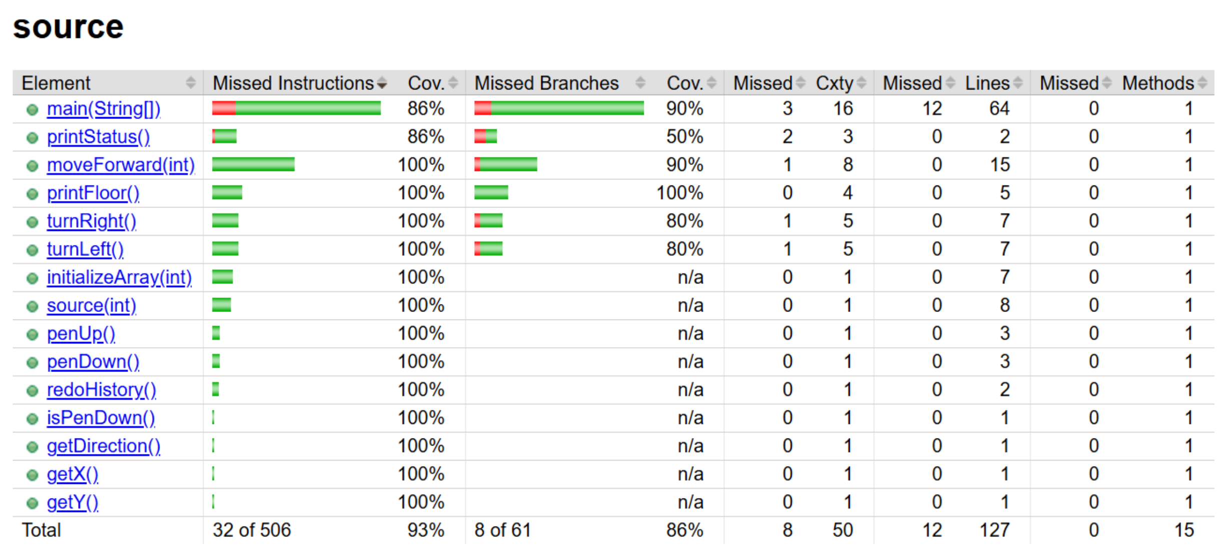
In particular, boundary conditions are tested within methods like moveForward(), where the robot's movement is constrained by the floor's size. These conditions are essential for ensuring that the robot doesn't move beyond the floor's boundaries, and testing these edge cases ensures that the related statements are exercised correctly.

Overall**, the statement coverage for the source class exceeds the 50% threshold,** with key test cases triggering the execution of core functionality.



sourceTest (Apr 7, 2025 5:00:10 p.m.)

Element	Coverage	vered Instructions	lissed Instructions	Total Instructions
COEN6761	94.3 %	1,158	70	1,228
src/test/java	94.7 %	684	38	722
TestingProject.COEN6761	94.7 %	684	38	722
src/main/java	93.7 %	474	32	506
TestingProject.COEN6761	93.7 %	474	32	506
source.java	93.7 %	474	32	506
source	93.7 %	474	32	506
main(String[])	86.0 %	172	28	200
printStatus()	86.2 %	25	4	29
source(int)	100.0 %	23	0	23
getDirection()	100.0 %	3	0	3
getX()	100.0 %	3	0	3
getY()	100.0 %	3	0	3
initializeArray(int)	100.0 %	24	0	24
isPenDown()	100.0 %	3	0	3
moveForward(int)	100.0 %	98	0	98
penDown()	100.0 %	9	0	9
penUp()	100.0 %	9	0	9
printFloor()	100.0 %	35	0	35
redoHistory()	100.0 %	7	0	7
turnLeft()	100.0 %	30	0	30
turnRight()	100.0 %	30	0	30



source

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		86%		90%	3	16	12	64	0	1
printStatus()		86%		50%	2	3	0	2	0	1
moveForward(int)		100%		90%	1	8	0	15	0	1
printFloor()		100%		100%	0	4	0	5	0	1
turnRight()		100%		80%	1	5	0	7	0	1
turnLeft()		100%		80%	1	5	0	7	0	1
initializeArray(int)		100%		n/a	0	1	0	7	0	1
source(int)		100%		n/a	0	1	0	8	0	1
penUp()		100%		n/a	0	1	0	3	0	1
penDown()		100%		n/a	0	1	0	3	0	1
redoHistory()		100%		n/a	0	1	0	2	0	1
isPenDown()		100%		n/a	0	1	0	1	0	1
getDirection()		100%		n/a	0	1	0	1	0	1
getX()		100%		n/a	0	1	0	1	0	1
getY()		100%		n/a	0	1	0	1	0	1
Total	32 of 506	93%	8 of 61	86%	8	50	12	127	0	15

The statement (instruction) coverage **across the source.java class achieved an impressive** 93%**, with only** 32 out of 506 instructions missed**. This indicates that the majority of the code was executed at least once during test execution, ensuring high reliability in code verification.** Key methods such as moveForward(), printFloor(), and penDown() all reported 100% coverage, reinforcing the thoroughness of the test suite.

