THE DROID’S MOTION DELIVERABLE – 1

1st Hari Vinayak Sai Kandikattu
Department of Electrical and Computer Engineering
Concordia University
40219582

2nd Akash Rajmohan
Department of Electrical and Computer Engineering
Concordia University
40218469

3rd Karthini Rajasekharan
Department of Electrical and Computer Engineering
Concordia University
40238556

4th Manish Pejathaya
Department of Electrical and Computer Engineering
Concordia University
40194909

***Abstract*—The main objective of this project is to design a robot that can stroll around a room. In this project, the robot is designed using a Java program, and is anticipated to print the pattern and trace its course in accordance with the user’s instructions. Each of the function is tested separately using unit testing and has its own set of functional requirements.**

Repository URL : https://github.com/CURepo/COEN6761_DROID-s_Motion.git

I. INTRODUCTION

The project aims to develop a robot, using Java programming language that is capable of navigating within a room. The project is designed in such a way that, that robot can hold the pen in two positions, mainly – up and down. The conditions say that when the pen is in the up position, the robot can move around and does not trace any paths. While the pen in down position, the robot can trace its path. The instructions are given by the user and the robot can navigate accordingly. The robot is positioned at [0,0] with a floor size of N-by-N array, with the pen up, facing north. Section 2 depicts the picture of the use case diagram for the motion of the droid, along with an explanation, Section 3 lists out the various functional requirements, each with unique identifier. We have added the screenshots of all the functions in Section 4. Section 5 displays a table linking the unit test cases and its corresponding requirements with the screenshots. Section 6 tells us about the GUI interface that we have used to display our output using JavaFX. Finally, section 7, a table with the test case results and conclusions. The GitHub URL has been attached to view the unique development and test code.

II. USE CASE DIAGRAM

The droid’s motion is depicted in the form of a use case diagram with all the constraints given. The functions like move pen up to walk without any traces, move pen down to trace the paths, turn left, turn right, depicting the path traced, terminating the program, and printing the current position, which includes the cell, the direction, and the pen status is mentioned.

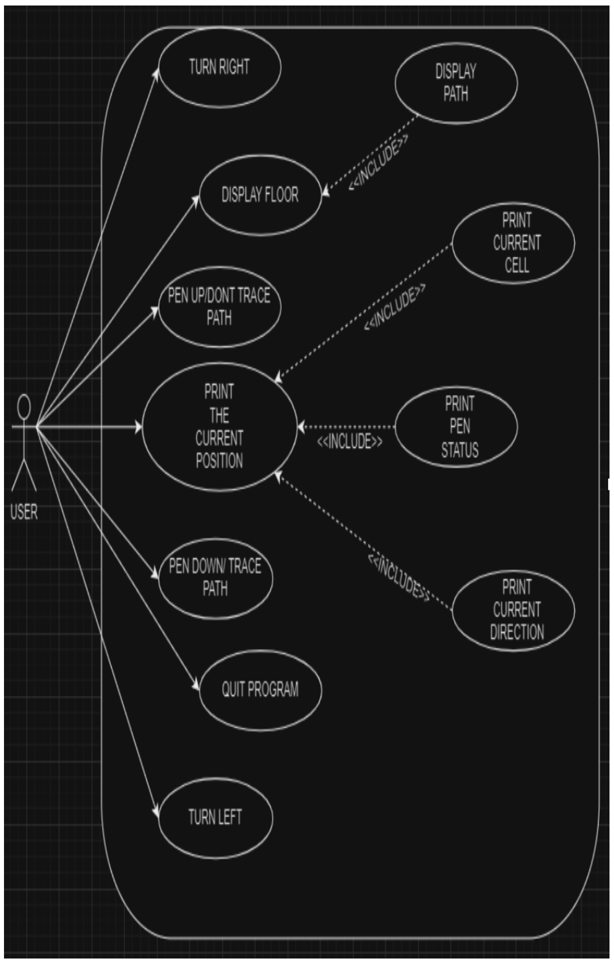


Fig. 1. **Use Case Diagram**

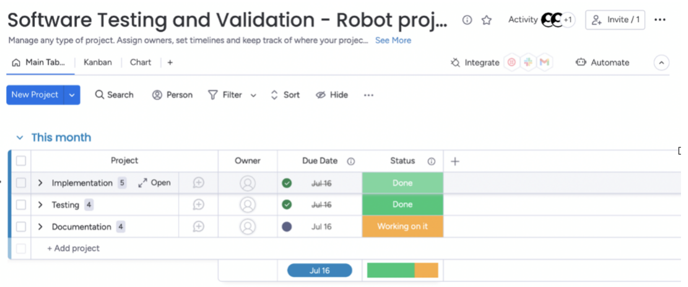
III. FUNCTIONAL REQUIREMENTS

Below is the table that shows the functional requirements of each step.

R	Functional Requirements
R1	User instruction to robot to move the pen down
R2	User instruction to robot to move the pen up
R3	User instruction to a west facing robot to turn right and face north
R4	User instruction to a south facing robot to turn right and face west
R5	User instruction to an east facing robot to turn right and face south
R6	User instruction to a north facing robot to turn right and face east
R7	User instruction to an east facing robot to turn left and face north
R8	User instruction to a south facing robot to turn left and face east
R9	User instruction to a west facing robot to turn left and face south
R10	User instruction to a north facing robot to turn left and face west
R11	User instruction to robot to trace n steps with pen down and facing north
R12	User instruction to robot to trace n steps with pen down and facing east
R13	User instruction to robot to trace n steps with pen down and facing south
R14	User instruction to robot to trace n steps with pen down and facing west
R15	User instruction to robot to not to trace n steps with pen up and facing north
R16	User instruction to robot to not to trace n steps with pen up and facing east
R17	User instruction to robot to not to trace n steps with pen up and facing south
R18	User instruction to robot to not to trace n steps with pen up and facing west
R19	Upon user request the robot is anticipated to trace its path
R20	Upon user request the robot is anticipated to print its current position
R21	Set initial position to x=0, y=0, while facing north and pen up
R22	With the command Q, user can quit the program

