PROJECT ROBOT MOTION - TEST REPORT

Task 3: Black-Box and White-Box Testing

COEN 6761 - Software Testing and Validation

Team Code Blooded

Rajat Rajat - 40160245
Sharul Dhiman - 40195730
Rohan Kodavalla - 40196377
Rishi Murugesan Gopalakrishnan - 40200594

Department of Electrical and Computer Engineering

Gina Cody School of Engineering and Computer Science

Winter 2023

TEST PLAN

APPLICATION UNDER TEST : FLOOR ROBOT
PREPARED BY : Development Team 6
Mayank Gupta
Krishna Bhatt
Uday Putreddy

Introduction

The application being tested here is **floor robot**. The application simulates a robot that can walk around a room and the robot is operated using a Java program. The robot is equipped with a pen and the pen can be held either in an upward position or downward position. The robot can trace the path as it moves. When the pen is up, the robot can move freely without tracing the path, whereas when the pen is down it will trace the path as it moves. The floor is represented using an N by N array which is initially set to all zeros.
Initially, the floor values are set to zero and the robot starts at position[0,0] with the pen facing up and the robot facing north. The robot receives commands from the user and the following are the set of commands.

Command	Use
I n | i n	Initializes the n x n array floor, n >0. This command will also set the robot's initial position to (0,0) and will set the pen's position to up and the robot's direction as north.
U | u	Pen Up
D | d	Pen Down
R | r	Turn Right
L | l	Turn Left
M s | m s	This command is to make the robot move forward 's' spaces where 's' is a non-negative number.
P | p	Prints the NxN array and displays the indices in the console.
C | c	This command will print the current position of the

	pen and whether it is up or down and which direction it is facing.
Q | q	Terminate the program

Objectives

- The main objective of this test phase is to evaluate the application by utilizing a range of approaches such as black-box testing, white-box testing or any other relevant techniques.
- Specifically for White-box testing, test cases should be formed in such a way that it provides
  - statement coverage (show percentage covered > 50%)
  - decision coverage (show percentage covered > 50%)
  - condition coverage (show percentage covered > 50%)
  - multiple condition coverage (show percentage covered > 50%)
- Select one function and perform Mutation Testing
- Select one function and perform Data Flow Testing

Testing Strategy

Testing of this application is planned to be done using white-box and black-box testing approach. The major activities include developing test cases using white box and black box approach. This may include applying Boundary value analysis, domain partitioning, choice coverage, data flow testing and mutation testing. All these tasks can be performed using Java IDE of the tester's choice, preferably Eclipse or IntelliJ. Coverage and Mutation testing can be performed using plugins such as EclEmma and PIT Mutation respectively.

Test Schedule

MILESTONES	ESTIMATED TIME
Black - box testing	1 day
White-box testing	1 day
Code Analysis and Coverage	1 day
Mutation and Data Flow Testing	1 day

Features to be Tested

As the QA team we will test the following features from the Robot program. We have mapped the features along with the corresponding functions in Robot.java

Feature	**Method Name**
Read Commands from User	runRobot ( )
Initialize Floor for the robot	initializeFloor ( )
Turn pen down	setPenDown ( )
Turn pen up	setPenUp ( )
Turn Robot Right	turnRight ( )
Turn Robot Left	turnLeft ( )
Display Current Position and Status of Robot	showCurrentPositionStatus ( )
Move the robot in the forward direction	moveForward ( )
Display the floor matrix	displayMatrix ( )
Quit Program	runRobot ( )

Roles and Responsibilities

- Rajat Rajat - Create and code test cases using white-box testing approach
- Sharul Dhiman - Create and code test cases using black-box testing approach
- Rohan Kodavalla - Code Analysis and perform manual code coverage to verify
- Rishi Murugesan Gopalakrishnan - Perform Mutation and Data Flow Testing

Tools Used

- Eclipse Java IDE
- EclEmma plugin in Eclipse
- IntelliJIDE
- draw.io
- Google Docs

Approvals

Rajat Rajat

Rohan Kodavalla

Sharul Dhiman

Rishi Murugesan Gopalakrishnan

Black Box Testing

In black box testing, we treated the software as a "black box" and tests it based on the inputs and expected outputs, without having knowledge of its internal structure, design, or implementation details. We designed test cases based on the application requirements and tests the software to verify that it meets the specified requirements. The goal of black box testing is to identify defects in the software's functionality and ensure that it behaves as expected, without considering its internal workings.

Test case	Invalid Test case values	valid Test case values	Invalid Test case values
Moveforwardtest() (assuming I =10)	(Min Value - 1)= -1	(Min, +Min, Max, -Max)= 1, 2, 9, 8	(Max Value + 1)= 10
initializingFloorTest ()	Min Value-1)= 0	(Min, +Min, Max, -Max)= 1,2,-,-	Max Value + 1)= -

Above table shows the test cases and all the possible values it can take by considering boundary values. So, we have tested it for all the values and we have seen that all the test cases passed in black boxing.

White Box Testing

White box testing is a software testing approach that allows testers to access the internal structure and workings of the software being tested. Unlike black box testing, where the tester only examines the external behavior of the software, white box testing provides the tester with knowledge of the code, algorithms, and data structures used to develop the software. The primary objective of white box testing is to identify and eliminate any defects or bugs that might exist in the software's code.

White box testing can be carried out at various stages of the software development cycle, such as unit testing, integration testing, and system testing. To ensure that every line of code and every possible decision point is tested, white box testing techniques such as statement coverage, branch coverage, path coverage, and condition coverage are often employed.

When combined with other testing techniques such as black box testing, white box testing can help ensure that software is of high quality and free of bugs and errors.

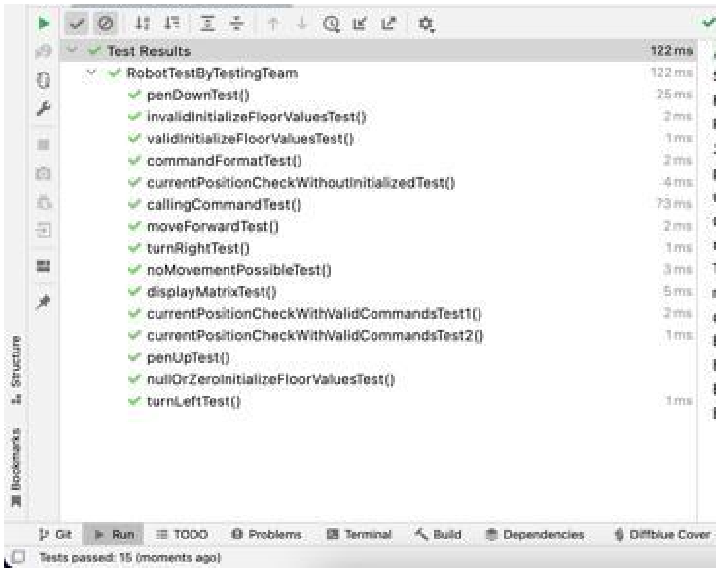
Below is the table for White Box Testcases:

Test case No.	Test case	Expected Test case values	Test case result
1.	currentPositionCheckWithValidCommandsTest1()	Pen Status: Up Direction Facing: North Robot Position: (4,8)	Pass
2.	currentPositionCheckWithValidCommandsTest2()	Pen Status: Up Direction Facing: South Robot Position: (10,25)	Pass
3.	displayMatrixTest()	int[][] floor = {{1,1,1,0},{0,0,1,0},{0,0,1,0},{0,0,1,0}}	Pass
4.	penDownTest()	Pen Status: Down	Pass
5.	penUpTest()	Pen Status: Up	Pass
6.	nullOrZeroInitializeFloorValue	Desired Pen Status: Not	Pass

	sTest()	null Desired Direction: Not null Desired Floor Size: Not Zero	
7.	validInitializeFloorValuesTest()	Desired Pen Status: Up Desired Direction: North Desired Floor Size: 40 Robot Position at X axis: 0 Robot Position at Y axis: 0	Pass
8.	invalidInitializeFloorValuesTest()	Initialize Floor: false	Pass
9.	currentPositionCheckWithoutInitializedTest()	Current Position Status: StringIndexOutOfBoundsException	Pass
10.	noMovementPossibleTest()	Current position same as position before any movement	Pass
11.	callingCommandTest()	NA	Pass
12.	turnRightTest()	Direction After Turning Right: East Direction After Turning Right: South Direction After Turning Right: West Direction After Turning Right: North	Pass
13.	turnLeftTest()	Direction After Turning Right: West Direction After Turning Right: North Direction After Turning Right: East Direction After Turning	Pass