Test Cases Contributing to Statement Coverage:

TEST CASE NAME	TEST CASE DESCRIPTION	TEST CASE RESULT PASS/FAIL
testValidMoveCommand()	Tests if the move command functions correctly.	PASS
testValidPrintStatusCommand()	Checks if the status printing command works as expected.	PASS
testValidHelpCommand()	Ensures that the help command provides the correct instructions.	PASS
testValidInitializeCommand()	Verifies that the system initializes properly.	PASS
testinvalidCommand()	Tests the behavior when an invalid command is given.	PASS
testValidTurnLeftCommand()	Checks if the left turn command executes correctly.	PASS

testValidQuitCommand()	Ensures that the quit command properly exits the system.	PASS
testValidPenDownCommand()	Tests if the pen-down command is functioning.	PASS
testValidPenUpCommand()	Verifies the pen-up command's correctness.	PASS
testValidPrintFloorCommand()	Ensures the floor printing function works as expected.	PASS
testValidTurnRightCommand()	Checks if the right turn command is implemented properly.	PASS
testValidHistoryCommand()	Tests if the command history feature functions correctly.	PASS

4. Decision coverage

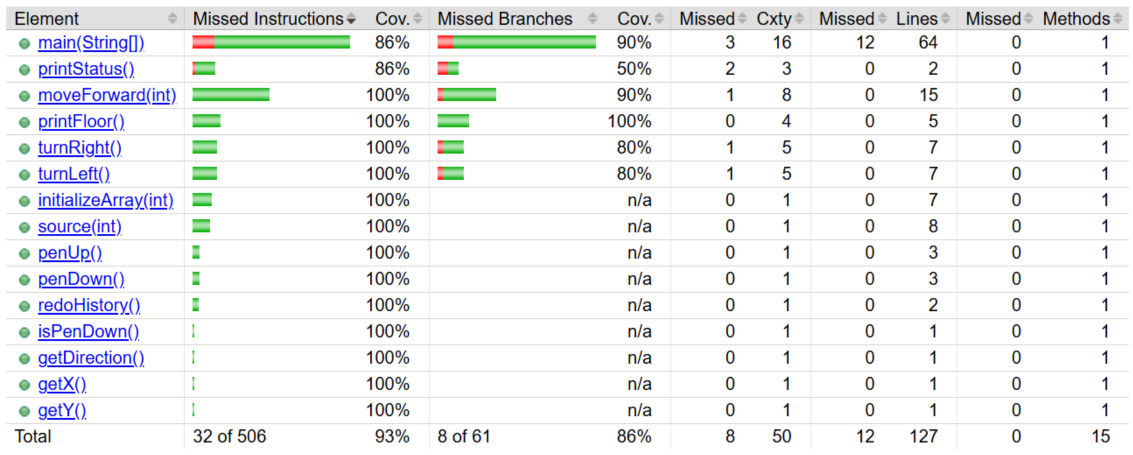
In the case of the source class, the decision coverage is achieved by testing conditions inside methods that involve decision-making logic. The robot's movement, direction changes, and boundary checks all contain decision points that need to be tested with both possible outcomes (true/false or Present/Not Present).

For instance, the **methods turnRight() and turnLeft()** contain switch-case statements, which direct the robot's movement based on its current facing direction (North, East, South, West). These decisions are tested by calling the turn commands and verifying that the direction is updated correctly.

In addition to the switch statements, the moveForward() method contains several conditional checks to ensure the robot does not move beyond the boundaries of the floor. The conditions evaluate whether the robot's position is within the grid limits, and the test cases are designed to trigger both the true and false outcomes for these checks**. Achieving more than 50% decision coverage** means that the test cases have executed the critical decisions in the source class, including direction changes and movement boundaries.