A. *USER STORIES AND EFFORT TRACKING*

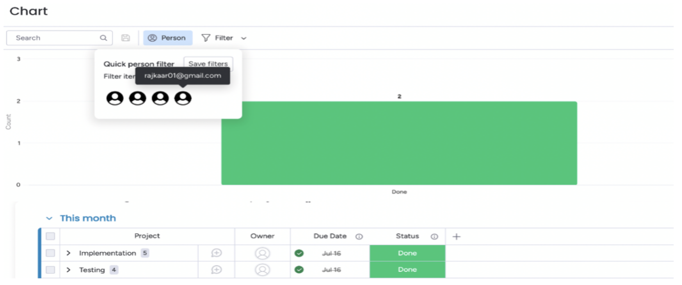
We used https://monday.com, effort tracking and project management. The tool simplified our effort tracking through task allocation, time tracking, visual dashboards, automation, collaboration, and real-time updates. It enhances productivity and project success. We divided the user requirements into 5-implementation stories and 4-unit testing stories. Each user story was assigned to a team member. The validation and implementation stories were divided in such a way that every team member gets to work both in the development and validation part of the project. The link to our work is attached below: https://ashakash85s-team.monday.com/boards/4814848928/


Software Testing and Validation - Robot proj...

Manage any type of project. Assign owners, set timelines and keep track of where your projec... See More

Project	Owner	Due Date	Status
Implementation 5		~~Jul 16~~	Done
Testing 4		~~Jul 16~~	Done
Documentation 4		Jul 16	Working on it
+ Add project			
		Jul 16	

Fig. 2. **Effort Tracking of Project**



Project	Owner	Due Date	Status
Implementation 5		~~Jul 16~~	Done
Testing 4		~~Jul 16~~	Done

Fig. 3. **Project Progress Chart**

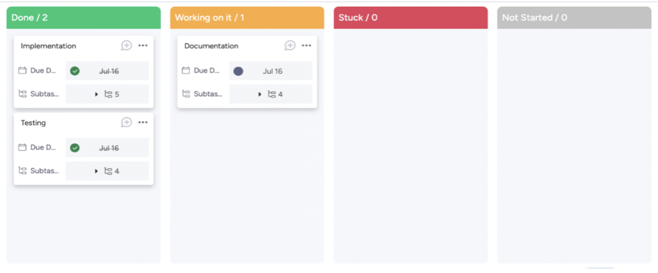


Fig. 4. **Kanban View of the Project**

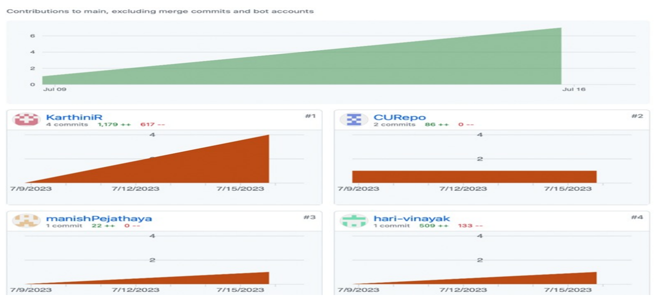
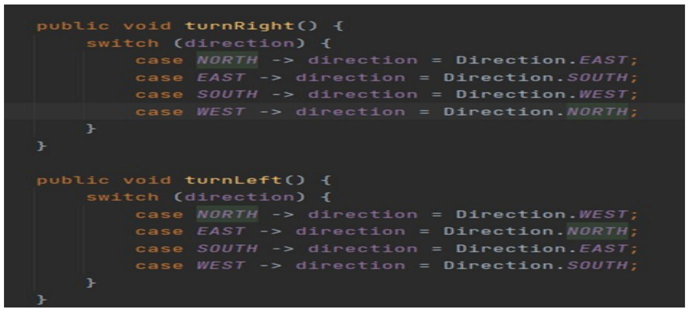


Fig. 5. **GitHub Contributors Work**

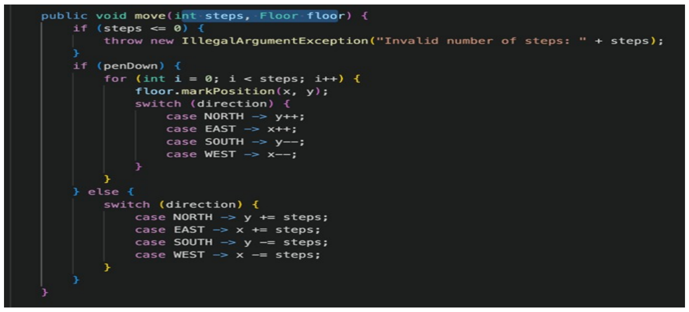
IV. SCREENSHOTS OF FUNCTIONS

The screenshots of functions are attached below for reference.