		Right: South	
14.	moveForwardTest()	Current Position and Status: Position: 0, 2 - Pen: down - Facing: north Current Position and Status: Position: 6, 2 - Pen: down - Facing: east Current Position and Status: Position: 6, 6 - Pen: down - Facing: north Current Position and Status: Position: 6, 3 - Pen: down - Facing: south	Pass
15.	commandFormatTest()	invalidIIntegerValue: -1 invalidCommandFormat: -1	Pass

Below is the screenshot for the whitebox testcases:



```
Test Results                                              122 ms
  RobotTestByTestingTeam                                  122 ms
    penDownTest()                                          25 ms
    invalidInitializeFloorValuesTest()                      2 ms
    validInitializeFloorValuesTest()                        1 ms
    commandFormatTest()                                     2 ms
    currentPositionCheckWithoutInitializedTest()            4 ms
    callingCommandTest()                                   73 ms
    moveForwardTest()                                       2 ms
    turnRightTest()                                         1 ms
    noMovementPossibleTest()                                3 ms
    displayMatrixTest()                                     5 ms
    currentPositionCheckWithValidCommandsTest1()            2 ms
    currentPositionCheckWithValidCommandsTest2()            1 ms
    penUpTest()
    nullOrZeroInitializeFloorValuesTest()
    turnLeftTest()                                          1 ms

Tests passed: 15 (moments ago)
```

Code Analysis for White Box Testing

Code coverage is a metric used to measure percentage of code lines, statements, methods, branches or conditions which are executed during testing of an application. It indicates how much of the code has been tested by the test cases and can also help identify parts of the code that weren't tested properly.

For this task below type of code coverage can be done:

- Statement coverage
- Decision coverage
- Condition coverage
- Multiple Condition coverage

Statement Coverage

Statement coverage is a specific type of code coverage that measures the proportion of executable statements in a program that are executed during the execution of a test suite.

To calculate statement coverage, the code is instrumented with special instructions that count the number of times each statement is executed during the running of a test suite. After the test suite has been executed, the coverage tool calculates the percentage of statements that have been executed at least once. It helps to assess the thoroughness of the testing process by indicating which parts of the code have been exercised and which parts have not.

Robot.java methods	**No. of Statements in each method**	**No. of Statements covered by test cases**	**Statement Coverage Percentage**
getfloor()	3	3	100%
splitArray()	46	46	100%
runRobot()	90	90	100 %
initializeFloor()	32	32	100 %
setPenUp()	6	6	100 %

setPenDown()	6	6	100 %
turnRight()	32	32	100 %
turnLeft()	32	32	100 %
showCurrentPositionStatus()	56	56	100 %
moveForward()	263	236	89.7 %
displayMatrix()	114	114	100%
TOTAL	**686**	**659**	**96.1%**

Decision Coverage

Decision coverage is a measure of the degree to which the decision points in a program have been exercised by a test suite. It is a type of code coverage that focuses on evaluating whether every possible outcome of a decision has been tested. In other words, it determines if every branch in a decision structure (such as an if statement, switch statement, or loop) has been executed at least once during testing.

Functions in Robot.java	**Number of decision points**	**Number of decision points covered by test cases**	**Decision Coverage Percentage**
getfloor()	NA	NA	0 %
splitArray()	4	4	100 %
runRobot()	7	7	100 %
initializeFloor()	1	1	100 %
setPenUp()	1	1	100 %
setPenDown()	1	1	100 %

turnRight()	2	2	100 %
turnLeft()	2	2	100 %
showCurrentPositionStatus()	1	1	100 %
moveForward()	5	5	100 %
displayMatrix()	1	1	100 %
TOTAL			**100 %**

Condition & Multiple - Condition Coverage

Condition coverage ensures that every possible condition in the code is evaluated at least once. The aim of this coverage metric is to detect the errors caused by complex logical expressions. It is a useful measure in detecting flaws in decision-making statements such as if-else statements and switch cases.
Below is the table defining the condition coverage:

Robot.java methods	**Conditional Statements Type**	**Number of Conditional statements**	**Number of Multiple conditional statements**	**Number of Conditional statements passed by test cases**	**Conditional Coverage Percentage**
getfloor()	N/A	NA	NA	NA	NA
splitArray()	if-else-if-else-else	3	0	3	100%

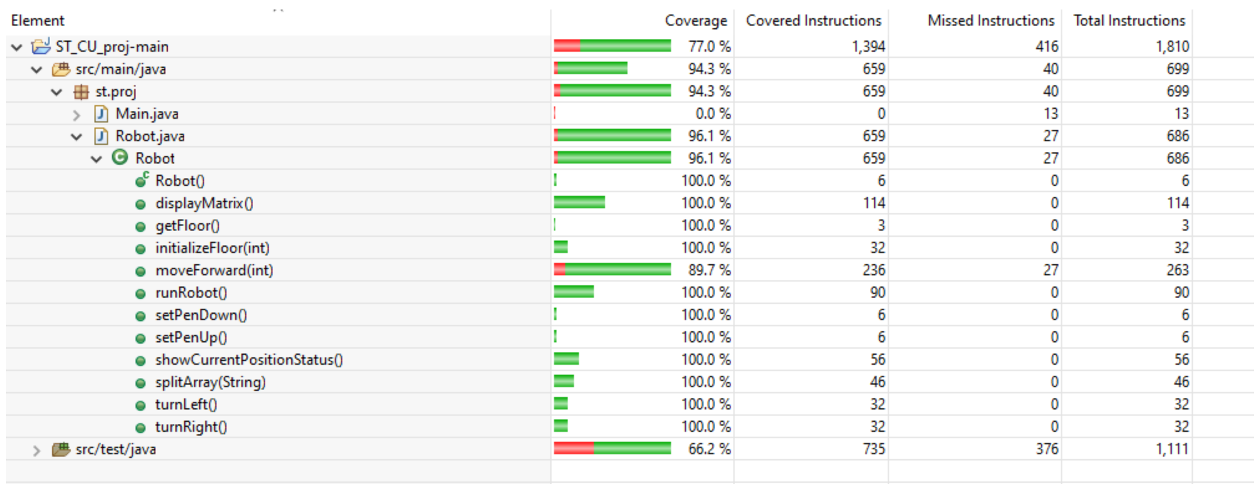
runRobot()	Switch-if, if, while	4	0	4	100%
initializeFloor()	If-else	1	0	1	100%
setPenUp()	N/A	-	-	-	N/A
setPenDown()	N/A	-	-	-	N/A
turnRight()	If, switch	2	0	2	100%
turnLeft()	If, switch	2	0	2	100%
showCurrentPositionStatus()	if	1	0	1	100%
moveForward()	If, switch	18	4	18 (3 conditional & 1 multiple conditional missed)	81.8%
displayMatrix()	if	5	0	5	100%
TOTAL		36	4	33-conditional, 3 multiple conditional PASSED	

Therefore, from above table the condition coverage for the Robot.java code is classified into:

- Condition coverage = 91.6% , meaning every possible condition in the code is evaluated at least once.
- Multiple Condition coverage =75 % , portraying the set of multiple conditions that are evaluated.