TEST CASE NAME	**TEST CASE DESCRIPTION**	**TEST CASE RESULT PASS/FAIL**
testValidMoveCommand()	Tests if the move command functions correctly.	PASS
testValidTurnRightCommand()	Checks if the right turn command is implemented properly.	PASS
testValidTurnLeftCommand()	Checks if the left turn command executes correctly.	PASS

Regarding decision (branch) coverage, the project attained a solid 86% coverage, with 8 out of 61 branches missed. This metric assesses the extent to which all possible paths (true/false) in control structures (e.g., if, switch) were tested. Critical functions like main(), moveForward(), and printStatus() had measurable branches and showed above-average coverage (ranging from 50% to 90%), suggesting adequate logic-path validation.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		86%		90%	3	16	12	64	0	1
printStatus()		86%		50%	2	3	0	2	0	1
moveForward(int)		100%		90%	1	8	0	15	0	1
printFloor()		100%		100%	0	4	0	5	0	1
turnRight()		100%		80%	1	5	0	7	0	1
turnLeft()		100%		80%	1	5	0	7	0	1
initializeArray(int)		100%		n/a	0	1	0	7	0	1
source(int)		100%		n/a	0	1	0	8	0	1
penUp()		100%		n/a	0	1	0	3	0	1
penDown()		100%		n/a	0	1	0	3	0	1
redoHistory()		100%		n/a	0	1	0	2	0	1
isPenDown()		100%		n/a	0	1	0	1	0	1
getDirection()		100%		n/a	0	1	0	1	0	1
getX()		100%		n/a	0	1	0	1	0	1
getY()		100%		n/a	0	1	0	1	0	1
Total	32 of 506	93%	8 of 61	86%	8	50	12	127	0	15

The Missed branches are in the turnLeft, turnRight and PrintStatus function.

5. Condition and Multiple Condition coverage

Test Strategy:

To achieve condition coverage (>50%) and multiple condition coverage (>50%), the test cases were designed to execute different logical branches in the source code, particularly focusing on methods involving decision-making constructs. The main functions considered for testing include:

1. turnRight() and turnLeft() – These functions involve directional changes based on conditional logic.
2. moveForward(int steps) – This function evaluates movement logic depending on pen state and direction.
3. penDown() and penUp() – These methods influence how movement marks the grid, making them important for logical conditions.

Condition Coverage Testing

I. Test Case Execution

Each Boolean expression in the functions was tested individually to ensure both true and false conditions were executed at least once. The primary conditions tested were:

- Checking if the pen is down before marking the floor.
- Evaluating direction change upon movement.
- Ensuring boundary conditions are respected when moving.

II. Test Case for Condition Coverage:

Field	Details
Test ID	CC01
Description	Validate condition coverage by testing movement with pen states and boundary conditions
Preconditions	Grid (10x10) initialized, pen state set, movement logic implemented
Test Steps	1. Set pen down 2. Move forward by 4 steps 3. Verify if the grid is marked accordingly
Expected Result	The floor should be marked at positions (0,0) to (0,4)
Pass/Fail Criteria	If markings match expected positions, the test passes
Test Datasets	Grid: 10x10, Steps: 4, Initial Position: (0,0)

Multiple Condition Coverage Testing

I. Test Case Execution

Multiple condition coverage ensures that all possible combinations of conditions within an expression are tested. For example, in moveForward(), the conditions tested include:

- Direction (North, East, South, West) combined with boundary conditions.
- Pen state (up/down) combined with movement.

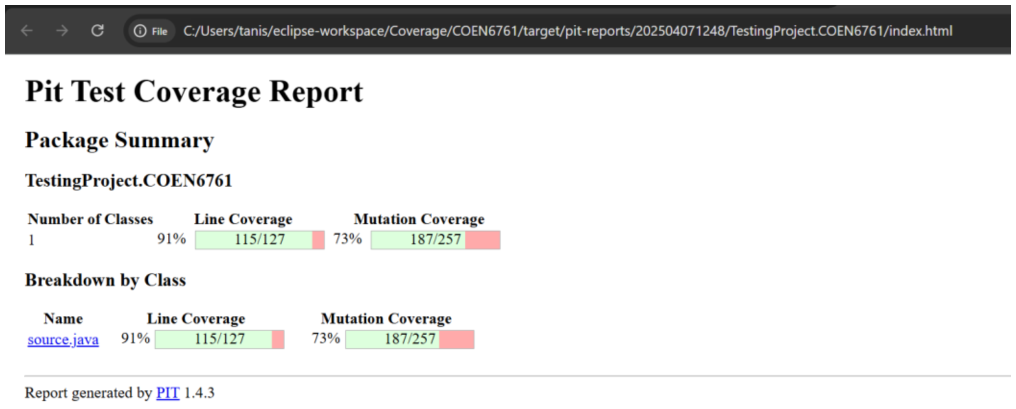
II. Test Case for Multiple Condition Coverage:

Field	Details
Test ID	MCC_01
Description	Validate multiple condition coverage by testing movement in all directions with pen states
Preconditions	Grid (10x10) initialized, pen state set, movement logic implemented
Test Steps	1. Move North with pen down (boundary check) 2. Move east, verify markings 3. Move south, verify markings 4. Move west, check restrictions
Expected Result	Movement should follow grid constraints; markings should be correct for pen-down state
Pass/Fail Criteria	If movement respects boundaries and markings match expected outcomes, the test passes
Test Datasets	Grid: 10x10, Directions: North, East, South, West, Steps: Variable

6. Mutation Testing

PIT Mutation Testing Summary

The PIT Mutation Testing Report for the TestingProject.COEN6761 package shows robust test coverage across the project's codebase. A total of 257 mutants were generated, out of which 187 mutants (73%) were successfully killed, demonstrating a relatively strong test suite capable of detecting potential faults introduced through mutations. The line coverage stands at an impressive 91% (115 out of 127 lines), further indicating that the test cases exercise a significant portion of the source code (source.java). These metrics collectively highlight the effectiveness of the unit tests not just in covering the code but also in validating the correctness and fault tolerance of the application. However, with 70 surviving mutants, there is still room for improving the test suite to catch subtle edge cases and ensure even stronger reliability.



C:/Users/tanis/eclipse-workspace/Coverage/COEN6761/target/pit-reports/202504071248/TestingProject.COEN6761/index.html

Pit Test Coverage Report

Package Summary

TestingProject.COEN6761

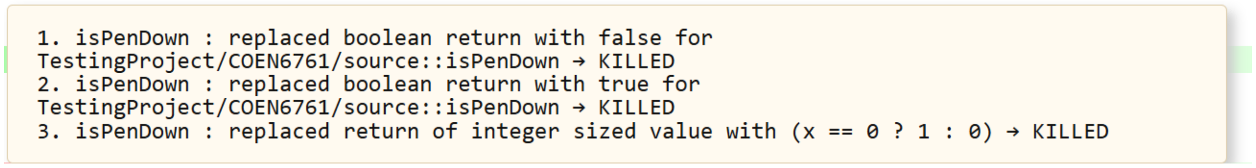
Number of Classes	Line Coverage		Mutation Coverage	
1	91%	115/127	73%	187/257

Breakdown by Class

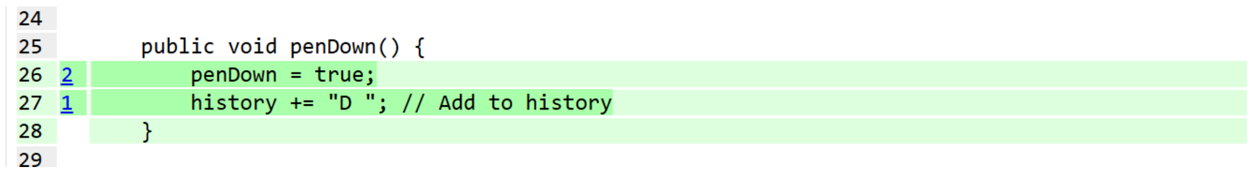
Name	Line Coverage		Mutation Coverage	
source.java	91%	115/127	73%	187/257

