



COEN 6761 Software Testing and Validation

Coverage Testing 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	3
	1.2 Objective	3
	1.3 GitHub-Url.....	3
	1.4 Tools and Technology used	3
2	User Stories and Requirements	4
3	Funtions and Methods	5
4	Implementation and Testing Strategy	11
	4.1 Development	11
	4.2 Testing.....	11
5	Mapping Unit Test VS Requirements	13
6	Test Case Excecution	14
7	Project Timeline	14
8	Timeline Chart.....	15
9	Code Analysis and Software release	17
	9.1 Function coverage	20
	9.2 Statement coverage	21
	9.3 Path coverage	22
	9.4 Condition coverage	23
	9.5 Line coverage.....	25
10	Conclusion.....	27
11	AI transparency	28

1. Introduction:

This project involves creating a Java-based robot simulation where a robot moves across a grid ($N \times N$) and performs various tasks based on user commands. The robot can toggle its pen between "up" and "down" states. When the pen is down, it traces paths as it moves, when the pen is up, it moves freely without leaving a mark. The robot starts at the position $[0, 0]$, facing north, and the floor grid is initially empty (all zeros). Users can control the robot's movements, pen state, and view the current state of the floor grid through a set of commands. Commands include moving forward, turning, printing the grid, and replaying the robot's actions. This simulation allows the user to interact with the robot, observe its actions, and track its path on the floor grid.

1.1 Objective:

The objective of this project was to develop the robot code, verify its functionality, and then create and execute corresponding test cases to ensure the program behaves as expected. The robot responds to various commands like moving, turning, toggling the pen, and displaying the current state of the grid. The testing focuses on checking the robot's movement, pen functionality, and how well the grid updates as the robot performs actions. The project aims to ensure that the robot behaves reliably and accurately by thoroughly testing each part of the system with well-structured test cases.

1.2 GitHub Url:

<https://github.com/Vraj-2011/COEN-6761>

1.3 Tools and Technologies Used:

- **Java:** programming language used for the implementation of the robot simulation.
- **JUnit:** A testing framework used to create unit tests for the program. Testing ensures that each class and method in the system works as expected.
- **Maven:** An automation tool used for managing project dependencies and building the project. It helps ensure that dependencies like JUnit are properly included, and it provides a structured way to build and package the project.

- **Jira:** A project management and issue-tracking tool used for time tracking. Jira will be used to monitor the time spent on different tasks such as development, testing, and documentation.
- **GitHub:** Version control using GitHub will store the source code, track changes, and allow for collaboration or sharing of the project. All code, including classes and test cases, will be committed to GitHub for easy access and version history.

2. User Stories and Requirements:

A. Requirement-1 History [H | h]:

As a user, I want to replay all the steps I have taken since the previous start of the program so that I can review my actions and correct any mistakes. The replay should occur in the original order displaying all the previous steps taken by the robot.

B. Requirement-2 printHelp():

As a user, I want a comprehensive help function in the application so that I can easily access guidance and support for using its features. The help function should include list of commands for the movement of the robot.

C. Requirement-3 Invalid():

As a user, I want the robot to start from a default state, including the starting position at (0, 0), the pen being up, and facing north. This ensures that I have a consistent starting point for my commands, making it easier to plan my movements.

D. Requirement-4 initialize(int n):

As a user, I want to initialize the robot's environment with a specified grid size n, so I can control the robot within a defined space. If I provide an invalid size, I expect the program to inform me that the grid cannot be initialized, allowing me to correct my input.

E. Requirement-5 penup():

As a user, I want to raise the robot's pen so that it stops tracing paths on the grid. This feature allows me to move the robot freely without marking the floor, which is useful when I want to reposition the robot without leaving a trace

F. Requirement-6 pendown():

As a user, I want to lower the robot's pen so that it starts tracing paths on the grid as it moves. This enables me to create shapes or patterns on the floor, allowing us to trace the robot's operation.

G. Requirement-7 turnright():

As a user, I want to turn the robot to the right so that I can change its facing direction without moving. This capability is essential for navigating corners and adjusting the robot's orientation to create more complex shapes.

H. Requirement-8 turnleft():

As a user, I want to turn the robot to the left so that I can change its facing direction without moving. This flexibility allows me to easily navigate the grid and adjust the robot's path as needed.

I. Requirement-9 move(int steps):

As a user, I want to move the robot forward by a specified number of steps in the direction it is currently facing. If the pen is down, I expect the robot to mark its path on the grid, allowing me to visualize the robot's movements.

J. Requirement-10 printFloor():

As a user, I want to print the current state of the grid so that I can see the shapes traced by the robot. The output should clearly display asterisks for marked areas and blanks for unmarked areas, helping me understand the robot's path.

K. Requirement-11 getFloor():

As a user, I want to be able to successfully execute the quit command in the application so that I can exit the program without any errors. The application should confirm the action and ensure that all unsaved data is either saved or discarded appropriately before closing. This will enhance my experience by providing a smooth and reliable way to exit the application.

L. Requirement-12 printStatus():

As a user, I want to check the robot's current position, pen status, and facing direction so that I can keep track of its state. This information is vital for making informed decisions about the next commands to issue, ensuring smooth operation of the robot.

3. Functions & Methods:

A. public static void main(String[] args):

```
public class Main {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        RobotController robot = new RobotController();  
    }  
}
```

- **Purpose:** This class initializes the program, sets up necessary objects, and manages user input to control the robot simulation.
- **Functionality:**

- Initializes a Scanner object to read user input from the console.
- Creates an instance of RobotController to manage the robot's actions.
- Enters an infinite loop to continuously accept and process user commands until the user decides to exit by entering 'q'.
- Parses user input to determine the appropriate action, such as initializing the grid, moving the robot, changing the pen state, turning, printing the grid or status, displaying help, or showing command history.
- Manages invalid commands by prompting the user with an error message.

B. private static void printHelp():

```
private static void printHelp() {  
    System.out.println("[U|u] Pen up");  
    System.out.println("[D|d] Pen down");  
    System.out.println("[R|r] Turn right");  
    System.out.println("[L|l] Turn left");  
    System.out.println("[M s|m s] Move forward s spaces");  
    System.out.println("[P|p] Print the grid");  
    System.out.println("[C|c] Print current status");  
    System.out.println("[I n|i n] Initialize the system");  
    System.out.println("[H|h] Print the history of actions");  
    System.out.println("[Q|q] Quit the program");  
}
```

- **Purpose:** Provides users with a list of available commands and their descriptions to assist in interacting with the robot simulation.
- **Functionality:**
 - Prints a series of lines to the console, each describing a command that the user can input, such as lifting or lowering the pen, turning the robot, moving forward, printing the grid, checking the robot's status, initializing the system, viewing the history of actions, and quitting the program

C. public RobotController() (Constructor):

```
public RobotController() {  
    // Default initialization  
    x = 0;  
    y = 0;  
    penDown = false;  
    direction = Direction.NORTH;  
    history = new ArrayList<>();  
}
```

- **Purpose:** Initializes the robot with default values.