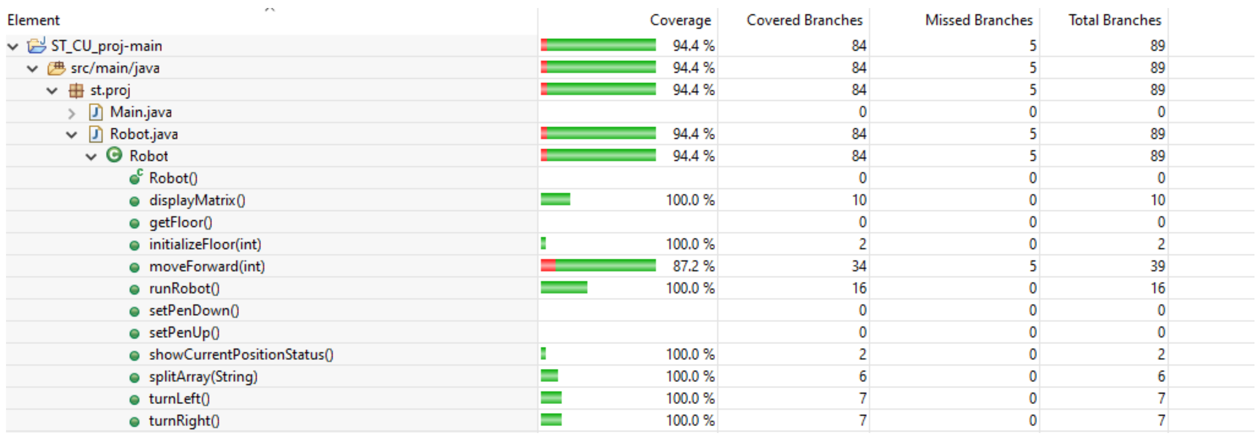
Coverage from IDE Tool

Instruction Counter



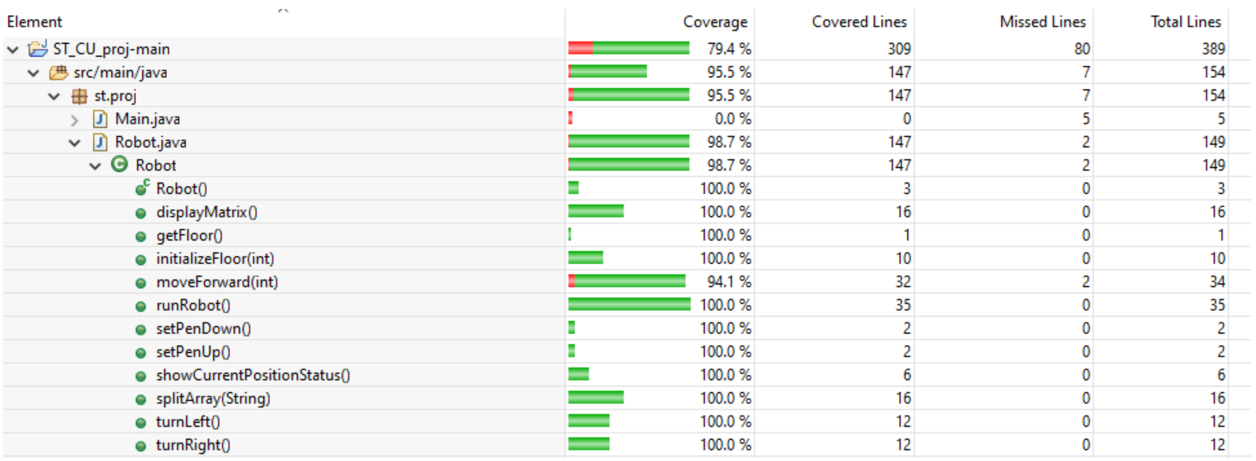
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ST_CU_proj-main	77.0 %	1,394	416	1,810
src/main/java	94.3 %	659	40	699
st.proj	94.3 %	659	40	699
Main.java	0.0 %	0	13	13
Robot.java	96.1 %	659	27	686
Robot	96.1 %	659	27	686
Robot()	100.0 %	6	0	6
displayMatrix()	100.0 %	114	0	114
getFloor()	100.0 %	3	0	3
initializeFloor(int)	100.0 %	32	0	32
moveForward(int)	89.7 %	236	27	263
runRobot()	100.0 %	90	0	90
setPenDown()	100.0 %	6	0	6
setPenUp()	100.0 %	6	0	6
showCurrentPositionStatus()	100.0 %	56	0	56
splitArray(String)	100.0 %	46	0	46
turnLeft()	100.0 %	32	0	32
turnRight()	100.0 %	32	0	32
src/test/java	66.2 %	735	376	1,111

Branch Counter



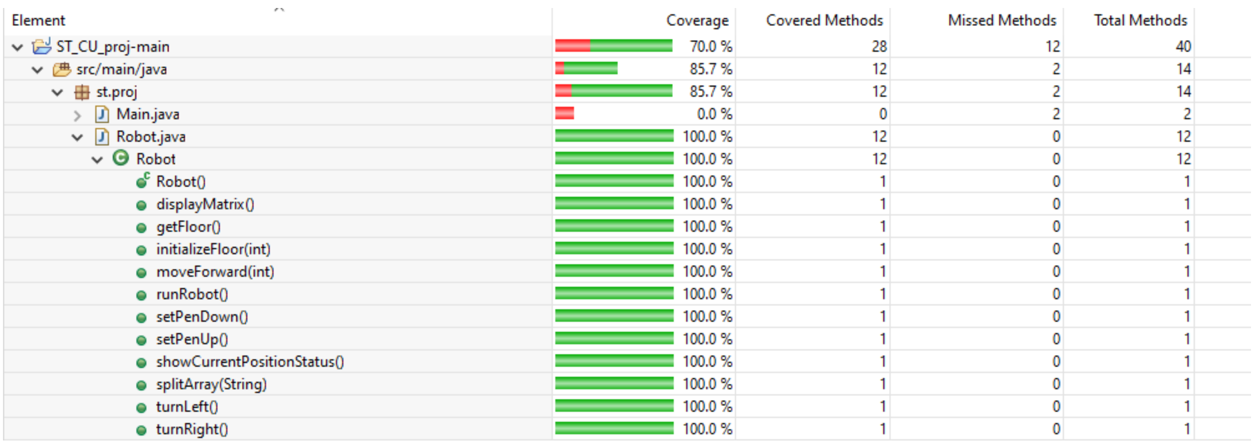
Element	Coverage	Covered Branches	Missed Branches	Total Branches
ST_CU_proj-main	94.4 %	84	5	89
src/main/java	94.4 %	84	5	89
st.proj	94.4 %	84	5	89
Main.java		0	0	0
Robot.java	94.4 %	84	5	89
Robot	94.4 %	84	5	89
Robot()		0	0	0
displayMatrix()	100.0 %	10	0	10
getFloor()		0	0	0
initializeFloor(int)	100.0 %	2	0	2
moveForward(int)	87.2 %	34	5	39
runRobot()	100.0 %	16	0	16
setPenDown()		0	0	0
setPenUp()		0	0	0
showCurrentPositionStatus()	100.0 %	2	0	2
splitArray(String)	100.0 %	6	0	6
turnLeft()	100.0 %	7	0	7
turnRight()	100.0 %	7	0	7

Line Counter



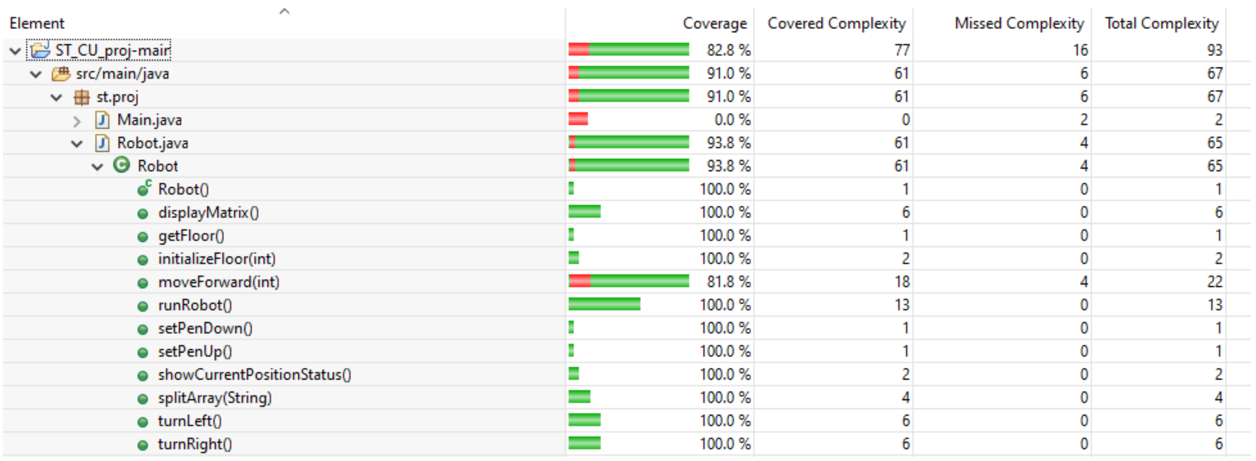
Element	Coverage	Covered Lines	Missed Lines	Total Lines
ST_CU_proj-main	79.4 %	309	80	389
src/main/java	95.5 %	147	7	154
st.proj	95.5 %	147	7	154
Main.java	0.0 %	0	5	5
Robot.java	98.7 %	147	2	149
Robot	98.7 %	147	2	149
Robot()	100.0 %	3	0	3
displayMatrix()	100.0 %	16	0	16
getFloor()	100.0 %	1	0	1
initializeFloor(int)	100.0 %	10	0	10
moveForward(int)	94.1 %	32	2	34
runRobot()	100.0 %	35	0	35
setPenDown()	100.0 %	2	0	2
setPenUp()	100.0 %	2	0	2
showCurrentPositionStatus()	100.0 %	6	0	6
splitArray(String)	100.0 %	16	0	16
turnLeft()	100.0 %	12	0	12
turnRight()	100.0 %	12	0	12

