DELIVERABLE – 3

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*—In this report, being the Quality Assurance team, we have performed several testing methods for the Development team. We have established new unit test cases and have reported the same. We have used JaCoCo coverage tool to produce the coverage results. In addition to this, we have done mutation testing and data flow testing for which the screenshots of the results are attached. Our GitHub repository URL has been attached here. Our GitHub repository URL has been attached here.**
Repository URL : https://github.com/CURepo/QAteam_Dev6.git

I. INTRODUCTION

We have completed various kinds of testing in this report for the Development team 5.

Unit testing: Testing with various input values for the robot command function and each individual class of the command class to verify it proper working.

Black-box testing: Program testing is performed without having any internal working knowledge, just by giving the input and checking for the output. The action involves the verification of the expected output of the program for the corresponding variety of inputs used.

White-box testing: In Whitebox testing, the program is tested based on internal working. This involves code analysis, potential edge case identification, and boundary condition.

Manual testing: Manual testing checks that the program's output reflects the desired behaviour by entering commands manually. This can help to figure out any unexpected behaviour that automated testing might not have shown.

Mutation Testing: It is a technique that allows us to assess the adequacy of a test set using mutant programs. Mutation is terminated when the test case is failed, and mutation is continued when test cases are passed.

Data Flow Testing: A specific kind of structural testing is data flow testing. It is a technique for locating a program's test processes in accordance with where variables are defined and utilized. It's unrelated to data flow diagrams. It is concerned with statements that assign values to variables and statements that make use of or refer to those values.

II. BLACK BOX TESTING

1) **Command I 10**: The system should be initialized as a 10 x 10 array with all elements set to zero. Moreover, the robot should be positioned at [0, 0], with the pen up and facing north
2) **Command C**: Position: (0, 0), Pen: up, facing: north The program's output provides information about the robot's position and whether the pen is currently up or down
3) **Enter command: D** Now, the user commands the robot to lower the pen
4) **Command M 6**: The user instructs the robot to move forward by 6 cells, resulting in the robot's new position being [0, 6]
5) **Enter Command: C**: Position: (0, 6), Pen: down, Facing: north
6) **Enter command: R** This command orders the robot to turn right from its current position
7) **Enter command: C** Position: (0, 6), Pen: down, Facing: east
8) **Enter command: M 7** The user instructs the robot to move forward by 7 cells, resulting in the robot's new position being [7, 6]
9) **Enter command: C** Position: (7, 6), Pen: down, Facing: east
10) **Enter command: P** Prints the position matrix, as shown in Figure 1.

```
**************************************************
Enter your choice:c
Position:7,6 - Pen:Down - Facing:East

********* Welcome to the Console Menu *********
1. To initialize the room enter I followed by an integer:
2. To Print the current position of the robot enter C:
3. To pen up enter U:
4. To pen down enter D:
5. To Turn right enter R:
6. To Turn left enter L:
7. To move the robot forward enter M followed by an integer:
8. To print the room enter P:
9. To Stop the program enter Q:
**************************************************
Enter your choice:p
0 0 0 0 0 0 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 1 1 1 1 1 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
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
```

Fig. 1. **Result of Black Box Testing**

III. CORNER CASES

The below figure shows the position as (0,0), Pen is facing down and facing the west direction. From this position we try to move the Robot westwards by 2 blocks, by using the command “M 2”. The code works partially correct but gives some unnecessary outputs as shown in Figure 2.

```
************************************************
Enter your choice:c
Position:0,0 - Pen:Down - Facing:West

********* Welcome to the Console Menu *********
1. To initialize the room enter I followed by an integer:
2. To Print the current position of the robot enter C:
3. To pen up enter U:
4. To pen down enter D:
5. To Turn right enter R:
6. To Turn left enter L:
7. To move the robot forward enter M followed by an integer:
8. To print the room enter P:
9. To Stop the program enter Q:
************************************************
Enter your choice:m 2
0
-1
******Oppps, Cant go further change the direction.******
```

Fig. 2. **Figure 2**

```
********* Welcome to the Console Menu *********
1. To initialize the room enter I followed by an integer:
2. To Print the current position of the robot enter C:
3. To pen up enter U:
4. To pen down enter D:
5. To Turn right enter R:
6. To Turn left enter L:
7. To move the robot forward enter M followed by an integer:
8. To print the room enter P:
9. To Stop the program enter Q:
************************************************
Enter your choice:m
Invalid input format. Please provide a valid number.
```

Fig. 3. **Figure 3**

When command “M” is given in invalid fashion the code responds properly, as shown in Figure 3.

IV. TEST CASES DEVELOPED

We have established 22 Unit Test Cases (UTC) for the source code of the development team, out of which 3 failed and the rest are all passed successfully.