- **Functionality:**
 - Sets the starting position ($x = 0, y = 0$).
 - Sets the pen to be **up** (`penDown = false`).
 - Sets the robot's initial direction to **NORTH**.
 - Initializes an empty **history list** to track actions.

D. public void initialize(int n):

```
public void initialize(int n) {  
    if (n <= 0) {  
        System.out.println("Grid size must be greater than 0.");  
        return;  
    }  
    floor = new int[n][n];  
    x = 0;  
    y = 0;  
    penDown = false;  
    direction = Direction.NORTH;  
    history.add("Initialized " + n + "x" + n + " grid.");  
    System.out.println("Initialized " + n + "x" + n + " grid.");  
}
```

- **Purpose:** Creates an $n \times n$ grid and resets the robot.
- **Functionality:**
 - If n is less than or equal to zero, prints an error.
 - Initializes a floor matrix of size $n \times n$.
 - Resets the robot's position to $(0, 0)$, sets the pen **up**, and sets direction to **NORTH**.
 - Adds an entry to history and prints a confirmation message.

E. public void penUp():

```
public void penUp() {  
    penDown = false;  
    history.add("Pen is now up.");  
    System.out.println("Pen is now up.");  
}
```

- **Purpose:** Lifts the robot's pen so it **does not draw** while moving.

- **Functionality:**
 - Sets penDown = false.
 - Adds an entry to history and prints a message.

F. public void penDown():

```
public void penDown() {  
    penDown = true;  
    history.add("Pen is now down.");  
    System.out.println("Pen is now down.");  
}
```

- **Purpose:** Lowers the pen so the robot **starts drawing** while moving.
- **Functionality:**
 - Sets penDown = true.
 - Adds an entry to history and prints a message.

G. public void turnRight():

```
public void turnRight() {  
    switch (direction) {  
        case NORTH -> direction = Direction.EAST;  
        case EAST -> direction = Direction.SOUTH;  
        case SOUTH -> direction = Direction.WEST;  
        case WEST -> direction = Direction.NORTH;  
    }  
    history.add("Turned right. Now facing " + direction + ".");  
    System.out.println("Turned right. Now facing " + direction + ".");  
}
```

- **Purpose:** Rotates the robot **90 degrees clockwise**.
- **Functionality:**
 - Updates the direction based on the current direction.
 - Adds an entry to history and prints the new direction.

H. public void turnLeft():

```
public void turnLeft() {
    switch (direction) {
        case NORTH -> direction = Direction.WEST;
        case WEST -> direction = Direction.SOUTH;
        case SOUTH -> direction = Direction.EAST;
        case EAST -> direction = Direction.NORTH;
    }
    history.add("Turned left. Now facing " + direction + ".");
    System.out.println("Turned left. Now facing " + direction + ".");
}
```

- **Purpose:** Rotates the robot **90 degrees counterclockwise**.
- **Functionality:**
 - Updates the direction based on the current direction.
 - Adds an entry to history and prints the new direction.

I. public void move(int steps):

```
public void move(int steps) {
    for (int i = 0; i < steps; i++) {
        switch (direction) {
            case NORTH -> y = Math.min(y + 1, floor.length - 1);
            case SOUTH -> y = Math.max(y - 1, 0);
            case EAST -> x = Math.min(x + 1, floor[0].length - 1);
            case WEST -> x = Math.max(x - 1, 0);
        }
        if (penDown) {
            floor[y][x] = 1;
        }
    }
    history.add("Moved " + steps + " steps.");
    System.out.println("Moved " + steps + " steps.");
}
```

- **Purpose:** Moves the robot forward in the current direction.
- **Functionality:**
 - Moves one step at a time up to steps while ensuring **it does not move out of bounds**.
 - If the pen is **down**, marks the grid cell ($\text{floor}[y][x] = 1$).
 - Adds an entry to history and prints a message.

J. public void printFloor():

```
public void printFloor() {
    if (floor == null) {
        System.out.println("Error: Grid is not initialized.");
        return;
    }
    System.out.println("Floor:");
    for (int i = floor.length - 1; i >= 0; i--) {
        for (int j = 0; j < floor[i].length; j++) {
            System.out.print(floor[i][j] == 1 ? "*" : " ");
        }
        System.out.println();
    }
}
```

- **Purpose:** Displays the current grid, showing where the robot has drawn.
- **Functionality:**
 - If the grid is not initialized, prints an error.
 - Iterates over the floor array and prints "*" where 1 is found (indicating drawn lines) and " " for 0 (empty spaces).

K. public int[][] getFloor():

```
public int[][] getFloor() {
    return floor;
}
```

- **Purpose:** Returns the current grid (floor).
- **Functionality:** Returns the floor array for external use.

L. public void printStatus():

```
public void printStatus() {
    System.out.println("Position: [" + x + ", " + y + "] - Pen: " + (penDown ? "down" : "up") + " - Facing: " + direction);
}
```

- **Purpose:** Prints the robot's **current position**, **pen status**, and **facing direction**.
- **Functionality:**
 - Displays x, y coordinates.

- Shows whether the pen is **up or down**.
- Shows the direction the robot is facing.

M. public void printHistory():

```
public void printHistory() {  
    if (history.isEmpty()) {  
        System.out.println("No history available.");  
    } else {  
        System.out.println("History:");  
        for (String record : history) {  
            System.out.println(record);  
        }  
    }  
}
```

- **Purpose:** Displays a list of all commands executed by the robot.
- **Functionality:**
 - If the history list is empty, prints "No history available."
 - Otherwise, prints each action stored in history

4. Implementation and Testing Strategy:

4.1 Development:

The development process follows an iterative and incremental approach, with major milestones to ensure functionality is added progressively and validated.

Step 1: Create Basic Classes

- 1) Implement the RobotController class first to handle the robot's position, pen state, and facing direction.
- 2) Implement the Main class to handle user input and simulate the robot's actions.