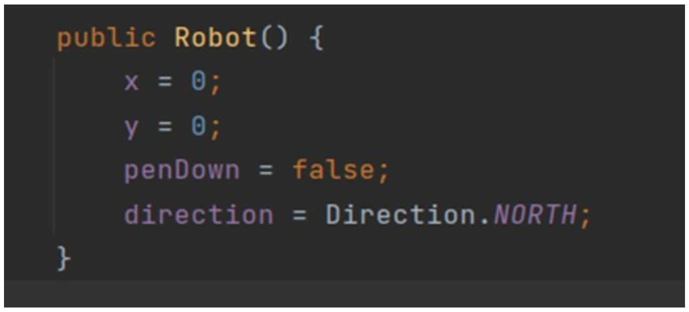
```java
public void turnRight() {
    switch (direction) {
        case NORTH -> direction = Direction.EAST;
        case EAST -> direction = Direction.SOUTH;
        case SOUTH -> direction = Direction.WEST;
        case WEST -> direction = Direction.NORTH;
    }
}

public void turnLeft() {
    switch (direction) {
        case NORTH -> direction = Direction.WEST;
        case EAST -> direction = Direction.NORTH;
        case SOUTH -> direction = Direction.EAST;
        case WEST -> direction = Direction.SOUTH;
    }
}
```

Fig. 6. **Function for move right and move left**



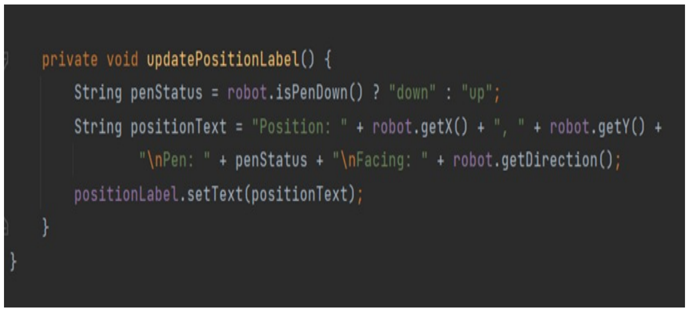
```java
public void move(int steps, Floor floor) {
    if (steps <= 0) {
        throw new IllegalArgumentException("Invalid number of steps: " + steps);
    }
    if (penDown) {
        for (int i = 0; i < steps; i++) {
            floor.markPosition(x, y);
            switch (direction) {
                case NORTH -> y++;
                case EAST -> x++;
                case SOUTH -> y--;
                case WEST -> x--;
            }
        }
    } else {
        switch (direction) {
            case NORTH -> y += steps;
            case EAST -> x += steps;
            case SOUTH -> y -= steps;
            case WEST -> x -= steps;
        }
    }
}
```

Fig. 7. **Function for robot movement**



```java
public Robot() {
    x = 0;
    y = 0;
    penDown = false;
    direction = Direction.NORTH;
}
```

Fig. 8. **Function for initialization**



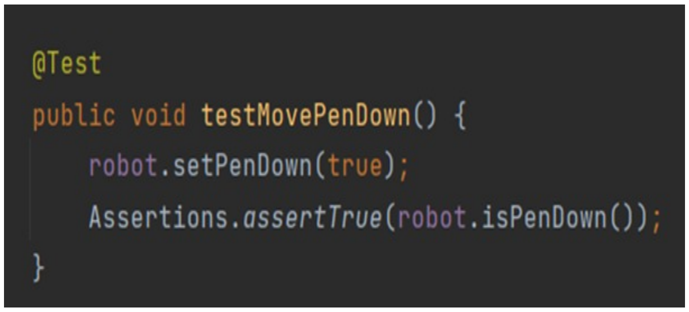
```java
private void updatePositionLabel() {
    String penStatus = robot.isPenDown() ? "down" : "up";
    String positionText = "Position: " + robot.getX() + ", " + robot.getY() +
            "\nPen: " + penStatus + "\nFacing: " + robot.getDirection();
    positionLabel.setText(positionText);
  }
}
```

Fig. 9. **Function for robot position**

V. Unit Test Cases with their Corresponding Functional Requirements

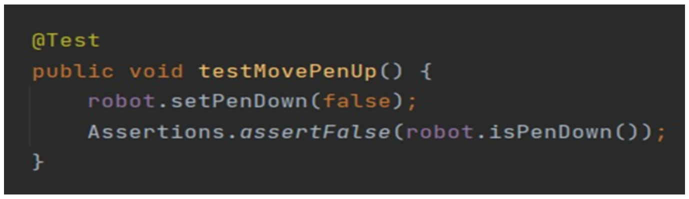
R	**Unit Test Cases (UTC)**	**Functional Requirements (F)**
1	UTC1	R1
2	UTC2	R2
3	UTC3	R3
4	UTC4	R4
5	UTC5	R5
6	UTC6	R6
7	UTC7	R7
8	UTC8	R8
9	UTC9	R9
10	UTC10	R10
11	UTC11	R11
12	UTC12	R12
13	UTC13	R13
14	UTC14	R14
15	UTC15	R15
16	UTC16	R16
17	UTC17	R17
18	UTC18	R18
19	UTC19	R19
20	UTC20	R20
21	UTC21	R21
22	UTC22	R22

Below are the screenshots of all the unit test cases that passed successfully.



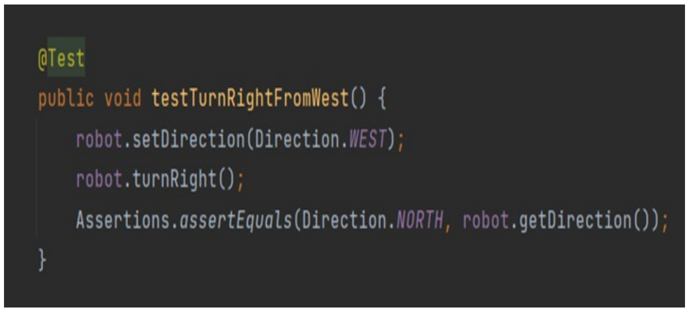
```java
@Test
public void testMovePenDown() {
    robot.setPenDown(true);
    Assertions.assertTrue(robot.isPenDown());
}
```