Method Counter



Element	Coverage	Covered Methods	Missed Methods	Total Methods
ST_CU_proj-main	70.0 %	28	12	40
src/main/java	85.7 %	12	2	14
st.proj	85.7 %	12	2	14
Main.java	0.0 %	0	2	2
Robot.java	100.0 %	12	0	12
Robot	100.0 %	12	0	12
Robot()	100.0 %	1	0	1
displayMatrix()	100.0 %	1	0	1
getFloor()	100.0 %	1	0	1
initializeFloor(int)	100.0 %	1	0	1
moveForward(int)	100.0 %	1	0	1
runRobot()	100.0 %	1	0	1
setPenDown()	100.0 %	1	0	1
setPenUp()	100.0 %	1	0	1
showCurrentPositionStatus()	100.0 %	1	0	1
splitArray(String)	100.0 %	1	0	1
turnLeft()	100.0 %	1	0	1
turnRight()	100.0 %	1	0	1

Complexity Coverage



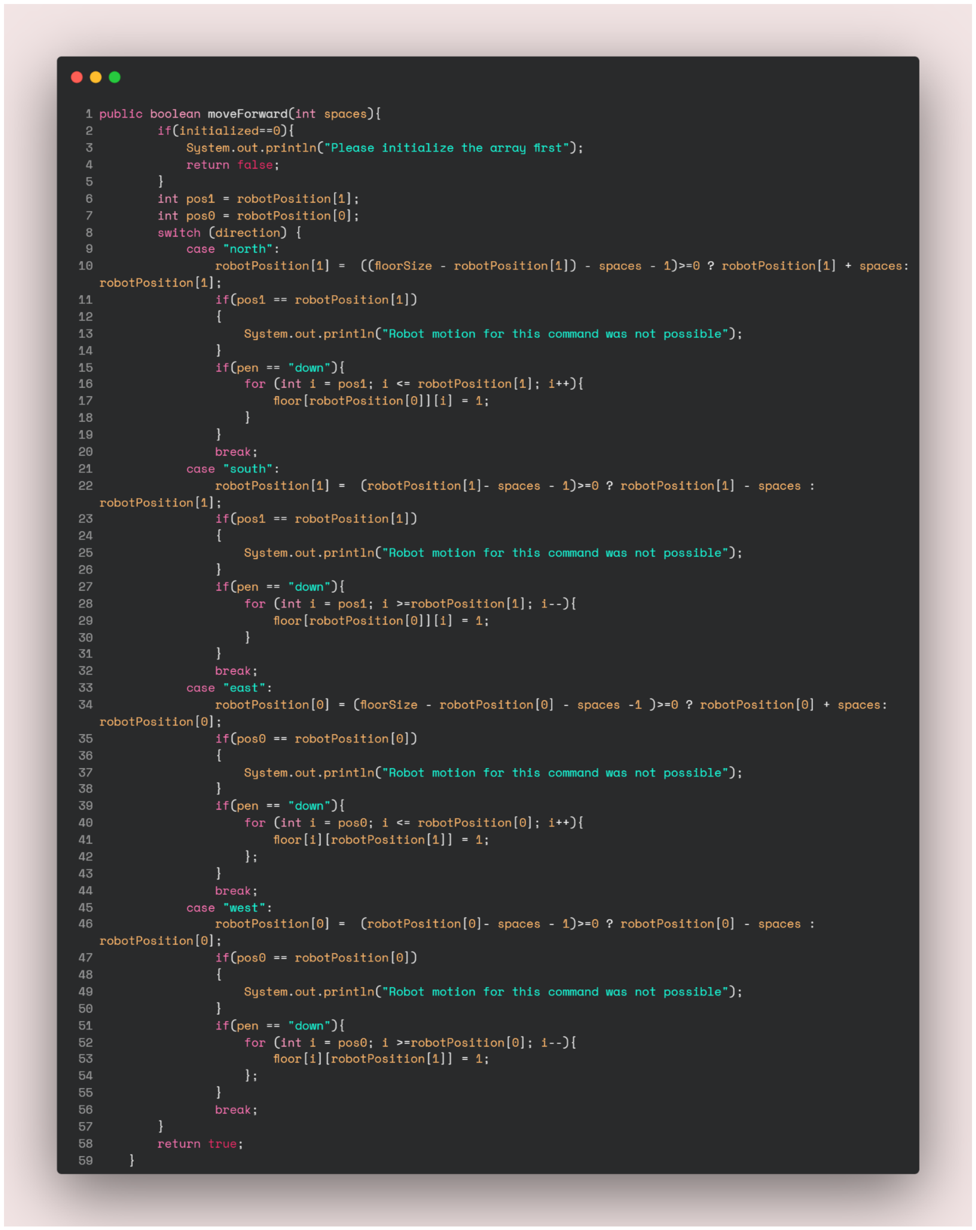
Element	Coverage	Covered Complexity	Missed Complexity	Total Complexity
ST_CU_proj-main	82.8 %	77	16	93
src/main/java	91.0 %	61	6	67
st.proj	91.0 %	61	6	67
Main.java	0.0 %	0	2	2
Robot.java	93.8 %	61	4	65
Robot	93.8 %	61	4	65
Robot()	100.0 %	1	0	1
displayMatrix()	100.0 %	6	0	6
getFloor()	100.0 %	1	0	1
initializeFloor(int)	100.0 %	2	0	2
moveForward(int)	81.8 %	18	4	22
runRobot()	100.0 %	13	0	13
setPenDown()	100.0 %	1	0	1
setPenUp()	100.0 %	1	0	1
showCurrentPositionStatus()	100.0 %	2	0	2
splitArray(String)	100.0 %	4	0	4
turnLeft()	100.0 %	6	0	6
turnRight()	100.0 %	6	0	6

Data Flow Testing

The QA team has to select one method from the application and perform data flow testing. In order to select a method, we calculated the number of def use pairs so that we could apply data flow testing for a complex function.

The moveForward ( ) method has the most definitions, c-used variables and p-used variables. Hence the data flow testing has been performed for the moveForward ( ) function from Robot.java.