Step 2: Define Robot Behaviour

- 1) Implement movement logic for the robot. The move() method will update the robot's position in the floor array, considering its facing direction and the pen's state.
- 2) Implement turning logic using the turnLeft() and turnRight() methods to change the robot's facing direction.

Step 3: Add Pen Functionality

- 1) Implement the penUp() and penDown() methods to allow the user to toggle the pen.

- 2) Update the floor array when the pen is down, marking cells as "1" when the robot moves over them.

Step 4: User Interaction and Command Parsing

- 1) Implement logic to handle user input for commands such as moving, turning, and toggling the pen.
- 2) We Have used a while loop to recognise and toggle commands and invoke the appropriate methods in the RobotController class.

Step 5: Implement Replay and History Functionality

Track all robot actions in a history list. Implement the replayHistory() method to replay actions from the start using the "H" command.

4.2 Testing Strategy:

The testing strategy for this project is designed to validate the behaviour and functionality of two main classes: the Main Class and the RobotController Class.

Main Class: This class handles all user inputs, processes the commands, and generates the output. The focus of testing for this class is to ensure that the user input is correctly parsed and that the appropriate actions are performed.

We will test:

- 1) Command parsing: Ensuring that the correct function is called for each user input.
- 2) Output generation: Verifying that the correct output is displayed, such as the robot's position, pen state, and grid status.
- 3) Handling edge cases, such as invalid inputs or commands that might cause the robot to exceed the grid boundaries.

RobotController Class: This class contains the functions that define the robot's behaviour, such as moving, changing the pen state, and displaying the grid. The testing of this class will focus on ensuring the correctness of the robot's movements, status updates, etc. We will test:

- 1) Movement Functions: Verifying that the moveLeft(), moveRight(), and movement logic are implemented correctly, and that the robot moves to the correct position on the grid when commands are executed.
- 2) Pen Functions: Testing the penUp() and penDown() functions to ensure the pen state changes as expected.
- 3) Status Functions: Validating the currentStatus() method to confirm the robot's position, pen state, and facing direction are accurately tracked and displayed.
- 4) Grid Display: Ensuring the printFloor() function correctly displays the grid with "1" (for traced areas) and "0" (for untraced areas) and accurately reflects the robot's movements.
- 5) Replay History: Testing the history replay function to ensure the robot's actions can be correctly replayed from the start, as intended.

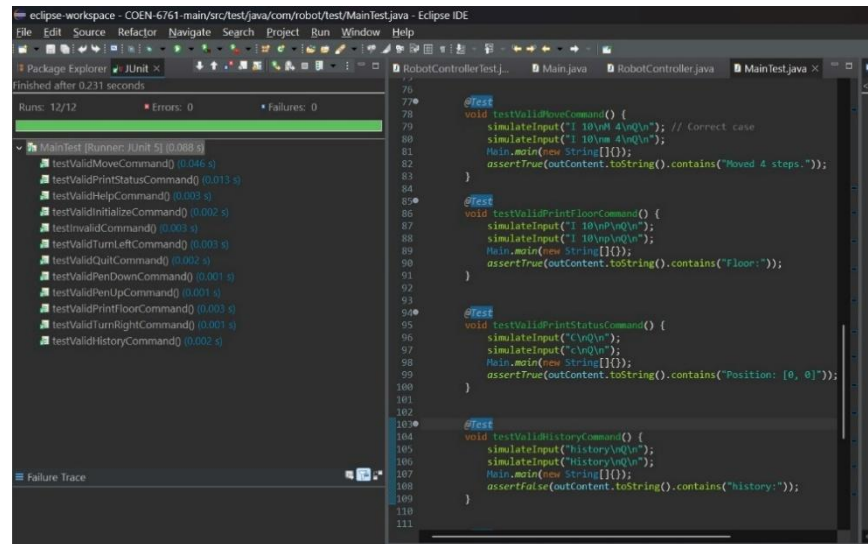
5. Mapping Unit test vs. Requirements

- Testcase-1 (T1): testValidHistoryCommand()
- Testcase-2 (T2): testValidHelpCommand()
- Testcase-3 (T3): testInvalidCommand()
- Testcase-4 (T4): testValidInitializerCommand()
- Testcase-5 (T5): testValidPenUpCommand()
- Testcase-6 (T6): testValidPenDownCommand()
- Testcase-7 (T7): testValidTurnRightCommand()
- Testcase-8 (T8): testValidTurnLeftCommand()
- Testcase-9 (T9): testValidMoveCommand()
- Testcase-10 (T10): testValidPrintFloorCommand()
- Testcase-11 (T11): testValidQuitCommand()
- Testcase-12 (T12): testValidPrintStatusCommand()

Requirements	Unit Test
Requirement-1(R1)	Testcase-1(T1)
Requirement-2(R2)	Testcase-2(T2)
Requirement-3(R3)	Testcase-3(T3)
Requirement-4(R4)	Testcase-4(T4)
Requirement-5(R5)	Testcase-5(T5)
Requirement-6(R6)	Testcase-6(T6)
Requirement-7(R7)	Testcase-7(T7)
Requirement-8(R8)	Testcase-8(T8)
Requirement-9(R9)	Testcase-9(T9)
Requirement-10(R10)	Testcase-10(T10)
Requirement-11(R11)	Testcase-11(T11)
Requirement-12(R12)	Testcase-12(T12)