All screenshots of successful and failed Unit Test Cases are attached below for reference.

```
@Test
void testRobotTurningRight() {
    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.East, robot.direction);

    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.South, robot.direction);

    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.West, robot.direction);

    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.North, robot.direction);
}
```

Fig. 4. **UTC 1 testRobotTurningRight**

```
@Test
public void testRobotTurningLeft() {
    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.West, robot.direction);

    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.South, robot.direction);

    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.East, robot.direction);

    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.North, robot.direction);
}
```

Fig. 5. **UTC 2 testRobotTurningLeft**

```
@Test
public void testRobotSettingPenUp() {
    robot.setPenUp();
    Assertions.assertFalse(robot.pen);
}
```

Fig. 6. **UTC 3 testRobotSettingPenUp**

```
@Test
public void testRobotSettingPenDown() {
    robot.setPenDown();
    Assertions.assertTrue(robot.pen);
}
```

Fig. 7. **UTC 4 testRobotSettingPenDown**

```java
@Test
public void testRobotInitialPosition() {
    Assertions.assertEquals( expected: 0, robot.x);
    Assertions.assertEquals( expected: 0, robot.y);
}
```

Fig. 8. **UTC 5 testRobotInitialPosition**

```java
@Test
public void testRobotSettingPosition() {
    robot.setXY( x: 3,  y: 2);
    Assertions.assertEquals( expected: 3, robot.x);
    Assertions.assertEquals( expected: 2, robot.y);
}
```

Fig. 9. **UTC 6 testRobotSettingPosition**

```java
@Test
public void testRoomConstructor() {
    int expectedSize = 5;
    Assertions.assertEquals(expectedSize, room.room.length);
    Assertions.assertEquals(expectedSize, room.room[0].length);
}
```

Fig. 10. **UTC 7 testRoomConstructor**

```java
@Test
public void testRoomPrint() {
    // Set a pattern in the room
    room.room[1][2] = 1;
    room.room[3][4] = 1;

    String expectedOutput =
            "0 0 0 0 0 \n" +
                    "0 0 1 0 0 \n" +
                    "0 0 0 0 0 \n" +
                    "0 0 0 0 1 \n" +
                    "0 0 0 0 0 \n";

    Assertions.assertEquals(expectedOutput, getRoomPrintOutput());
}
```

Fig. 11. **UTC 8 testRoomPrint**

```java
@Test
public void testRobotTurningRightAndLeft() {
    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.East, robot.direction);

    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.North, robot.direction);

    robot.turnRight();
    Assertions.assertEquals(Robot.Direction.East, robot.direction);

    robot.turnLeft();
    Assertions.assertEquals(Robot.Direction.North, robot.direction);
}
```

Fig. 12. **UTC 9 testRobotTurningRightandLeft**

```java
@Test
public void testRobotSettingPenUpAndDown() {
    robot.setPenUp();
    Assertions.assertFalse(robot.pen);

    robot.setPenDown();
    Assertions.assertTrue(robot.pen);

    robot.setPenUp();
    Assertions.assertFalse(robot.pen);
}
```

Fig. 13. **UTC 10 testRobotSettingPenUpandDown**

```java
@Test
public void testRoomConstructor_2() {
    Room room1 = new Room( size: 5);
    Assertions.assertEquals( expected: 5, room1.room.length);
    Assertions.assertEquals( expected: 5, room1.room[0].length);

    Room room2 = new Room( size: 3);
    Assertions.assertEquals( expected: 3, room2.room.length);
    Assertions.assertEquals( expected: 3, room2.room[0].length);
}
```

Fig. 14. **UTC 11 testRoomConstructor 2**

```java
@Test
public void testRoomPrint_2() {
    Room room = new Room( size: 5);

    // Pattern 1: Empty room
    String expectedOutput1 = "0 0 0 0 0 \n" +
            "0 0 0 0 0 \n" +
            "0 0 0 0 0 \n" +
            "0 0 0 0 0 \n" +
            "0 0 0 0 0 \n";
    Assertions.assertEquals(expectedOutput1, getRoomPrintOutput(room));

    // Pattern 2: Set a few cells to 1
    room.room[1][2] = 1;
    room.room[3][4] = 1;
    String expectedOutput2 = "0 0 0 0 0 \n" +
            "0 0 1 0 0 \n" +
            "0 0 0 0 0 \n" +
            "0 0 0 0 1 \n" +
            "0 0 0 0 0 \n";
    Assertions.assertEquals(expectedOutput2, getRoomPrintOutput(room));
}
```