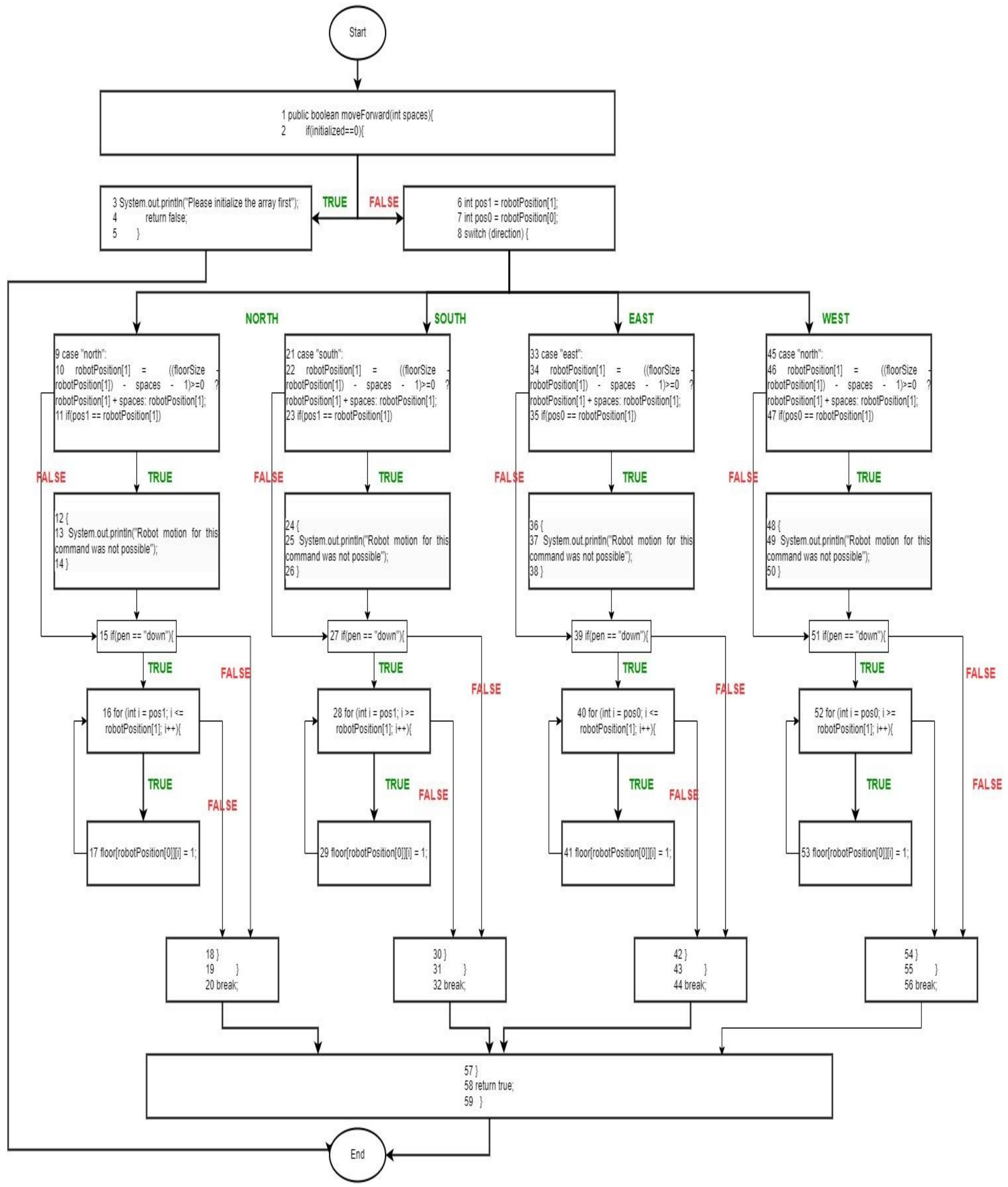
The initial step is to identify the nodes and construct Control Flow and Data Flow Graph for the function.



```
 1 public boolean moveForward(int spaces){
 2         if(initialized==0){
 3             System.out.println("Please initialize the array first");
 4             return false;
 5         }
 6         int pos1 = robotPosition[1];
 7         int pos0 = robotPosition[0];
 8         switch (direction) {
 9             case "north":
10                 robotPosition[1] =  ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces:
   robotPosition[1];
11                 if(pos1 == robotPosition[1])
12                 {
13                     System.out.println("Robot motion for this command was not possible");
14                 }
15                 if(pen == "down"){
16                     for (int i = pos1; i <= robotPosition[1]; i++){
17                         floor[robotPosition[0]][i] = 1;
18                     }
19                 }
20                 break;
21             case "south":
22                 robotPosition[1] =  (robotPosition[1]- spaces - 1)>=0 ? robotPosition[1] - spaces :
   robotPosition[1];
23                 if(pos1 == robotPosition[1])
24                 {
25                     System.out.println("Robot motion for this command was not possible");
26                 }
27                 if(pen == "down"){
28                     for (int i = pos1; i >=robotPosition[1]; i--){
29                         floor[robotPosition[0]][i] = 1;
30                     }
31                 }
32                 break;
33             case "east":
34                 robotPosition[0] = (floorSize - robotPosition[0] - spaces -1 )>=0 ? robotPosition[0] + spaces:
   robotPosition[0];
35                 if(pos0 == robotPosition[0])
36                 {
37                     System.out.println("Robot motion for this command was not possible");
38                 }
39                 if(pen == "down"){
40                     for (int i = pos0; i <= robotPosition[0]; i++){
41                         floor[i][robotPosition[1]] = 1;
42                     };
43                 }
44                 break;
45             case "west":
46                 robotPosition[0] =  (robotPosition[0]- spaces - 1)>=0 ? robotPosition[0] - spaces :
   robotPosition[0];
47                 if(pos0 == robotPosition[0])
48                 {
49                     System.out.println("Robot motion for this command was not possible");
50                 }
51                 if(pen == "down"){
52                     for (int i = pos0; i >=robotPosition[0]; i--){
53                         floor[i][robotPosition[1]] = 1;
54                     };
55                 }
56                 break;
57         }
58         return true;
59     }
```

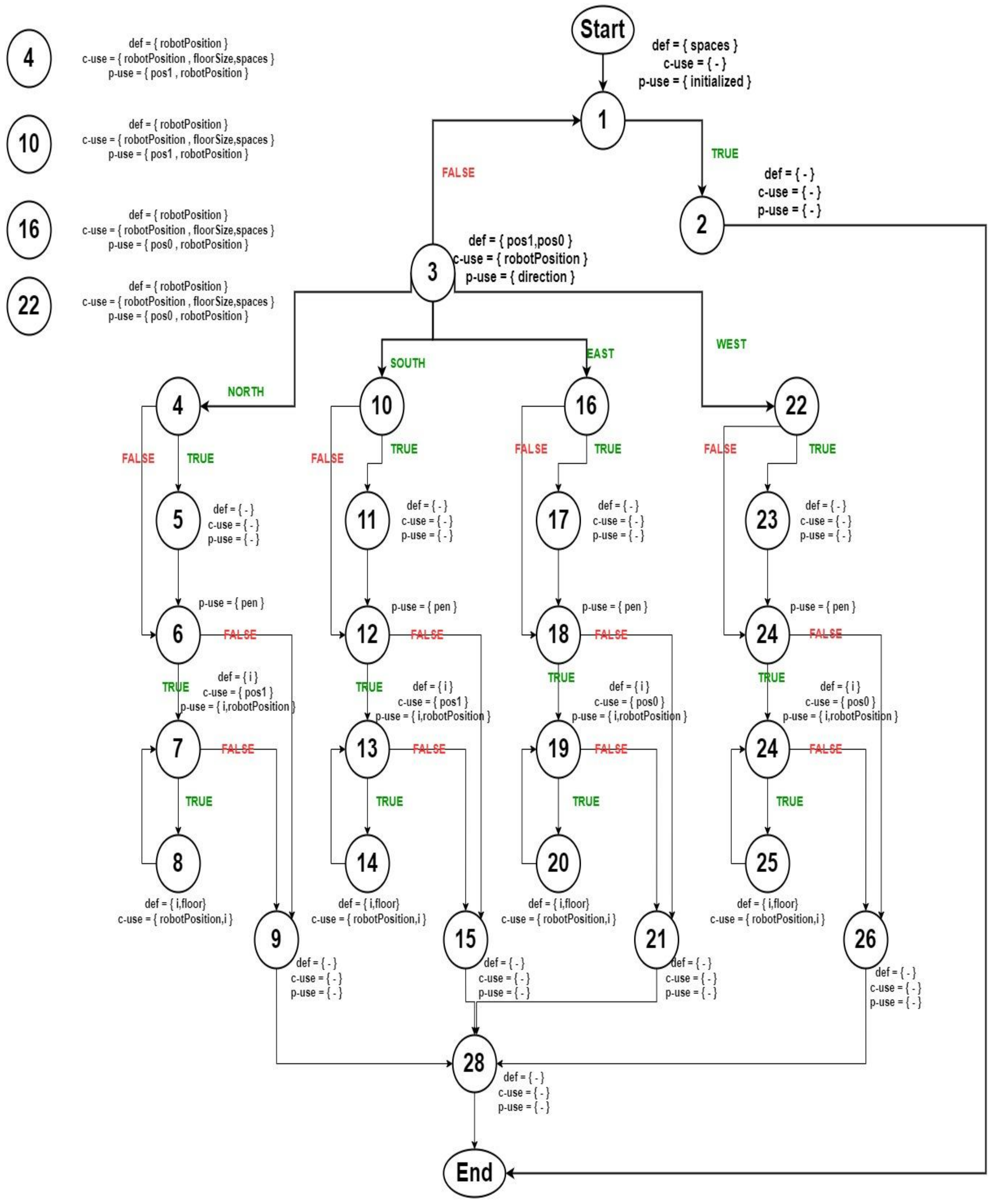
Nodes	**Lines**
Start	
1	1 , 2
2	3 , 4 , 5
3	6 , 7 , 8
4	9 , 10 , 11
5	12 , 13 , 14
6	15
7	16
8	17
9	18 , 19 , 20
10	21 , 22 , 23
11	24 , 25 , 26
12	27
13	28
14	29
15	30 , 31 , 32
16	33 , 34 , 35
17	36 , 37 , 38
18	39
19	40
20	41
21	42 , 43 , 44
22	45 , 46 , 47
23	48 , 49 , 50
24	51
25	52
26	53
27	54 , 55 , 56
28	57 , 58 , 59
End	