6. Test cases execution

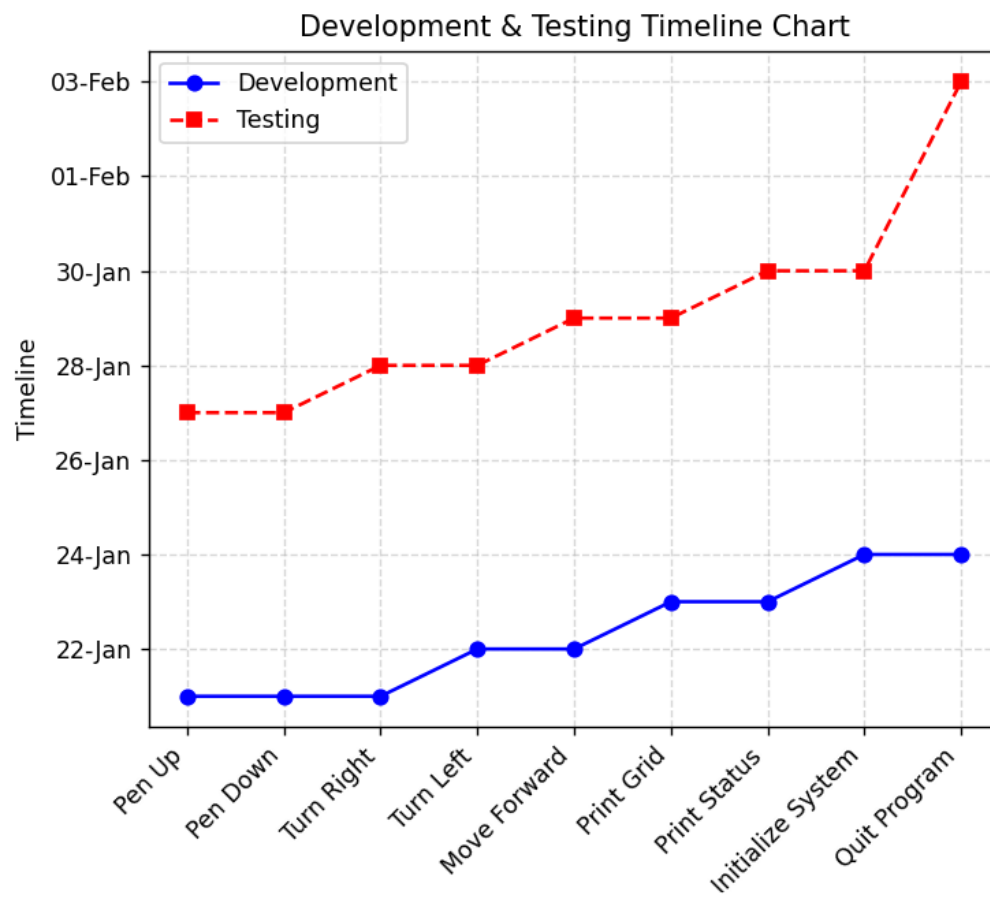
TEST CASE ID	TEST CASE NAME	TEST CASE DESCRIPTION	TEST CASE RESULT PASS/FAIL
TC1	testValidMoveCommand()	Tests if the move command functions correctly.	PASS
TC2	testValidPrintStatusCommand()	Checks if the status printing command works as expected.	PASS
TC3	testValidHelpCommand()	Ensures that the help command provides the correct instructions.	PASS
TC4	testValidInitializeCommand()	Verifies that the system initializes properly.	PASS
TC5	testInvalidCommand()	Tests the behavior when an invalid command is given.	PASS
TC6	testValidTurnLeftCommand()	Checks if the left turn command executes correctly.	PASS
TC7	testValidQuitCommand()	Ensures that the quit command properly exits the system.	PASS
TC8	testValidPenDownCommand()	Tests if the pen-down command is functioning.	PASS
TC9	testValidPenUpCommand()	Verifies the pen-up command's correctness.	PASS
TC10	testValidPrintFloorCommand()	Ensures the floor printing function works as expected.	PASS
TC11	testValidTurnRightCommand()	Checks if the right turn command is implemented properly.	PASS
TC12	testValidHistoryCommand()	Tests if the command history feature functions correctly.	PASS



7. Project Timeline

FUNCTION	APPLICATION DEVELOPMENT TIME	TESTING CASE DEVELOPMENT TIME
Pen up	21/01/2025 (1 DAY)	27/01/2025 (1 days)
Pen down	21/01/2025 (1 DAY)	27/01/2025 (1 day)
Turn right	21/01/2025 (1 DAY)	28/01/2025 (1 day)
Turn left	22/01/2025 (1 DAY)	28/01/2025 (1 day)
Move forward s spaces	22/01/2025 (1 DAY)	29/01/2025 (1 day)
Print the grid	23/01/2025 (1 DAY)	29/01/2025 (1 day)
Print status	23/01/2025 (1 DAY)	30/01/2025 (1 day)
Initialize the system	24/01/2025 (1 DAY)	30/01/2025 (1 day)
Quit the program	24/01/2025 (1 DAY)	03/02/2025 (1 day)

8. Timeline Chart



9. Code analysis and software releasing

1.] Updated Maven build file for new tools used :

JaCoCo has been integrated into the Maven build process to measure code coverage and ensure code quality. The plugin collects coverage data during test execution and generates detailed reports, providing insights into function, statement, branch, and line coverage. By enforcing predefined coverage thresholds, it helps maintain a robust testing strategy, ensuring that critical parts of the codebase are adequately tested. The reports generated by JaCoCo allow developers to identify untested areas and improve test effectiveness before release.

2.] Github link

<https://github.com/Vraj-2011/COEN-6761.git>

3.] Code coverage threshold values

The predefined code coverage threshold values for releasing the project ensure that the software meets a minimum quality standard before deployment. Specifically, the **instruction coverage** threshold is set at **80%**, meaning at least 80% of all executed instructions must be covered by tests. Additionally, the **branch coverage** threshold is set at **75%**, ensuring that at least 75% of all decision points, such as if-else conditions and loops, are executed during testing. These thresholds are enforced using the **JaCoCo Maven Plugin**, and if the coverage falls below these values, the build process will fail during the verification phase. By implementing these limits, the project maintains a strong level of test coverage, reducing the risk of undetected bugs and improving overall software reliability.

```

37     </plugin>
38     <plugin>
39         <groupId>org.jacoco</groupId>
40         <artifactId>jacoco-maven-plugin</artifactId>
41         <version>0.8.10</version>
42         <executions>
43             <execution>
44                 <goals>
45                     <goal>prepare-agent</goal>
46                 </goals>
47             </execution>
48             <execution>
49                 <id>report</id>
50                 <phase>verify</phase>
51                 <goals>
52                     <goal>report</goal>
53                 </goals>
54             </execution>
55         </executions>
56         <configuration>
57             <rules>
58                 <rule>
59                     <element>BUNDLE</element>
60                     <limits>
61                         <limit>
62                             <counter>INSTRUCTION</counter>
63                             <value>COVEREDRATIO</value>
64                             <minimum>0.80</minimum>
65                         </limit>
66                         <limit>
67                             <counter>BRANCH</counter>
68                             <value>COVEREDRATIO</value>
69                             <minimum>0.75</minimum>
70                         </limit>
71                     </limits>
72                 </rule>
73             </rules>
74         </configuration>
75     </plugin>
76 </plugins>
77 </build>
78 </project>
79
80