Fig. 15. **UTC 12 testRoomPrint 2**

Fig. 16. **UTC 13 testRobotSettingPenUpandDownWithInitialDirection**

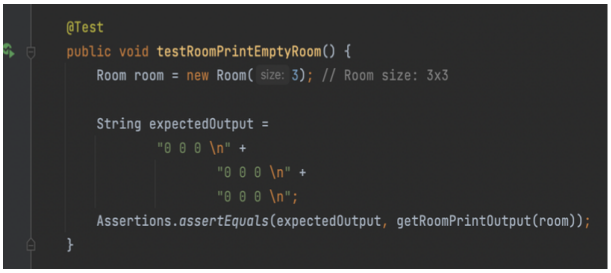


Fig. 17. **UTC 14 testRoomPrintEmptyRoom**

Fig. 18. **UTC 15 testRoomPrintNonEmptyRoom**

Fig. 19. **UTC 16 testRoomPrintNonEmptyRoomWithPenUp**

Fig. 20. **UTC 17 testRoomPrintFullRoom 2**

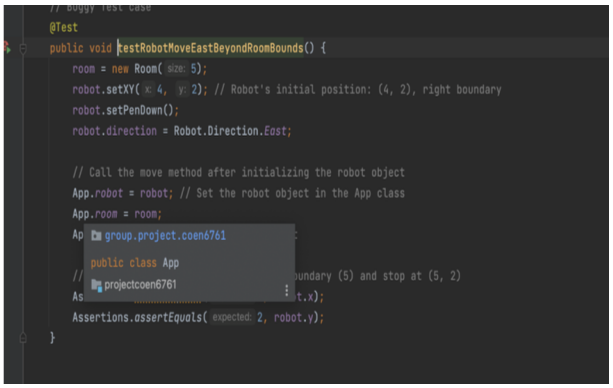


Fig. 21. **UTC 18 testRobotMoveEastBeyondRoomBounds-Failed**

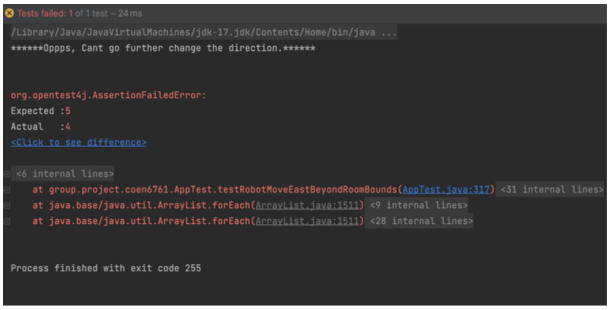


Fig. 22. **UTC 18 Failed**

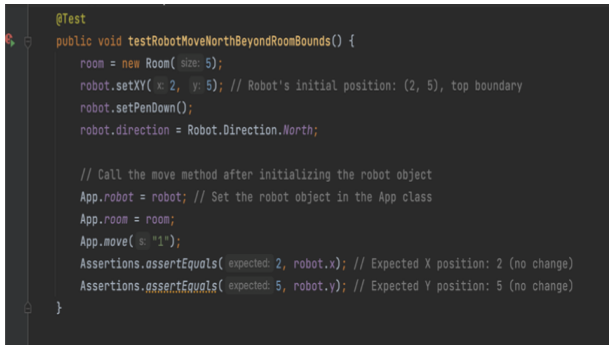


Fig. 23. **UTC 19 testRobotMoveNorthBeyondRoomBounds-Failed**

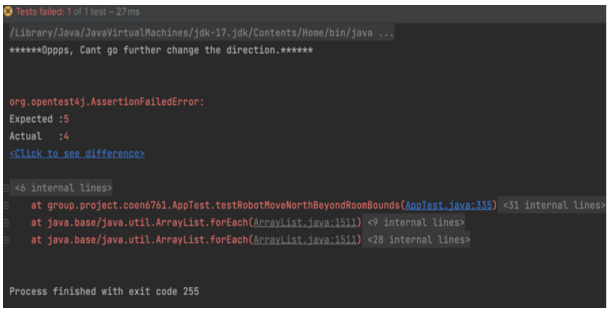


Fig. 24. **UTC 19 Failed**

```
// Buggy Test case
@Test
public void testRobotMoveSouthBeyondRoomBounds() {
    room = new Room( size: 5);
    robot.setXY( x: 2, y: 0); // Robot's initial position: (2, 0), bottom boundary
    robot.setPenDown();
    robot.direction = Robot.Direction.South;

    // Call the move method after initializing the robot object
    App.robot = robot; // Set the robot object in the App class
    App.room = room;
    App.move( s: "1");
    Assertions.assertEquals( expected: 2, robot.x); // Expected X position: 2 (no change)
    Assertions.assertEquals( expected: 0, robot.y); // Expected Y position: 0 (no change)
}
```