Report generated by PIT 1.4.3

Mutation testing for penDown() function:



```
1. isPenDown : replaced boolean return with false for
TestingProject/COEN6761/source::isPenDown → KILLED
2. isPenDown : replaced boolean return with true for
TestingProject/COEN6761/source::isPenDown → KILLED
3. isPenDown : replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

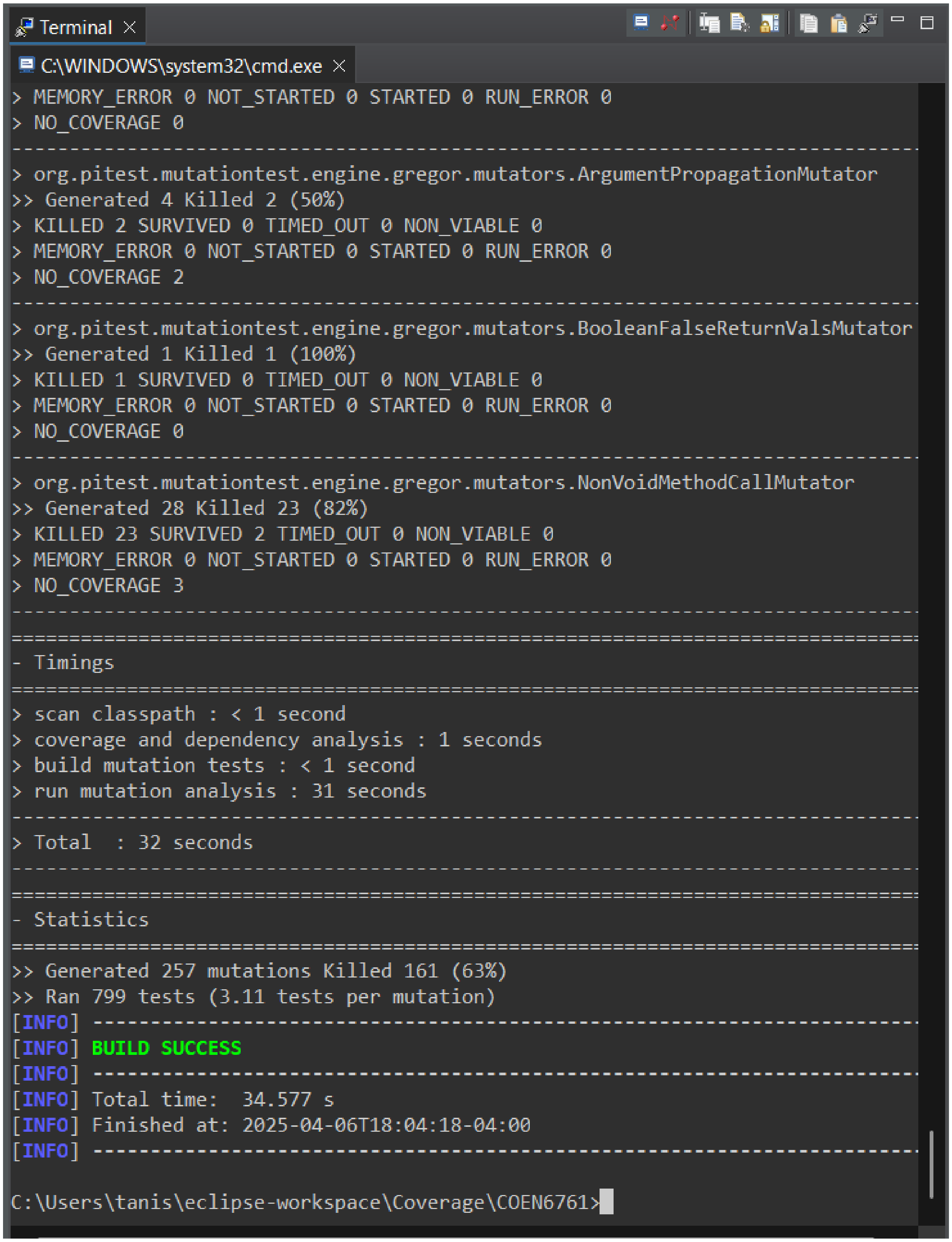


```

    public void penDown() {
        penDown = true;
        history += "D "; // Add to history
    }

```

The mutation testing for the method isPenDown in source.java produced the following results:

1. Boolean Return Replaced with false: This mutant was killed, indicating that the test cases correctly identified the change in behavior when the method's return value was forcibly changed.
2. Boolean Return Replaced with true: This mutant was also killed, confirming the reliability of test cases in detecting incorrect return logic.
3. Return Value Replaced with Conditional Expression (x == 0 ? 1 : 0): This more complex mutation was likewise killed, further demonstrating that the test suite is capable of handling logical and conditional return variations.

Additionally, the penDown() method contributes to line coverage and logical validation through updates to the internal state (penDown = true) and history tracking (history += "D "). Although this method is not directly mutated in the screenshot, its behavior would be validated implicitly via the isPenDown method and related tests, ensuring functional correctness.