```

Overview Dependencies Dependency Hierarchy Effective POM pom.xml

Fig 9: Pom.xml Plugins

Code coverage results

SR no.	Coverage Type	Main Class	RobotController Class	Overall Coverage
1	Function	100% (2/2 methods executed)	91% (11/12 methods executed)	92% (13/14 methods executed)
2	Statement	90% (46/51 lines executed)	74% (55/74 lines executed)	80% (101/125 lines executed)
3	Path	75	45	60%
4	Condition	88%	54%	71%
5	Line	80% (101/125)	74% (55/74)	78% (156/199)
6	Branch	88% (23/26 branches executed)	54% (19/35 branches executed)	69% (42/61 branches executed)

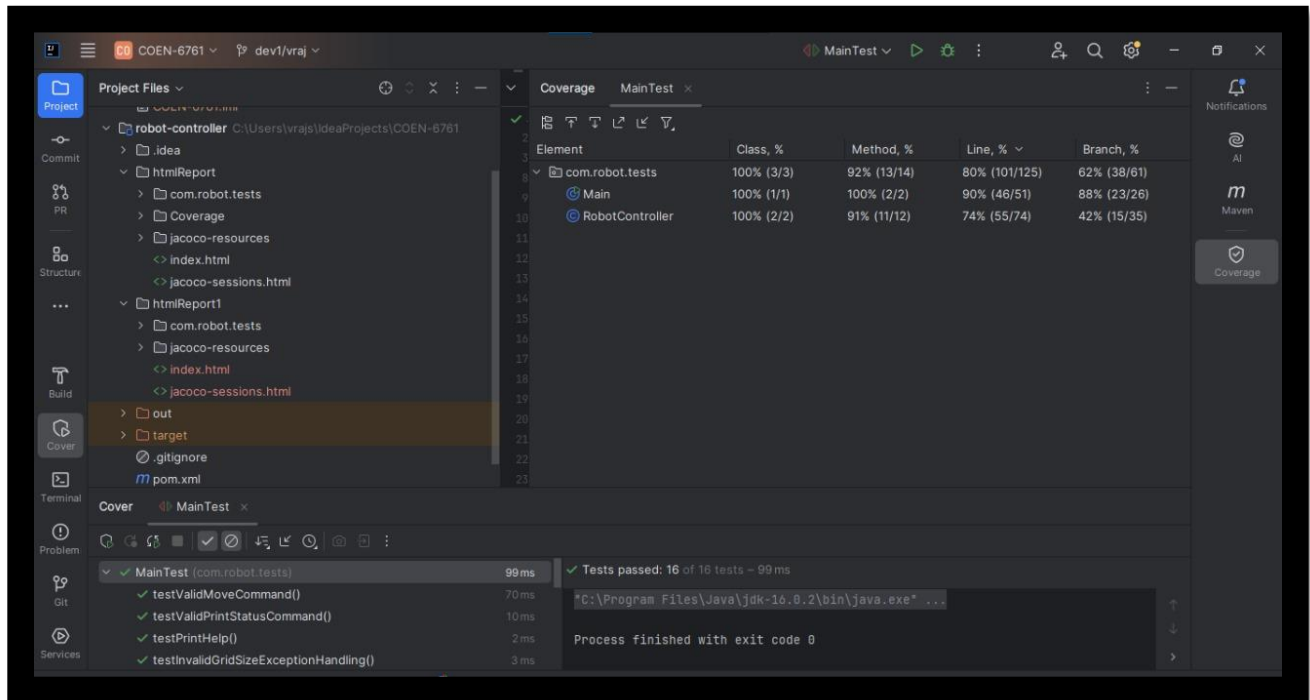


Fig 9.1: Main Class Coverage

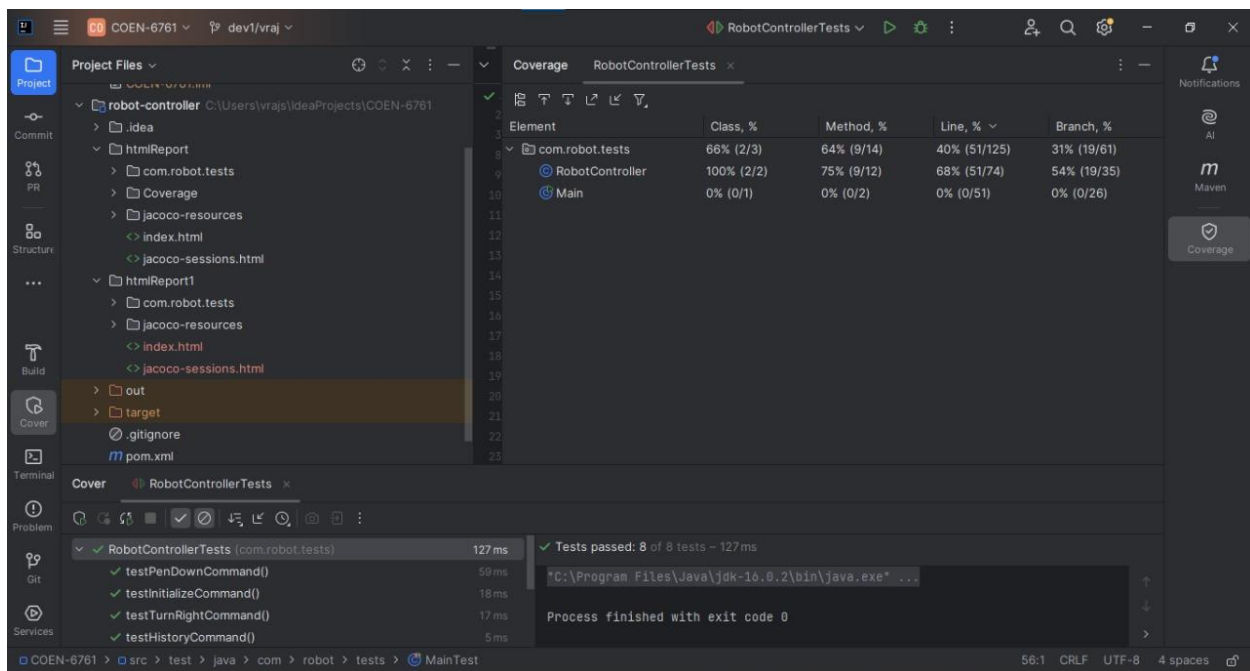


Fig 9.2: RobotController Class

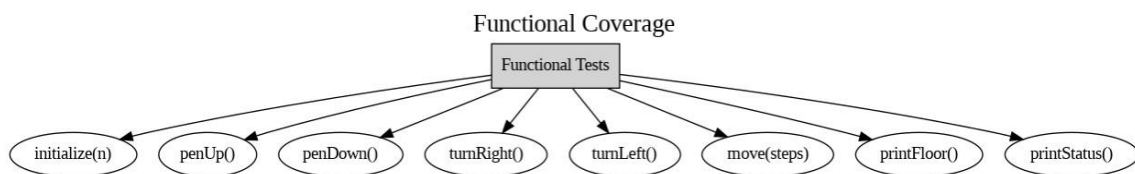
9.1 Functional Coverage:

Functional coverage is a metric that ensures all key functions in a program are executed at least once during testing. Unlike statement or branch coverage, it focuses on verifying whether the system's core functionalities are exercised, ensuring all functional requirements are met.