Fig. 10. **UTC1 for Functional Requirement R1 – to move pen down**



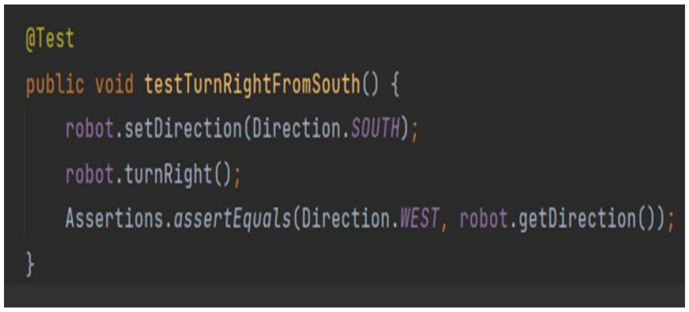
```java
@Test
public void testMovePenUp() {
    robot.setPenDown(false);
    Assertions.assertFalse(robot.isPenDown());
}
```

Fig. 11. **UTC2 for Functional Requirement 2 – to move pen up**



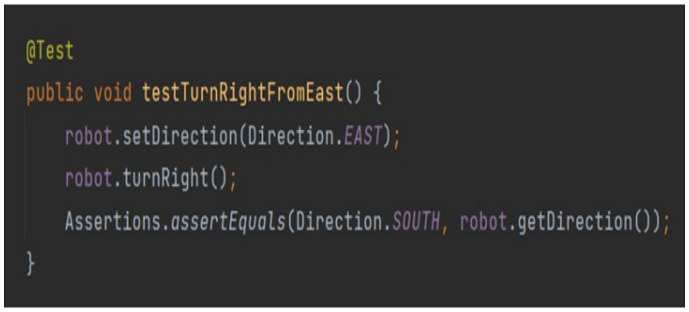
```
@Test
public void testTurnRightFromWest() {
    robot.setDirection(Direction.WEST);
    robot.turnRight();
    Assertions.assertEquals(Direction.NORTH, robot.getDirection());
}
```

Fig. 12. **UTC3 for Functional Requirement 3**



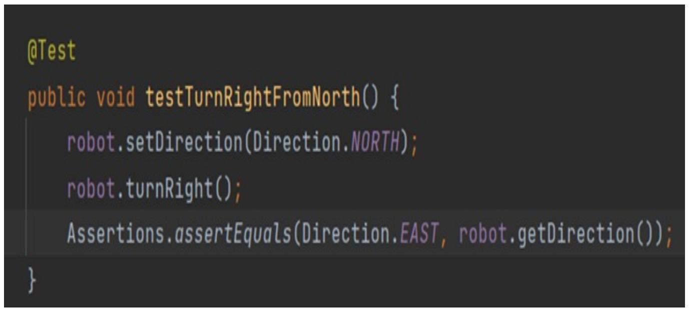
```
@Test
public void testTurnRightFromSouth() {
    robot.setDirection(Direction.SOUTH);
    robot.turnRight();
    Assertions.assertEquals(Direction.WEST, robot.getDirection());
}
```

Fig. 13. **UTC4 for Functional Requirement 4**



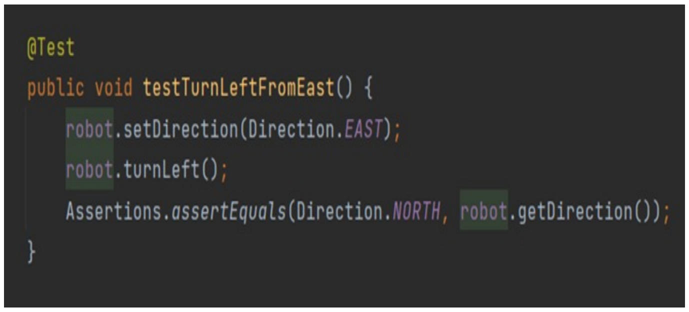
```
@Test
public void testTurnRightFromEast() {
    robot.setDirection(Direction.EAST);
    robot.turnRight();
    Assertions.assertEquals(Direction.SOUTH, robot.getDirection());
}
```

Fig. 14. **UTC5 for Functional Requirement 5**



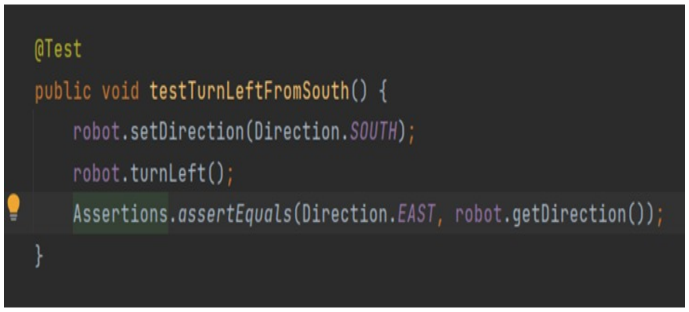
```
@Test
public void testTurnRightFromNorth() {
    robot.setDirection(Direction.NORTH);
    robot.turnRight();
    Assertions.assertEquals(Direction.EAST, robot.getDirection());
}
```

Fig. 15. **UTC6 for Functional Requirement 6**



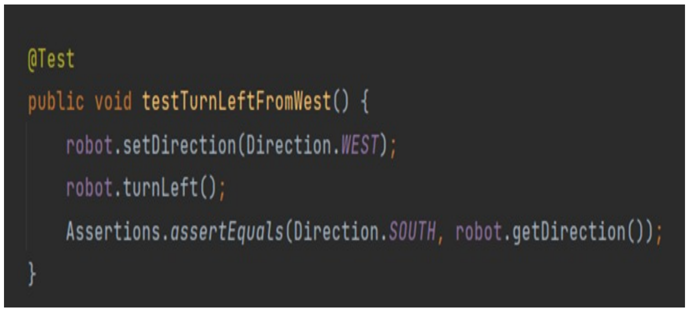
```
@Test
public void testTurnLeftFromEast() {
    robot.setDirection(Direction.EAST);
    robot.turnLeft();
    Assertions.assertEquals(Direction.NORTH, robot.getDirection());
}
```