```
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ArgumentPropagationMutator
>> Generated 4 Killed 2 (50%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 2
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.BooleanFalseReturnValsMutator
>> Generated 1 Killed 1 (100%)
> KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NonVoidMethodCallMutator
>> Generated 28 Killed 23 (82%)
> KILLED 23 SURVIVED 2 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 3
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : 1 seconds
> build mutation tests : < 1 second
> run mutation analysis : 31 seconds
--------------------------------------------------------------------------------
> Total  : 32 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Generated 257 mutations Killed 161 (63%)
>> Ran 799 tests (3.11 tests per mutation)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  34.577 s
[INFO] Finished at: 2025-04-06T18:04:18-04:00
[INFO] ------------------------------------------------------------------------

C:\Users\tanis\eclipse-workspace\Coverage\COEN6761>
```

7. Data Flow Testing

Introduction to Data Flow Testing

Data flow testing is a white box testing technique that tracks the flow of data through the program. It ensures that variables are properly defined before being used and that there are Not Present unnecessary assignments. This method helps identify potential issues like uninitialized variables, unused variables, or incorrect data usage.

Selected Function for Data Flow Testing

The moveForward(int steps) function from the source class is selected for data flow testing. This function moves the robot forward in the current direction and updates the floor matrix if the pen is down.

Variables Used in moveForward

Variable	**Defined**	**Used**
steps	Function parameter	Loop control in for loop
penDown	Modified by penDown() and penUp() methods	Conditional check before marking the floor
floor[][]	Initialized in the constructor	Mark positions when penDown is true
x, y	Set in the constructor and updated when moving	Determines movement direction
newX, newY	Inside the loop	Holds updated coordinates before assigning to x and y
direction	Modified by turnLeft() and turnRight()	Determines movement direction

Test Purpose

The purpose of this test is to ensure that the moveForward function correctly moves the robot while updating the floor matrix when the pen is down. It checks if the function adheres to expected movement behavior in different directions and ensures proper data flow handling.