Objective: Ensure every function is invoked during testing.

The Robot Project includes core functions, each covered by dedicated test cases:

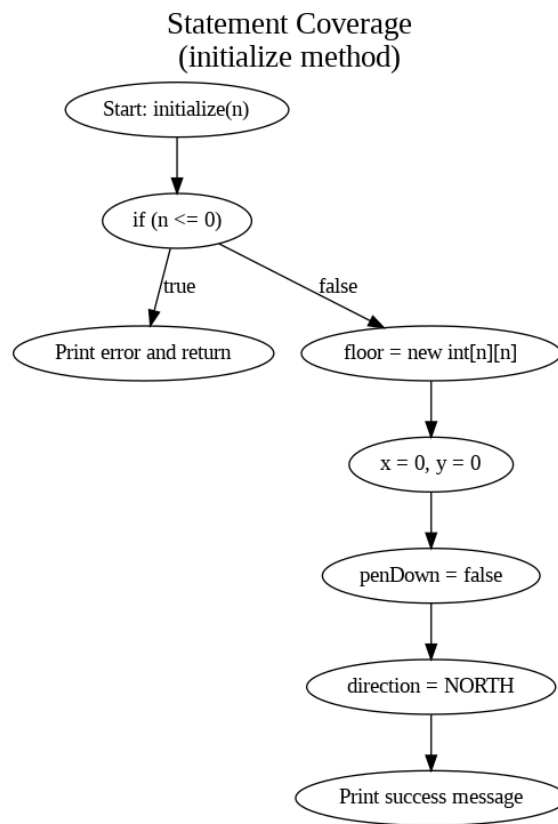
1. Initialization and Setup: Tests- testInitializeValidGrid: TC4
2. Pen Control: Tests- penUp(), penDown(): TC8, TC9
3. Direction Control: Tests- turnRight(): TC11
4. Movement: testValidMoveCommand(): TC1
5. Output Functions: printFloor(), printStatus() TC2, TC11
6. Interactive Input (Main Class): TC1, TC3, TC5, TC11



9.2 Statement Coverage:

Statement coverage measures the percentage of executed statements out of the total statements in the program.

90% (46/51 lines executed)	74% (55/74 lines executed)	80% (101/125 lines executed)
-----------------------------------	-----------------------------------	-------------------------------------



9.3 Path Coverage:

Path coverage measures **how many execution paths** through the program have been tested.

Main Class Path Coverage

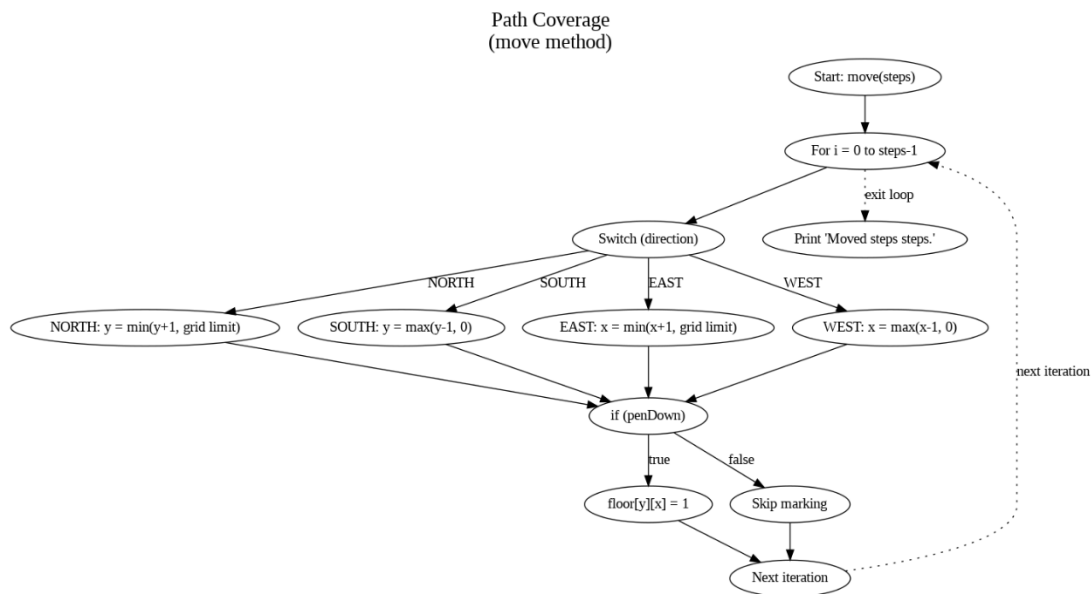
- Branch Coverage: 88% (3 of 26 branches missed)
- Total Complexity (Paths): 16
- Missed Complexity: 4
- Executed Paths: $16 - 4 = 12$

Path Coverage = $(12/16) \times 100 = 75\%$

RobotController Class Path Coverage

- Branch Coverage: 54% (16 of 35 branches missed)
- Total Complexity (Paths): 33
- Missed Complexity: 18
- Executed Paths: $33 - 18 = 15$

Path Coverage = $(15/33) \times 100 = 45\%$



9.4 Condition Coverage:

Condition coverage measures whether each Boolean condition has been tested for both true and false.