Control Flow Graph



```
1 public boolean moveForward(int spaces){
2       if(initialized==0){

3 System.out.println("Please initialize the array first");
4           return false;
5       }

6 int pos1 = robotPosition[1];
7 int pos0 = robotPosition[0];
8 switch (direction) {

9 case "north":
10 robotPosition[1] = ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces: robotPosition[1];
11 if(pos1 == robotPosition[1])

12 {
13 System.out.println("Robot motion for this command was not possible");
14 }

15 if(pen == "down"){

16 for (int i = pos1; i <= robotPosition[1]; i++){

17 floor[robotPosition[0]][i] = 1;

18 }
19       }
20 break;

21 case "south":
22 robotPosition[1] = ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces: robotPosition[1];
23 if(pos1 == robotPosition[1])

24 {
25 System.out.println("Robot motion for this command was not possible");
26 }

27 if(pen == "down"){

28 for (int i = pos1; i >= robotPosition[1]; i++){

29 floor[robotPosition[0]][i] = 1;

30 }
31       }
32 break;

33 case "east":
34 robotPosition[1] = ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces: robotPosition[1];
35 if(pos0 == robotPosition[1])

36 {
37 System.out.println("Robot motion for this command was not possible");
38 }

39 if(pen == "down"){

40 for (int i = pos0; i <= robotPosition[1]; i++){

41 floor[robotPosition[0]][i] = 1;

42 }
43       }
44 break;

45 case "north":
46 robotPosition[1] = ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces: robotPosition[1];
47 if(pos0 == robotPosition[1])

48 {
49 System.out.println("Robot motion for this command was not possible");
50 }

51 if(pen == "down"){

52 for (int i = pos0; i >= robotPosition[1]; i++){

53 floor[robotPosition[0]][i] = 1;

54 }
55       }
56 break;

57 }
58 return true;
59 }
```

Data Flow Graph



<u>Computational Use (C-use)</u>

When a variable occurs within an expression, in an output statement, return statement, assignment statement, as a parameter within a function call and in subscript expressions, all these are classified as **c-use (computational use)**.

<u>Predicate Use (P-use)</u>

If a variable occurs within an if, while, do-while and for statements, it is classified as **p-use (predicate use)**.

<u>Def-clear path</u>

A def-clear path for variable x is any route that begins at a node where x is defined and terminates at a node where x is used, without encountering any other redefinitions of x along the way (excluding the endpoint).

<u>dcu and dpu</u>

- Definition of a variable at line (l1) and its use at line (l2) constitute a **def-use** pair. l1 and l2 can be the same.
- **dcu (d_i (x))** - denotes the set of all nodes where di (x) is live and c - used
- **dpu (d_i (x))** - denotes the set of all edges where there is a clear path from node i to node k and x is p-used at node k.

The following two tables are made for all variables in moveForward ( ) method.

Node i	**def (i)**	**C-use (i)**	**edge (i,j)**	**p-use (i,j)**
Start				
1	spaces	-	(1,2) (1,3)	initialized
2	-	-	(2,end)	-
3	pos1 , pos0	robotPosition	(3,4) (3,10) (3,16) (3,22)	direction
4	robotPosition	robotPosition floorSize spaces	(4,5) (4,6)	pos1 robotPosition
5	-	-	(5,6)	-
6	-	-	(6,7) (6,9)	pen
7	i	pos1	(7,8) (7,9)	i

				robotPosition
8	floor i	robotPosition i	(8,7)	-
9	-	-	(9,28)	-
10	robotPosition	robotPosition floorSize spaces	(10,11) (10,12)	pos1 robotPostion
11	-	-	(11,12)	-
12	-	-	(12,13) (12,15)	pen
13	i	pos1	(13,14) (13,15)	i robotPosition
14	floor i	robotPosition i	(14,13)	-
15	-	-	(15,28)	-
16	robotPosition	robotPosition floorSize spaces	(16,17) (16,18)	pos0 robotPostion
17	-	-	(17,18)	-
18	-	-	(18,19) (18,21)	pen
19	i	pos0	(19,20) (19,21)	i robotPosition
20	floor i	robotPosition i	(20,19)	-
21	-	-	(21,28)	-
22	robotPosition	robotPosition floorSize spaces	(22,23) (22,24)	pos0 robotPostion
23	-	-	(23,24)	-
24	-	-	(24,25) (25,27)	pen
25	i	pos0	(25,26) (25,27)	i robotPosition
26	floor i	robotPosition i	(26,25)	-

27	-	-	(27,28)	-
28	-	-	(28, end)	-
End				

node (i)	dcu (v , i)	dpu (v , i)
1	dcu (spaces,1) = {4,10,16,22}	-

3	dcu (pos1,3) = {7,13}	dpu (pos1,3) = { (4,5) (4,6) }
		dpu (pos1,3) = { (10,11) (10,12) }

	dcu (pos0,3) = {19,25}	dpu (pos0,3) = { (16,17) (16,18)}
		dpu (pos0,3) = { (22,23) (22,24)}

4	dcu (robotPosition,4) = {8}	dpu (robotPosition,4) = {(4,5) (4,6)}
		dpu (robotPosition,4) = {(7,8) (7,9)}

7	dcu (i,7) = {8}	dpu (i,7) = {(7,8) (7,9)}
8	dcu (i,8) = {8}	dpu (i,8) = {(7,8) (7,9)}

10	dcu (robotPosition,10) = {14}	dpu (robotPosition,10) = {(10,11) (10,12)}
		dpu (robotPosition,10) = {(13,14) (13,15)}

13	dcu (i,13) = {14}	dpu (i,13) = {(13,14) (13,15)}
14	dcu (i,14) = {14}	dpu (i,14) = {(13,14) (13,15)}

16	dcu (robotPosition,16) = {20}	dpu (robotPosition,16) = {(16,17) (16,18)}
		dpu (robotPosition,16) = {(19,20) (19,21)}

19	dcu (i,19) = {20}	dpu (i,19) = {(19,20) (19,21)}
20	dcu (i,20) = {20}	dpu (i,20) = {(19,20) (19,21)}
22	dcu (robotPosition,22) =	dpu (robotPosition,22) = {(22,23)

	{26}	(22,24)}
		dpu (robotPosition,22) = {(25,26) (25,27)}

25	dcu (i,25) = {26}	dpu (i,25) = {(25,26) (25,27)}
26	dcu (i,26) = {26}	dpu (i,26) = {(25,26) (25,27)}

Developed test cases to cover all the dcu and dpu

The test cases created below covers all the def use pairs in the moveForward ( ) method

Test Case	**Values**	**(Variable ,node)**	**dcu's covered**	**dpu's covered**
Case 1	● floorSize = 5; ● robotPosition = {0, 0}; ● pen = "down"; ● direction = "north"; ● moveForward(2);	(spaces,1)	{4}	-
		(pos1,3)	{7}	{(4,6)}

		(robotPosition,4)	{8}	{(4,6)}
				{(7,8) (7,9)}

		(i,7)	{8}	{(7,8) (7,9)}
		(i,8)	{8}	{(7,8) (7,9)}
Case 2	● floorSize = 5; ● robotPosition = {5, 5}; ● pen = "down"; ● direction = "south"; ● moveForward(4);	(spaces,1)	{10}	-
		(pos1,3)	{13}	{(10,12)}

		(robotPosition,10)	{14}	{(10,12)}
				{(13,14) (13,15)}

		(i,13)	{14}	{(13,14) (13,15)}
		(i,14)	{14}	{(13,14) (13,15)}
Case 3	● floorSize = 5;	(spaces,1	{16}	-

Test Case	**Values**	**(Variable ,node)**	**dcu’s covered**	**dpu’s covered**
	● robotPosition = {0, 0}; ● pen = "down"; ● direction = "east"; ● moveForward(2);	)		
		(pos0,3)	{19}	{(16,18)}

		(robotPosition,16)	{20}	{(16,18)}
				{(19,20) (19,21)}

		(i,19)	{20}	{(19,20) (19,21)}
		(i,20)	{20}	{(19,20) (19,21)}
Case 4	● floorSize = 5; ● robotPosition = {0, 0}; ● pen = "down"; ● direction = "west"; ● moveForward(2);	(spaces,1)	{22}	-
		(pos0,3)	{25}	{(22,24)}

		(robotPosition,22)	{26}	{(22,24)}
				{(25,26) (25,27)}

		(i,25)	{26}	{(25,26) (25,27)}
		(i,26)	{26}	{(25,26) (25,27)}
Case 5	● floorSize = 5; ● robotPosition = {0, 4}; ● pen = "up"; ● direction = "north"; ● moveForward(1);	(spaces,1)	{4}	-
		(pos1,3)	{7}	{(4,5)}