Fig. 16. **UTC7 for Functional Requirement 7**



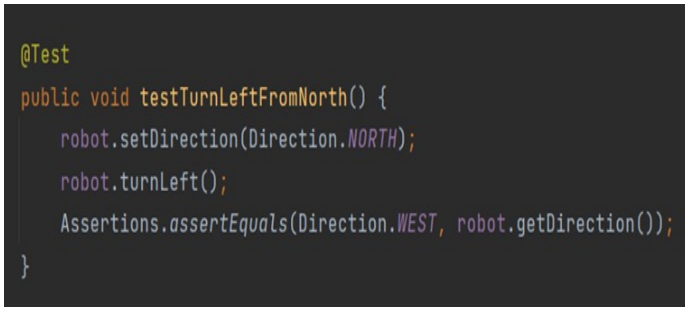
```
@Test
public void testTurnLeftFromSouth() {
    robot.setDirection(Direction.SOUTH);
    robot.turnLeft();
    Assertions.assertEquals(Direction.EAST, robot.getDirection());
}
```

Fig. 17. **UTC8 for Functional Requirement 8**



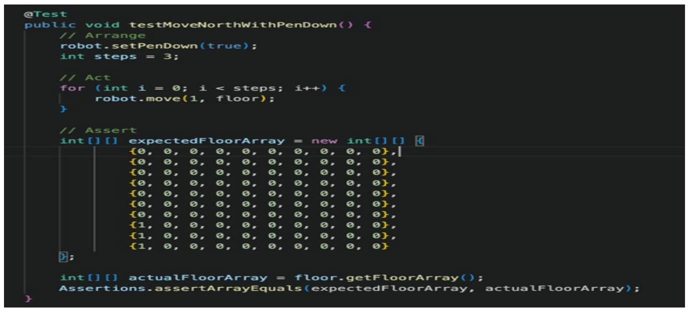
```
@Test
public void testTurnLeftFromWest() {
    robot.setDirection(Direction.WEST);
    robot.turnLeft();
    Assertions.assertEquals(Direction.SOUTH, robot.getDirection());
}
```

Fig. 18. **UTC9 for Functional Requirement 9**



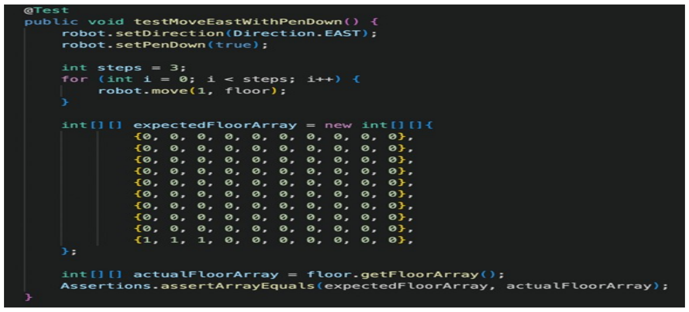
```
@Test
public void testTurnLeftFromNorth() {
    robot.setDirection(Direction.NORTH);
    robot.turnLeft();
    Assertions.assertEquals(Direction.WEST, robot.getDirection());
}
```

Fig. 19. **UTC10 for Functional Requirement 10**



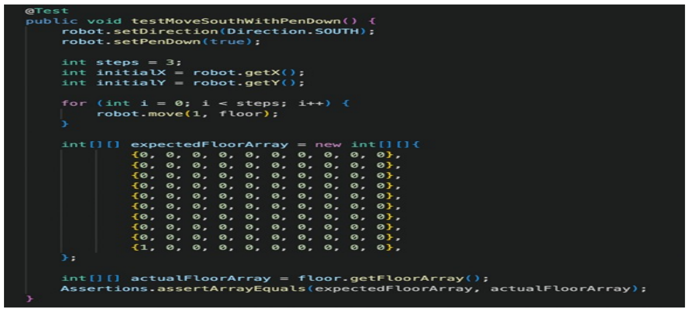
```java
@Test
public void testMoveNorthWithPenDown() {
    // Arrange
    robot.setPenDown(true);
    int steps = 3;

    // Act
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    // Assert
    int[][] expectedFloorArray = new int[][] {
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 20. **UTC11 for Functional Requirement 11**



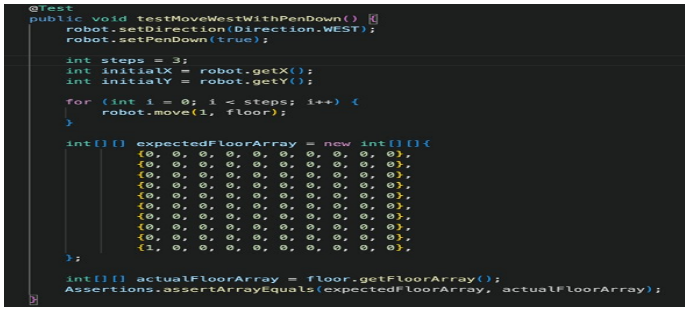
```java
@Test
public void testMoveEastWithPenDown() {
    robot.setDirection(Direction.EAST);
    robot.setPenDown(true);

    int steps = 3;
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 1, 1, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 21. **UTC12 for Functional Requirement 12**