Main Class Condition Coverage

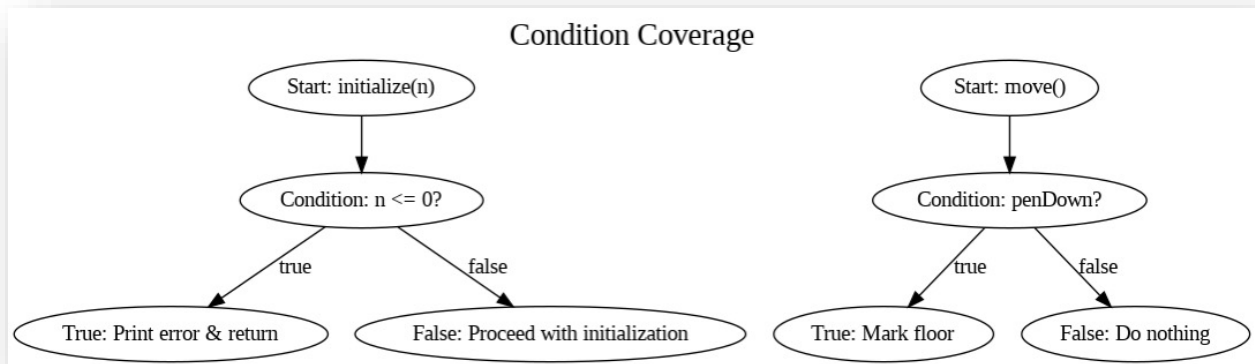
- Total Conditions: 26
- Missed Conditions: 3
- Executed Conditions: 23

Condition Coverage = $(23/26) \times 100 = 88\%$

RobotController Class Condition Coverage

- Total Conditions: 35
- Missed Conditions: 16
- Executed Conditions: 19

Condition Coverage = $(19/35) \times 100 = 54\%$



Cyclomatic Complexity Calculation for RobotController

Cyclomatic complexity (CC) is calculated using:

$$CC = E - N + 2P$$

where:

- **E** = Number of edges in the control flow graph
- **N** = Number of nodes in the control flow graph
- **P** = Number of connected components (typically 1 for a single method unless multiple functions are analysed separately)

Cyclomatic Complexity for Main Class

Decision Points:

- while (true) loop → 1
- Multiple if-else if conditions (checking user input) → 10
- Exception handling (try-catch) → 2

Total Complexity of main(): $CC=1+10+2=13$

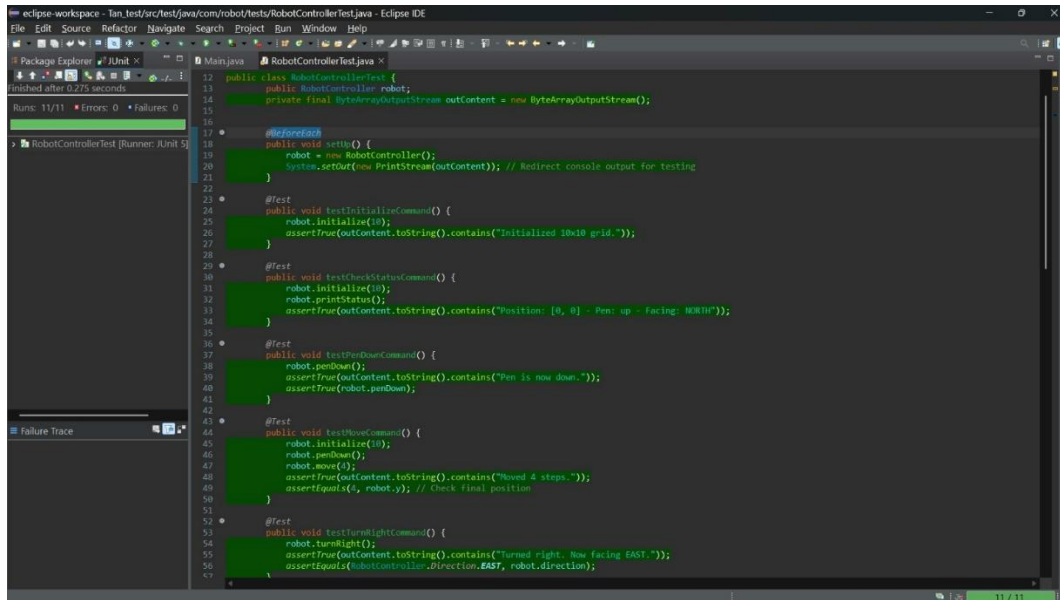
Cyclomatic Complexity for RobotController Class

Each function in RobotController has its own complexity:

- initialize(int n): 2 (one if statement)
- penUp(): 1 (no branching)
- penDown(): 1 (no branching)
- turnRight(): 5 (4 cases in switch)
- turnLeft(): 5 (4 cases in switch)
- move(int steps): 7 (loop + switch cases)
- printFloor(): 3 (one if, one nested for loop)
- printStatus(): 1 (no branching)
- printHistory(): 2 (one if condition)

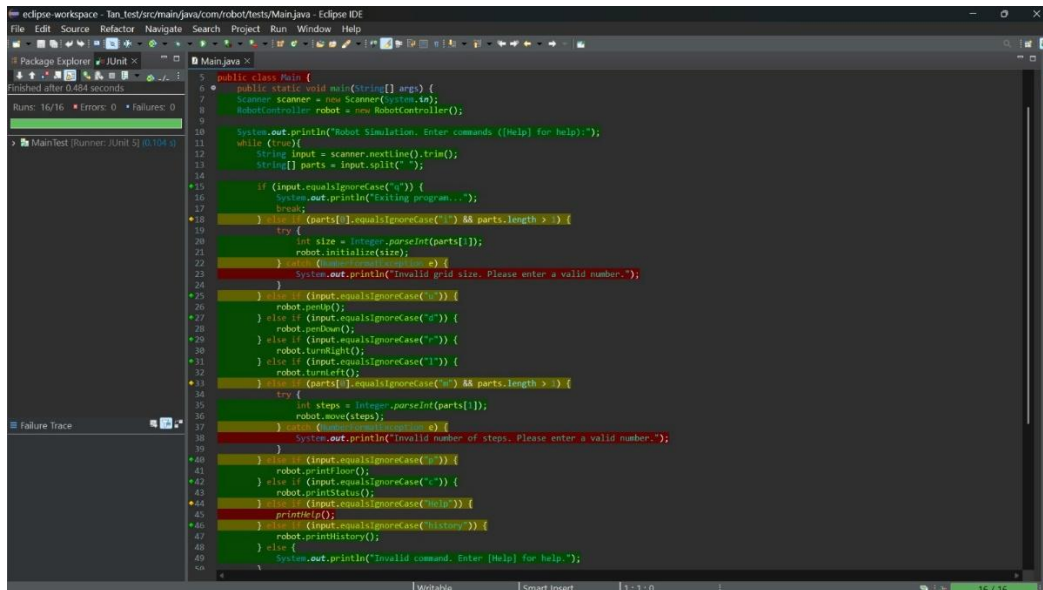
Total Complexity of RobotController: $2 + 1 + 1 + 5 + 5 + 7 + 3 + 1 + 2 = 27$