		(robotPosition,4)	{8}	{(4,5)}
				{(7,8) (7,9)}

		(i,7)	{8}	{(7,8) (7,9)}
		(i,8)	{8}	{(7,8) (7,9)}
Case 6	● floorSize = 5; ● robotPosition = {4, 0}; ● pen = "up"; ● direction = "south"; ● moveForward(1);	(spaces,1)	{10}	-
		(pos1,3)	{13}	{(10,11)}
		(robotPo	{14}	{(10,11)}

Test Case	**Values**	**(Variable ,node)**	**dcu's covered**	**dpu's covered**
		sition,10)		{(13,14) (13,15)}
		(i,13)	{14}	{(13,14) (13,15)}
		(i,14)	{14}	{(13,14) (13,15)}
Case 7	● floorSize = 5; ● robotPosition = {4, 0}; ● pen = "up"; ● direction = "east"; ● moveForward(1);	(spaces,1)	{16}	-
		(pos0,3)	{19}	{(16,17)}

		(robotPosition,16)	{20}	{(16,17)}
				{(19,20) (19,21)}

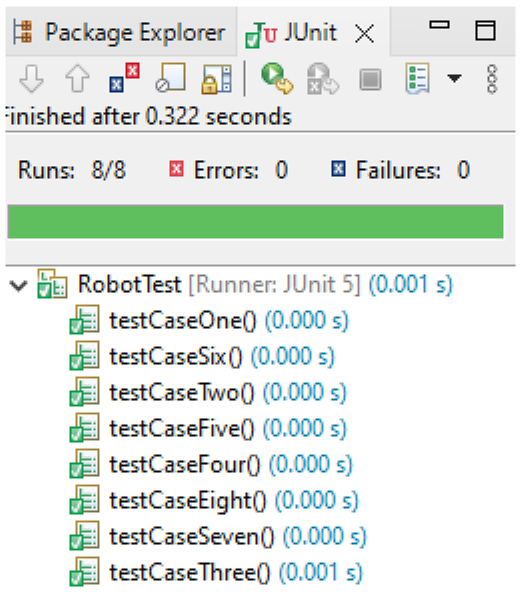
		(i,19)	{20}	{(19,20) (19,21)}
		(i,20)	{20}	{(19,20) (19,21)}
Case 8	● floorSize = 5; ● robotPosition = {0, 0}; ● pen = "up"; ● direction = "west"; ● moveForward(1);	(spaces,1)	{22}	-
		(pos0,3)	{25}	{(22,23)}

		(robotPosition,22)	{26}	{(22,23)}
				{(25,26) (25,27)}

		(i,25)	{26}	{(25,26) (25,27)}
		(i,26)	{26}	{(25,26) (25,27)}

Unit Test Cases

The following are the JUnit test cases execution for moveForward ( ) method based on the def-use pair approach.



```
inished after 0.322 seconds

Runs: 8/8    Errors: 0    Failures: 0

RobotTest [Runner: JUnit 5] (0.001 s)
    testCaseOne() (0.000 s)
    testCaseSix() (0.000 s)
    testCaseTwo() (0.000 s)
    testCaseFive() (0.000 s)
    testCaseFour() (0.000 s)
    testCaseEight() (0.000 s)
    testCaseSeven() (0.000 s)
    testCaseThree() (0.001 s)
```

C-Use Coverage

The C-Use Coverage of a test suite can be calculated as

$$\frac{CU_c}{(CU - CU_F)}$$

CU_c - Number of c-uses covered

CU_F - Number of infeasible c-uses

In this case,

CU **- 20** CU_c **- 20** CU_F **- 0**

$$\frac{CU_c}{(CU - CU_F)} = \frac{20}{20} = 1$$

P-Use Coverage

The P-Use Coverage of a test suite can be calculated as

$$\frac{PU_c}{(PU - PU_F)}$$

PU_c - Number of p-uses covered

PU_F - Number of infeasible p-uses

In this case,

PU **- 40** CP **- 40** PU_F **- 0**

$$\frac{PU_c}{(PU - PU_F)} = \frac{40}{40} = 1$$

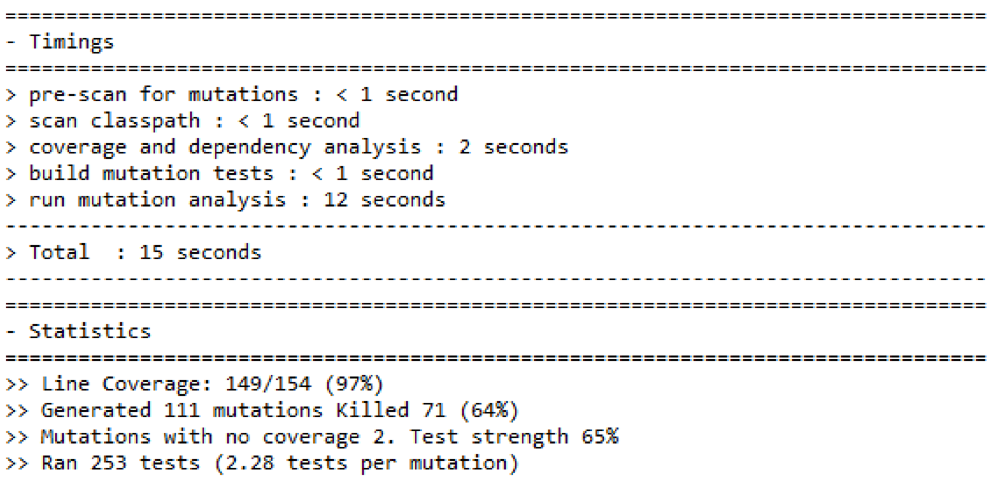
All-uses Coverage

The All-Use Coverage of a test suite can be calculated as

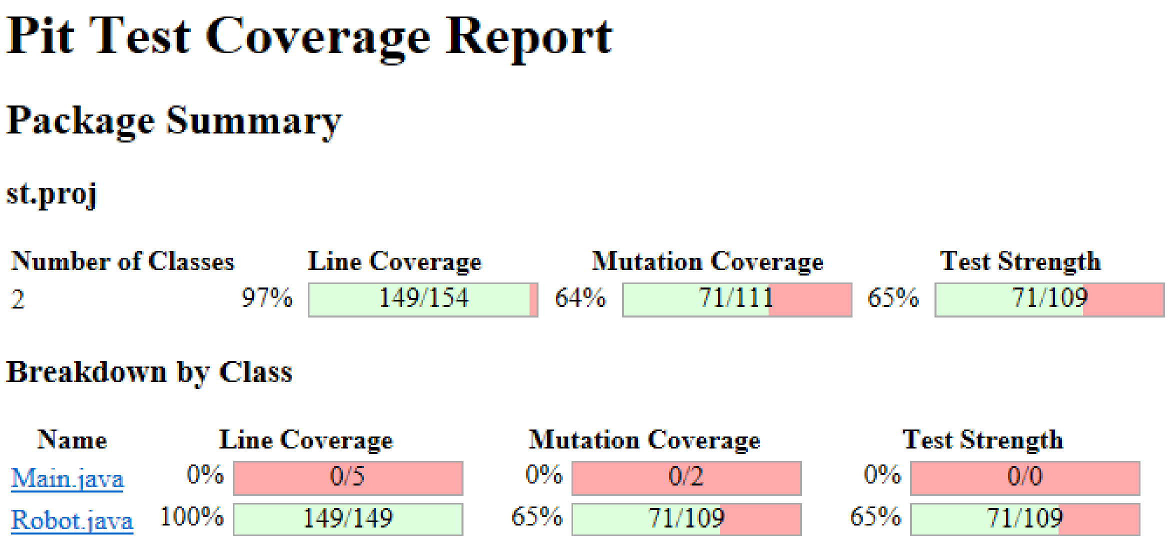
$$\frac{PU_c + CU_c}{(CU + PU - CU_F + PU_F)} = \frac{60}{60} = 1$$

Mutation Testing

Mutation testing is the process of testing software by introducing small deliberate alterations to the program's source code and verifying the code with available test cases. We have selected the moveForward ( ) method to perform mutation testing.



```
================================================================================
- Timings
================================================================================
> pre-scan for mutations : < 1 second
> scan classpath : < 1 second
> coverage and dependency analysis : 2 seconds
> build mutation tests : < 1 second
> run mutation analysis : 12 seconds
--------------------------------------------------------------------------------
> Total  : 15 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Line Coverage: 149/154 (97%)
>> Generated 111 mutations Killed 71 (64%)
>> Mutations with no coverage 2. Test strength 65%
>> Ran 253 tests (2.28 tests per mutation)
```