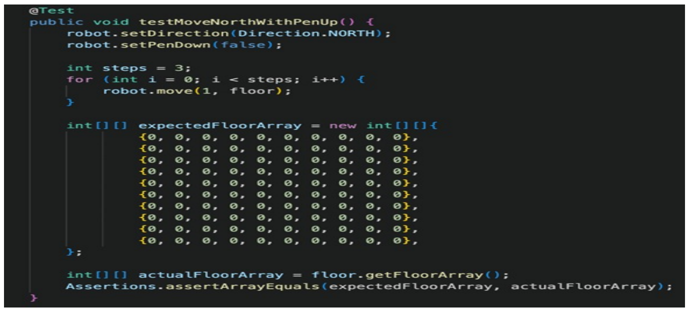
```java
@Test
public void testMoveSouthWithPenDown() {
    robot.setDirection(Direction.SOUTH);
    robot.setPenDown(true);

    int steps = 3;
    int initialX = robot.getX();
    int initialY = robot.getY();

    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 22. **UTC13 for Functional Requirement 13**



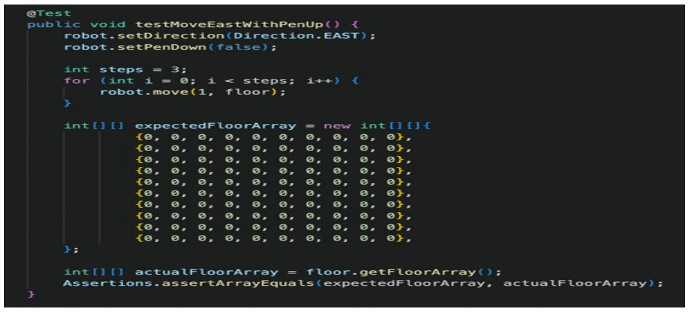
```java
@Test
public void testMoveWestWithPenDown() {
    robot.setDirection(Direction.WEST);
    robot.setPenDown(true);

    int steps = 3;
    int initialX = robot.getX();
    int initialY = robot.getY();

    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 23. **UTC14 for Functional Requirement 14**



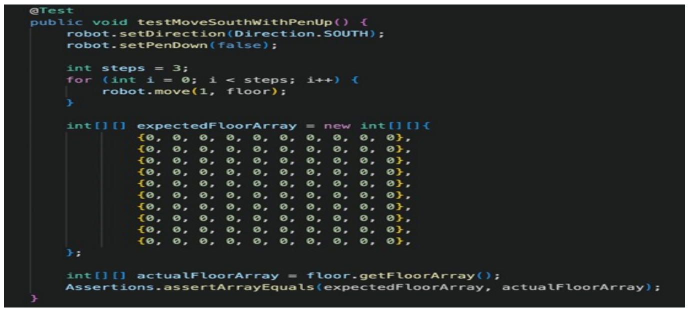
```java
@Test
public void testMoveNorthWithPenUp() {
    robot.setDirection(Direction.NORTH);
    robot.setPenDown(false);

    int steps = 3;
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 24. **UTC15 for Functional Requirement 15**



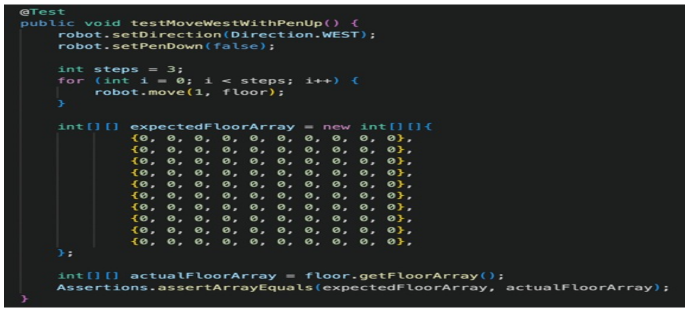
```java
@Test
public void testMoveEastWithPenUp() {
    robot.setDirection(Direction.EAST);
    robot.setPenDown(false);

    int steps = 3;
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 25. **UTC16 for Functional Requirement 16**



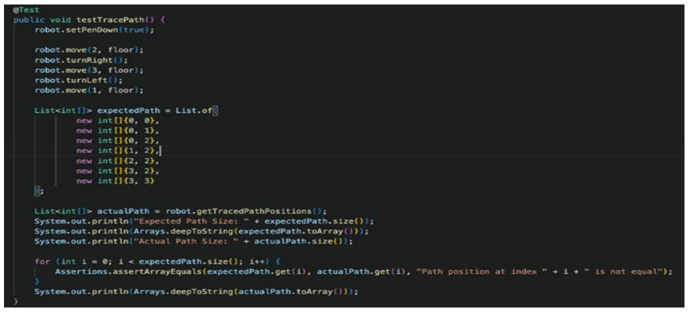
```java
@Test
public void testMoveSouthWithPenUp() {
    robot.setDirection(Direction.SOUTH);
    robot.setPenDown(false);

    int steps = 3;
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 26. **UTC17 for Functional Requirement 17**



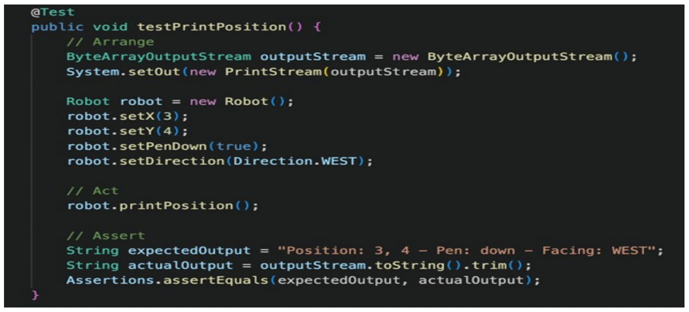
```java
@Test
public void testMoveWestWithPenUp() {
    robot.setDirection(Direction.WEST);
    robot.setPenDown(false);

    int steps = 3;
    for (int i = 0; i < steps; i++) {
        robot.move(1, floor);
    }

    int[][] expectedFloorArray = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    };