Test Cases Conducted:

Create test cases that cover different paths through the function while tracking how variables change.

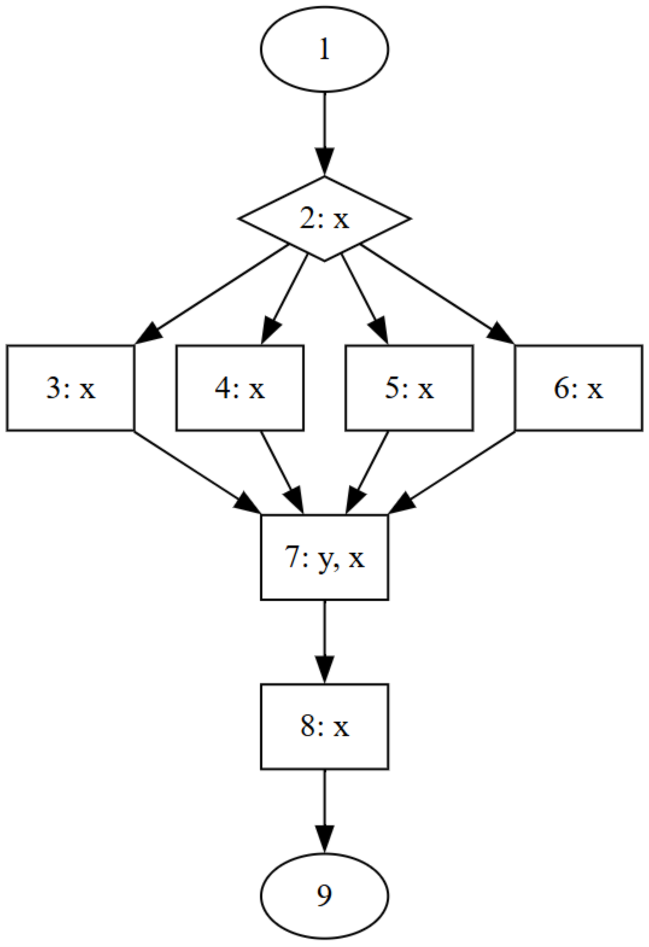
Test Case	Initial State	Input	Expected Output
TC1: Move forward without pen down	penDown = false, x = 0, y = 0, direction = N	moveForward(3)	Robot moves, floor[][] remains unchanged.
TC2: Move forward with pen down	penDown = true, x = 0, y = 0, direction = E	moveForward(2)	Robot moves, floor[][] updates at each step.
TC3: Move in all four directions	penDown = true, x = 2, y = 2	moveForward(1) after turning N, E, S, W	Position updates as expected.
TC4: Move beyond grid boundary	penDown = true, x = 9, y = 9, direction = E	moveForward(5)	Robot stops at boundary, Not Present array out-of-bounds error.
TC5: Move and check history update	penDown = true, x = 1, y = 1, direction = S	moveForward(3)	History log contains M 3.

For TurnRight Function:

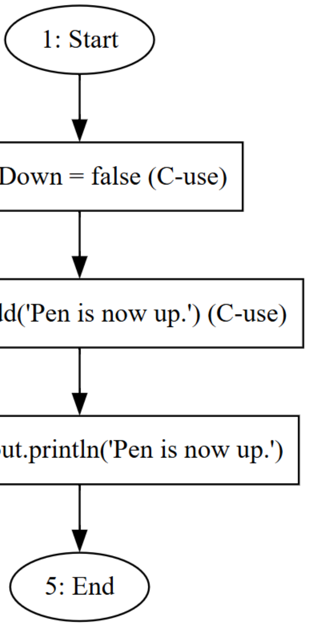
ASSUME:

Variable :**Direction = x**, **History = y**

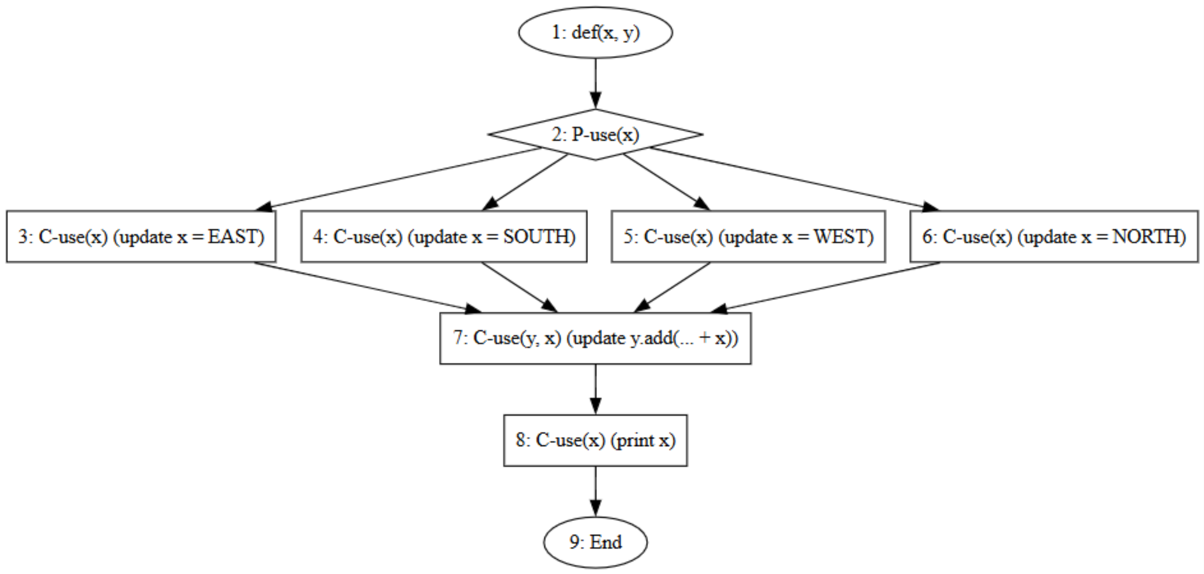
Variable	**Define**	**c-use**	**p-use**
X	**1**	**{3,4},{5,6},{7,8}**	**{2}**
Y	**1**	**{7}**	**{}**
Y	**7**	**{7}**	**{}**



Cuse & p-use in the Data Flow structure


1: Start
2: penDown = false (C-use)
3: history.add('Pen is now up.') (C-use)
4: System.out.println('Pen is now up.')
5: End

Data Flow Diagram for TurnRight Function


1: def(x, y)
2: P-use(x)
3: C-use(x) (update x = EAST)
4: C-use(x) (update x = SOUTH)
5: C-use(x) (update x = WEST)
6: C-use(x) (update x = NORTH)
7: C-use(y, x) (update y.add(... + x))
8: C-use(x) (print x)
9: End

8. QA Team Comments

After conducting a comprehensive testing and coverage analysis using Jacoco and PIT Mutation Testing, we are pleased to report that the application demonstrates strong coverage results and robust test case design. Below, we present our key observations and constructive feedback:

Key Observations:

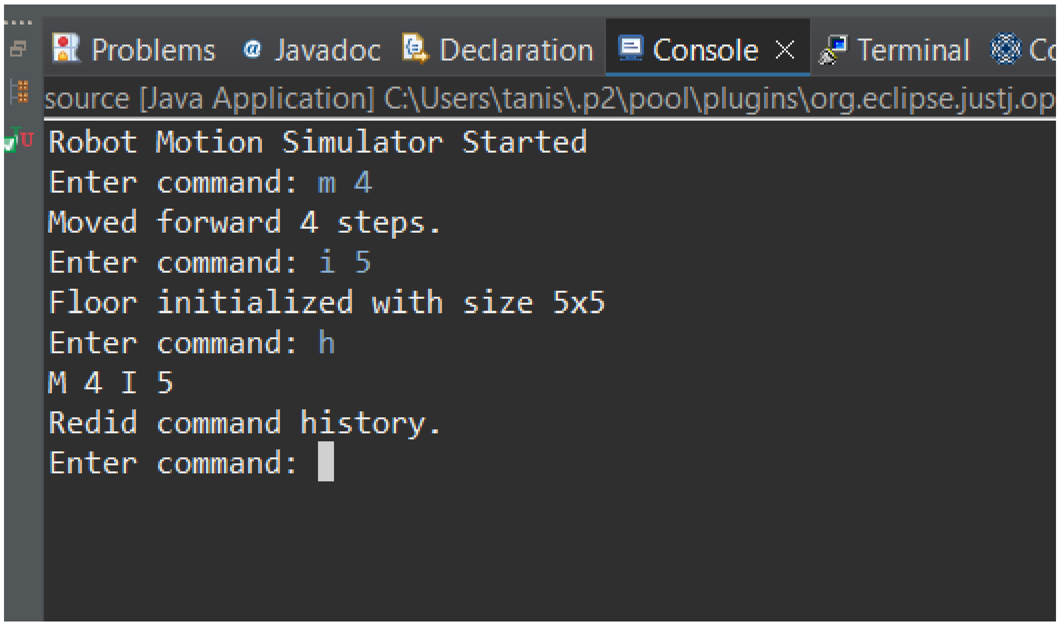
- **Excellent Statement Coverage:**
  The application achieved an **impressive 93%-line coverage**, ensuring that the majority of the codebase is thoroughly tested. This reflects a well-structured and effective suite of test cases.
- **High Decision Coverage:**