Pit Test Coverage Report

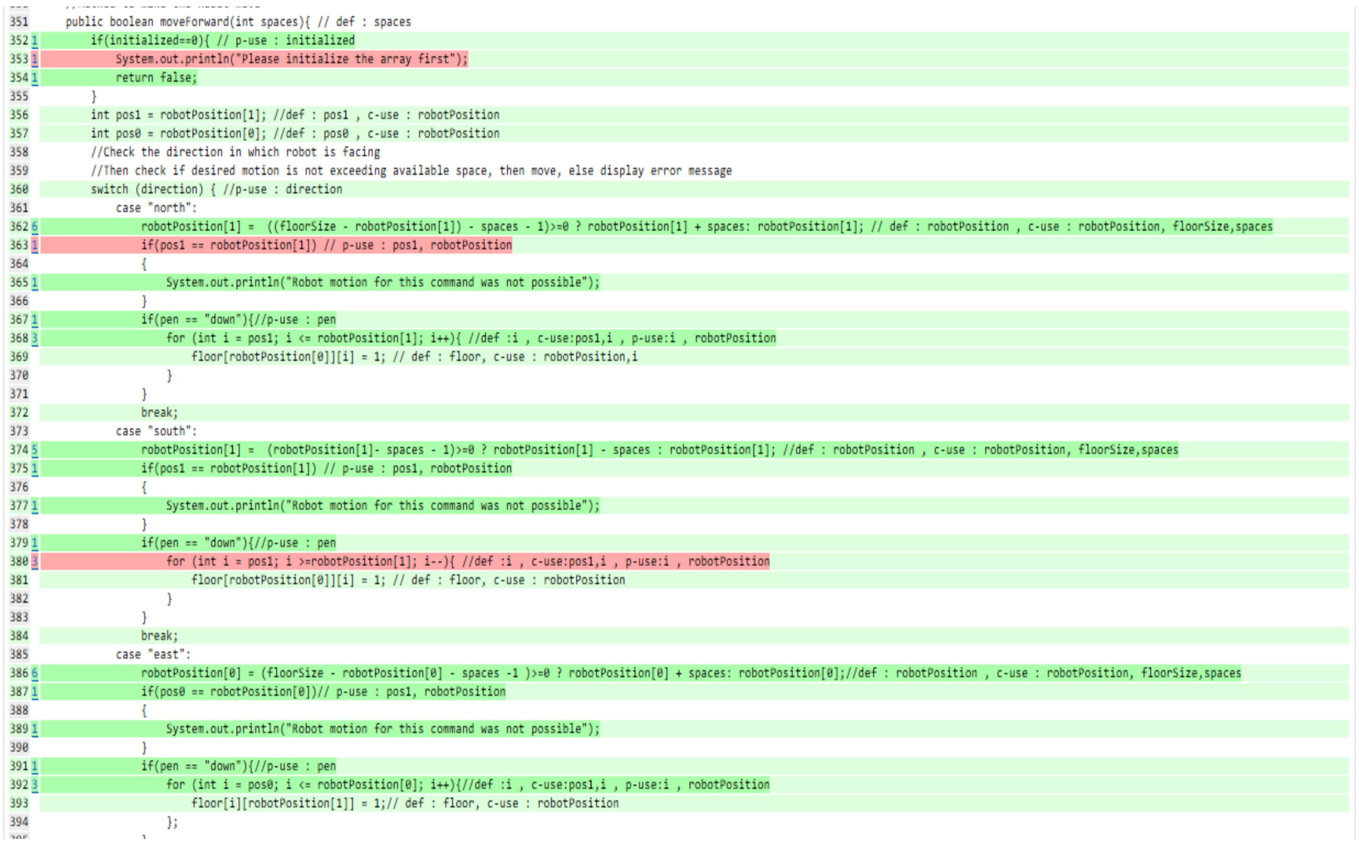
Package Summary

st.proj

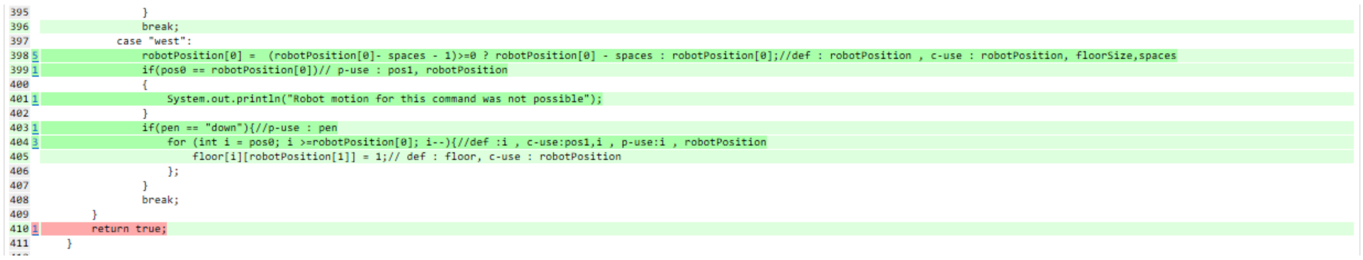
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	97%	149/154	64%	71/111	65%	71/109

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Main.java	0%	0/5	0%	0/2	0%	0/0
Robot.java	100%	149/149	65%	71/109	65%	71/109



```java
    public boolean moveForward(int spaces){ // def : spaces
        if(initialized==0){ // p-use : initialized
            System.out.println("Please initialize the array first");
            return false;
        }
        int pos1 = robotPosition[1]; //def : pos1 , c-use : robotPosition
        int pos0 = robotPosition[0]; //def : pos0 , c-use : robotPosition
        //Check the direction in which robot is facing
        //Then check if desired motion is not exceeding available space, then move, else display error message
        switch (direction) { //p-use : direction
            case "north":
                robotPosition[1] =  ((floorSize - robotPosition[1]) - spaces - 1)>=0 ? robotPosition[1] + spaces: robotPosition[1]; // def : robotPosition , c-use : robotPosition, floorSize,spaces
                if(pos1 == robotPosition[1]) // p-use : pos1, robotPosition
                {
                    System.out.println("Robot motion for this command was not possible");
                }
                if(pen == "down"){//p-use : pen
                    for (int i = pos1; i <= robotPosition[1]; i++){ //def :i , c-use:pos1,i , p-use:i , robotPosition
                        floor[robotPosition[0]][i] = 1; // def : floor, c-use : robotPosition,i
                    }
                }
                break;
            case "south":
                robotPosition[1] =  (robotPosition[1]- spaces - 1)>=0 ? robotPosition[1] - spaces : robotPosition[1]; //def : robotPosition , c-use : robotPosition, floorSize,spaces
                if(pos1 == robotPosition[1]) // p-use : pos1, robotPosition
                {
                    System.out.println("Robot motion for this command was not possible");
                }
                if(pen == "down"){//p-use : pen
                    for (int i = pos1; i >=robotPosition[1]; i--){ //def :i , c-use:pos1,i , p-use:i , robotPosition
                        floor[robotPosition[0]][i] = 1; // def : floor, c-use : robotPosition
                    }
                }
                break;
            case "east":
                robotPosition[0] = (floorSize - robotPosition[0] - spaces -1 )>=0 ? robotPosition[0] + spaces: robotPosition[0];//def : robotPosition , c-use : robotPosition, floorSize,spaces
                if(pos0 == robotPosition[0])// p-use : pos1, robotPosition
                {
                    System.out.println("Robot motion for this command was not possible");
                }
                if(pen == "down"){//p-use : pen
                    for (int i = pos0; i <= robotPosition[0]; i++){//def :i , c-use:pos1,i , p-use:i , robotPosition
                        floor[i][robotPosition[1]] = 1;// def : floor, c-use : robotPosition
                    };
```



```java
                }
                break;
            case "west":
                robotPosition[0] =  (robotPosition[0]- spaces - 1)>=0 ? robotPosition[0] - spaces : robotPosition[0];//def : robotPosition , c-use : robotPosition, floorSize,spaces
                if(pos0 == robotPosition[0])// p-use : pos1, robotPosition
                {
                    System.out.println("Robot motion for this command was not possible");
                }
                if(pen == "down"){//p-use : pen
                    for (int i = pos0; i >=robotPosition[0]; i--){//def :i , c-use:pos1,i , p-use:i , robotPosition
                        floor[i][robotPosition[1]] = 1;// def : floor, c-use : robotPosition
                    };
                }
                break;
        }
        return true;
    }
```

From the mutation Testing it is observed that few parts of the code are not covered completely from the mutation testing. Apart from that our test cases have covered 65% in mutation coverage testing.