    int[][] actualFloorArray = floor.getFloorArray();
    Assertions.assertArrayEquals(expectedFloorArray, actualFloorArray);
}
```

Fig. 27. **UTC18 for Functional Requirement 18**



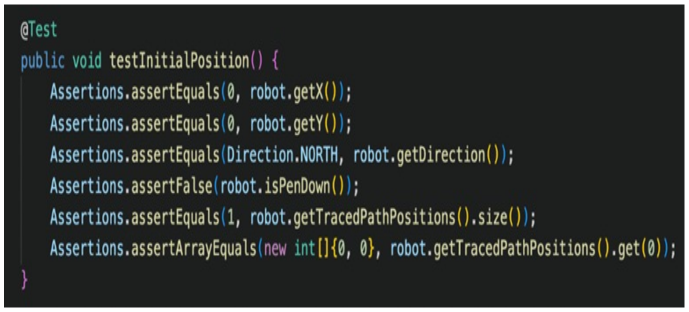
```java
@Test
public void testTracePath() {
    robot.setPenDown(true);

    robot.move(2, floor);
    robot.turnRight();
    robot.move(3, floor);
    robot.turnLeft();
    robot.move(1, floor);

    List<int[]> expectedPath = List.of(
            new int[]{0, 0},
            new int[]{0, 1},
            new int[]{0, 2},
            new int[]{1, 2},
            new int[]{2, 2},
            new int[]{3, 2},
            new int[]{3, 3}
    );

    List<int[]> actualPath = robot.getTracedPathPositions();
    System.out.println("Expected Path Size: " + expectedPath.size());
    System.out.println(Arrays.deepToString(expectedPath.toArray()));
    System.out.println("Actual Path Size: " + actualPath.size());

    for (int i = 0; i < expectedPath.size(); i++) {
        Assertions.assertArrayEquals(expectedPath.get(i), actualPath.get(i), "Path position at index " + i + " is not equal");
    }
    System.out.println(Arrays.deepToString(actualPath.toArray()));
}
```

Fig. 28. **UTC19 for Functional Requirement 19**



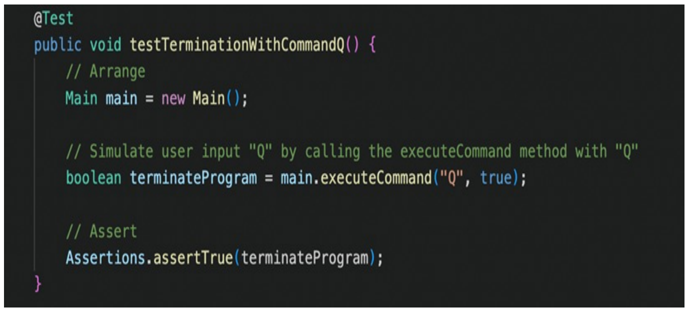
```java
@Test
public void testPrintPosition() {
    // Arrange
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outputStream));

    Robot robot = new Robot();
    robot.setX(3);
    robot.setY(4);
    robot.setPenDown(true);
    robot.setDirection(Direction.WEST);

    // Act
    robot.printPosition();

    // Assert
    String expectedOutput = "Position: 3, 4 - Pen: down - Facing: WEST";
    String actualOutput = outputStream.toString().trim();
    Assertions.assertEquals(expectedOutput, actualOutput);
}
```

Fig. 29. **UTC20 for Functional Requirement 20**



```java
@Test
public void testInitialPosition() {
    Assertions.assertEquals(0, robot.getX());
    Assertions.assertEquals(0, robot.getY());
    Assertions.assertEquals(Direction.NORTH, robot.getDirection());
    Assertions.assertFalse(robot.isPenDown());
    Assertions.assertEquals(1, robot.getTracedPathPositions().size());
    Assertions.assertArrayEquals(new int[]{0, 0}, robot.getTracedPathPositions().get(0));
}
```

Fig. 30. **UTC21 for Functional Requirement 21**



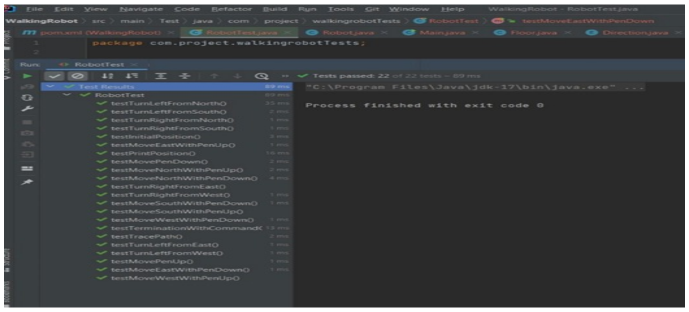
```java
@Test
public void testTerminationWithCommandQ() {
    // Arrange
    Main main = new Main();

    // Simulate user input "Q" by calling the executeCommand method with "Q"
    boolean terminateProgram = main.executeCommand("Q", true);

    // Assert
    Assertions.assertTrue(terminateProgram);
}
```

Fig. 31. **UTC22 for Functional Requirement 22**

VI. UNIT TEST CASE TABLE

The table below conveys the unit test case results for the corresponding functional requirements.

R	Unit Test Cases (UTC)	Functional Requirements (F)	Unit Test Case Results
1	UTC1	R1	Pass
2	UTC2	R2	Pass
3	UTC3	R3	Pass
4	UTC4	R4	Pass
5	UTC5	R5	Pass
6	UTC6	R6	Pass
7	UTC7	R7	Pass
8	UTC8	R8	Pass
9	UTC9	R9	Pass
10	UTC10	R10	Pass
11	UTC11	R11	Pass
12	UTC12	R12	Pass
13	UTC13	R13	Pass
14	UTC14	R14	Pass
15	UTC15	R15	Pass
16	UTC16	R16	Pass
17	UTC17	R17	Pass
18	UTC18	R18	Pass
19	UTC19	R19	Pass
20	UTC20	R20	Pass
21	UTC21	R21	Pass
22	UTC22	R22	Pass



File Edit View Navigate Code Refactor Build Run Tools Git Window Help
WalkingRobot - RobotTest.java
WalkingRobot > src > main > Test > java > com > project > walkingrobotTests > RobotTest > testMoveEastWithPenDown
pom.xml (WalkingRobot) RobotTest.java Robot.java Main.java Floor.java Direction.java
Run: RobotTest