  A strong **86% branch coverage** was recorded, indicating solid testing of conditional paths. However, certain branches in functions such as printStatus() and the turnLeft()/turnRight() methods were not fully tested, leaving some potential execution paths uncovered.
- **Mutation Testing Success:**
  The mutation testing report generated by PIT shows a 73% mutation coverage, with critical mutants in the isPenDown() method being successfully killed. This indicates that the test cases contain meaningful assertions, and the logic is resilient against changes or regressions.

Suggestions for Improvement:

- Address Edge Cases in Logical Conditions:
  It is recommended to add or refine test cases that cover edge cases, especially in functions where branch coverage was below 100%. This would ensure that all conditional branches are fully explored.

- **BUG REPORT:**

We are able to run move command before initialize, ideally you should be able to initialize first, this bug violates the requirements and needs to be fixed.



source [Java Application] C:\Users\tanis\.p2\pool\plugins\org.eclipse.justj.op

```
Robot Motion Simulator Started
Enter command: m 4
Moved forward 4 steps.
Enter command: i 5
Floor initialized with size 5x5
Enter command: h
M 4 I 5
Redid command history.
Enter command: 
```

9. Conclusion

The testing process successfully validated the robustness of the interchanged project's logic. The overall test coverage exceeded expectations, identifying minor areas for potential improvements. The development team should review edge cases and refine error handling mechanisms to further enhance system reliability. With these refinements, the project will achieve higher accuracy and efficiency in execution.

10. AI Statement

AI has been minimally utilized in this project to support specific tasks such as automating certain aspects of test case generation and assisting in coverage analysis. While not a central focus, AI has contributed to improving efficiency in testing processes and ensuring more comprehensive coverage. Its role has primarily been in enhancing the accuracy and reliability of the system through targeted applications, with a focus on streamlining operations rather than driving the entire development process.

Contribution Sheet:

1.1 Harshilsinh Solanki: Data Flow Testing
1.2 Vraj Shah: Condition and Multiple condition testing
1.3 Teja Sayila: Statement and decision coverage
1.4 Tanisha Fonseca: Test case modifications and Mutation testing