9.5 Line Coverage:



The screenshot shows the Eclipse IDE with the file `RobotControllerTest.java` open. The code is a JUnit test class for `RobotController`. The left sidebar shows the Package Explorer with `JUnit` selected. The bottom status bar indicates 11/11 lines covered, 0 errors, and 0 failures. The code includes several test methods: `setUp()`, `testInitializeCommand()`, `testCheckStatusCommand()`, `testPenDownCommand()`, `testMoveCommand()`, and `testTurnRightCommand()`. Each method is annotated with `@Test` and `@Before`. The code uses `System.out` to print status information and `assertEquals` to verify the output.

```
12 public class RobotControllerTest {
13     public RobotController robot;
14     private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
15
16     @Before
17     public void setUp() {
18         robot = new RobotController();
19         System.setOut(new PrintStream(outContent)); // Redirect console output for testing
20     }
21
22     @Test
23     public void testInitializeCommand() {
24         robot.initialize(10);
25         assertTrue(outContent.toString().contains("Initialized 10x10 grid.));
26     }
27
28     @Test
29     public void testCheckStatusCommand() {
30         robot.initialize(10);
31         robot.printStatus();
32         assertTrue(outContent.toString().contains("Position: [0, 0] - Pen: up - Facing: NORTH"));
33     }
34
35     @Test
36     public void testPenDownCommand() {
37         robot.penDown();
38         assertTrue(outContent.toString().contains("Pen is now down.));
39         assertTrue(robot.penDown());
40     }
41
42     @Test
43     public void testMoveCommand() {
44         robot.initialize(10);
45         robot.penDown();
46         robot.move(4);
47         assertTrue(outContent.toString().contains("Moved 4 steps.));
48         assertEquals(4, robot.y); // Check final position
49     }
50
51     @Test
52     public void testTurnRightCommand() {
53         robot.turnRight();
54         assertTrue(outContent.toString().contains("Turned right. Now facing EAST.));
55         assertEquals(RobotController.Direction.EAST, robot.direction);
56     }
57 }
```



The screenshot shows the Eclipse IDE with the file `Main.java` open. The code is a Java application that simulates a robot. The left sidebar shows the Package Explorer with `JUnit` selected. The bottom status bar indicates 16/16 lines covered, 0 errors, and 0 failures. The code includes a `main` method that takes command-line arguments and processes them. It uses `Scanner` to read input and `System.out` to print output. The code includes several conditional statements and loops to handle different commands.

```
1 public class Main {
2     public static void main(String[] args) {
3         Scanner scanner = new Scanner(System.in);
4         RobotController robot = new RobotController();
5
6         System.out.println("Robot Simulation. Enter commands ([help] for help.));
7         while (true) {
8             String input = scanner.nextLine().trim();
9             String[] parts = input.split(" ");
10
11             if (input.equalsIgnoreCase("h")) {
12                 System.out.println("Exiting program...");
13                 break;
14             }
15             if (parts[0].equalsIgnoreCase("s") && parts.length > 1) {
16                 try {
17                     int size = Integer.parseInt(parts[1]);
18                     robot.initialize(size);
19                 } catch (NumberFormatException e) {
20                     System.out.println("Invalid grid size. Please enter a valid number.));
21                 }
22             }
23             if (parts[0].equalsIgnoreCase("p")) {
24                 robot.penUp();
25             }
26             if (parts[0].equalsIgnoreCase("d")) {
27                 robot.penDown();
28             }
29             if (parts[0].equalsIgnoreCase("r")) {
30                 robot.turnRight();
31             }
32             if (parts[0].equalsIgnoreCase("l")) {
33                 robot.turnLeft();
34             }
35             if (parts[0].equalsIgnoreCase("m") && parts.length > 1) {
36                 try {
37                     int steps = Integer.parseInt(parts[1]);
38                     robot.move(steps);
39                 } catch (NumberFormatException e) {
40                     System.out.println("Invalid number of steps. Please enter a valid number.));
41                 }
42             }
43             if (parts[0].equalsIgnoreCase("f")) {
44                 robot.printFloor();
45             }
46             if (parts[0].equalsIgnoreCase("s")) {
47                 robot.printStatus();
48             }
49             if (parts[0].equalsIgnoreCase("h")) {
50                 printHelp();
51             }
52             if (parts[0].equalsIgnoreCase("history")) {
53                 robot.printHistory();
54             }
55             else {
56                 System.out.println("Invalid command. Enter [help] for help.));
57             }
58         }
59     }
60 }
```

10. Conclusion

In this project we have successfully developed and verified a robot control system that can follow commands like moving, turning, toggling the pen, and displaying the grid state. Through testing, we ensured the robot accurately updates its position, keeps the pen functionality in check, and properly reflects changes on the grid. The testing process confirmed that the system works reliably and correctly in different scenarios. This project highlights the use of test cases in ensuring accuracy, while also providing a strong base for future improvements.

11. AI Transparency

- **Used AI tools and details of the information gathered :**
We used AI tools to get some knowledge about our project and get some information about tests.
- **Tools used:**
 - 1) ChatGPT
 - 2) Deep seek

Prompts Used for Research:

- 1) How to resolve class not loading error in IntelliJ?
- 2) What should a help command contain for a command-driven application: exception handling?
- 3) What are the essential components of a system initialization function?
- 4) What are effective ways to manage invalid user input in a command-line application?
- 5) How can I provide meaningful feedback for invalid command: code?
- 6) How can I implement a drawing function using a pen-down command in a virtual environment?
- 7) How to generate User stories based on requirements?
- 8) I am making a robot controller testing project, can you give me an introduction and conclusion paragraph ?
- 9) I can structure file in Eclipse, but Main class does not load in IntelliJ, what are the changes I need to make to my file for code to run smoothly?
- 10) Can you create flowcharts for path coverage, condition and statement Coverage?
- 11) Code for Jacoco Plugin for testing in pom.xml?

Contribution Sheet:

- 11.1 **Harshilsinh Solanki:** Code development: command parsing and Documentation, Functional Coverage.
- 11.2 **Vraj Shah:** Development: Function generation and execution, Line Coverage.
- 11.3 **Teja Sayila(Scrum Master):** Test case execution, verification and Documentation, Path Coverage.
- 11.4 **Tanisha Fonseca:** Unit Test case development, verification and Documentation, Conditional and branch Coverage
(Note: Scrum master would be rotated for each task)