```
package com.project.walkingrobotTests;
```

```
Tests passed: 22 of 22 tests – 89 ms

Test Results                              89 ms
  RobotTest                               89 ms
    testTurnLeftFromNorth()               35 ms
    testTurnLeftFromSouth()                2 ms
    testTurnRightFromNorth()               1 ms
    testTurnRightFromSouth()               1 ms
    testInitialPosition()                  3 ms
    testMoveEastWithPenUp()                1 ms
    testPrintPosition()                   16 ms
    testMovePenDown()                      2 ms
    testMoveNorthWithPenUp()               2 ms
    testMoveNorthWithPenDown()             4 ms
    testTurnRightFromEast()
    testTurnRightFromWest()                1 ms
    testMoveSouthWithPenDown()             1 ms
    testMoveSouthWithPenUp()
    testMoveWestWithPenDown()              1 ms
    testTerminationWithCommandK           13 ms
    testTracePath()                        2 ms
    testTurnLeftFromEast()                 1 ms
    testTurnLeftFromWest()                 1 ms
    testMovePenUp()                        1 ms
    testMoveEastWithPenDown()              1 ms
    testMoveWestWithPenUp()

"C:\Program Files\Java\jdk-17\bin\java.exe" ...

Process finished with exit code 0
```

Fig. 32. **Test cases that are run successfully without any errors**

VII. GUI INTERFACE AND JAVAFX

JavaFX is a Java framework for generating visually appealing graphical user interfaces (GUIs). It has cross-platform compatibility, rich UI controls, CSS style, media support, animation capabilities, layout management, and seamless connection with the Java environment. It is open-source and has a vibrant developer community. JavaFX supports the responsive design technique, which allows developers to construct user interfaces that adapt and adjust to multiple screen sizes and resolutions. This ensures that JavaFX applications deliver a consistent and optimal user experience across a wide range of platforms, including desktops, laptops, tablets, and mobile phones.

Please refer to the README file in our GitHub repository to view the complete guidance on running our code using JavaFX. https://github.com/CURepo/COEN6761_DROID-s_Motion.git

VIII. OUTPUT OF THE GUI

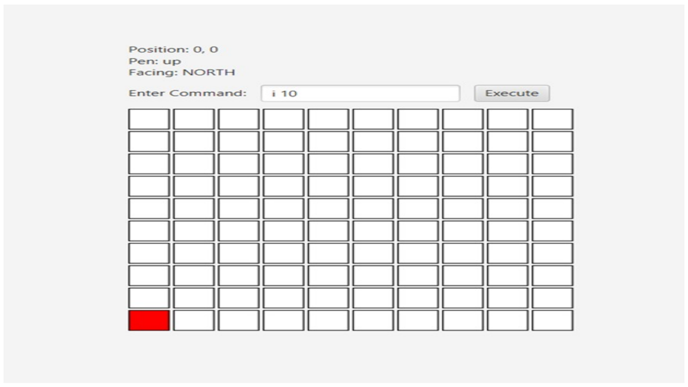


Fig. 33. **Robot Simulator – I10 is the command to initialize the system into a 10-by-10 array. The red square indicates the robot. The robot is now at [0, 0], pen up, pointed north, and the values on the array floor have all been set to zeros**

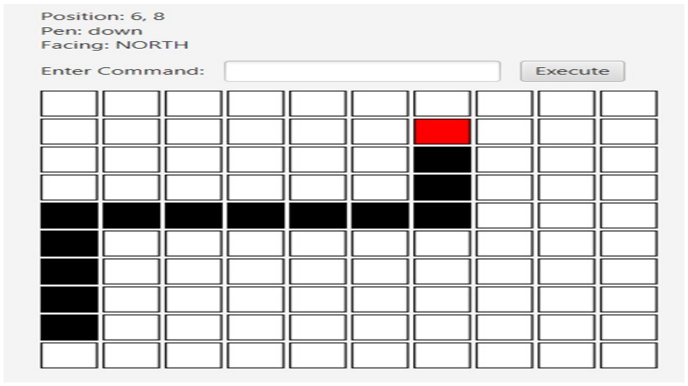


Fig. 34. **Trace of the robot when the pen is facing down**

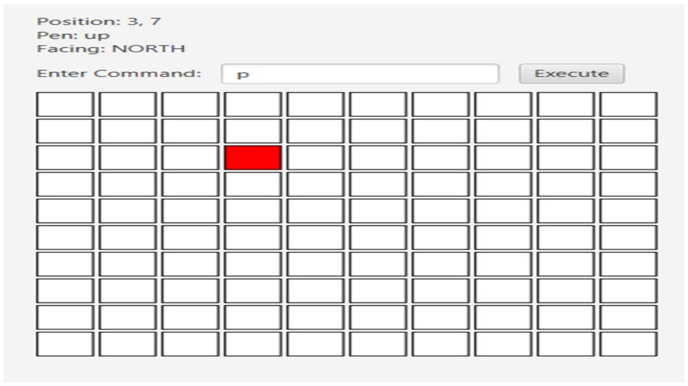


Fig. 35. **No Trace of the robot is found because the pen is facing up**

IX. CONCLUSION

In this report, we have done unit testing for 22 functional requirements and the output runs out successfully without any errors. Our works have been recorded in GitHub and our project time tracking is done using https://monday.com, a useful app that records the progress of our work. All the screenshots are attached in the report for reference.