Fig. 25. **UTC 20 testRobotMoveSouthBeyondRoomBounds-Failed**

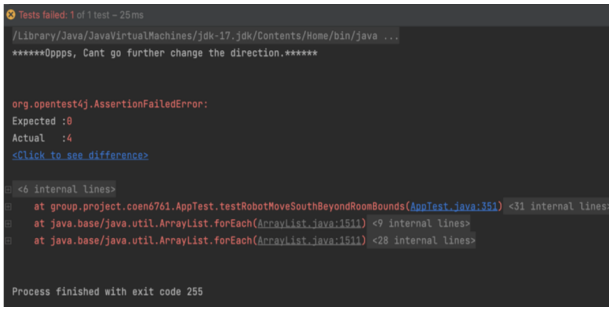


Fig. 26. **UTC 20 Failed**

```
@Test
public void testRobotMoveWestBeyondRoomBounds() {
    room = new Room( size: 5);
    robot.setXY( x: 0, y: 2); // Robot's initial position: (0, 2), left boundary
    robot.setPenDown();
    robot.direction = Robot.Direction.West;

    // Call the move method after initializing the robot object
    App.robot = robot; // Set the robot object in the App class
    App.room = room;
    App.move( s: "1");
    System.out.println(robot.x);
    System.out.println(robot.y);
    Assertions.assertEquals( expected: 0, robot.x); // Expected X position: 0 (no change)
    Assertions.assertEquals( expected: 2, robot.y); // Expected Y position: 2 (no change)
}
```

Fig. 27. **UTC 21 testRobotMoveWestBeyondRoomBounds**

```
@Test
public void testRobotMoveWestWithPenUp() {
    room = new Room( size: 5);
    robot.setXY( x: 2, y: 2); // Robot's initial position: (2, 2)
    robot.setPenUp();
    robot.direction = Robot.Direction.West;

    // Call the move method after initializing the robot object
    App.robot = robot; // Set the robot object in the App class
    App.room = room;
    App.move( s: "1"); // Move one step west

    // Verify the room remains unchanged (all cells should be 0)
    String expectedOutput =
            "0 0 0 0 0 \n" +
                    "0 0 0 0 0 \n" +
                    "0 0 0 0 0 \n" +
                    "0 0 0 0 0 \n" +
                    "0 0 0 0 0 \n";
    Assertions.assertEquals(expectedOutput, getRoomPrintOutput(room));
}
```

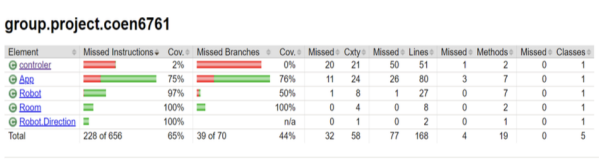
Fig. 28. **UTC 22 testRobotMoveWestWithPenUp**

V. UNIT TEST CASE TABLE

The unit test case results for the black box testing for the Development Team

R	Unit Test Cases (UTC)	Results
1	UTC1	Pass
2	UTC2	Pass
3	UTC3	Pass
4	UTC4	Pass
5	UTC5	Pass
6	UTC6	Pass
7	UTC7	Pass
8	UTC8	Pass
9	UTC9	Pass
10	UTC10	Pass
11	UTC11	Pass
12	UTC12	Pass
13	UTC13	Pass
14	UTC14	Pass
15	UTC15	Pass
16	UTC16	Pass
17	UTC17	Pass
18	UTC18	Fail
19	UTC19	Fail
20	UTC20	Fail
21	UTC21	Pass
22	UTC22	Pass

VI. WHITE BOX TESTING



group.project.coen6761

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
controler		2%		0%	20	21	50	51	1	2	0	1
App		75%		76%	11	24	26	80	3	7	0	1
Robot		97%		50%	1	8	1	27	0	7	0	1
Room		100%		100%	0	4	0	8	0	2	0	1
Robot.Direction		100%		n/a	0	1	0	2	0	1	0	1
Total	228 of 656	65%	39 of 70	44%	32	58	77	168	4	19	0	5

Fig. 29. **Code Coverage results of the Development Team**

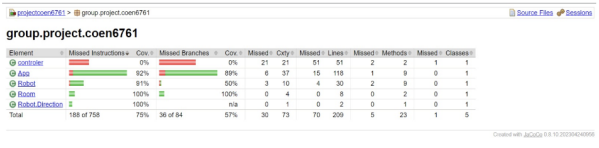


Fig. 30. **Code Coverage results produced by our Team**

The results prove that coverage produced by our team is 10% higher than the coverage produced by the development team.

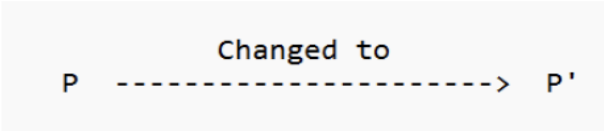
Fig. 31. **Code coverage for the move( ) function from Jacoco**

The below mentioned types of code coverage are calculated for the above-mentioned function.

1) **Statement Coverage**: This coverage makes sure that each statement is executed simultaneously. All of the lines, paths, and statements observed in the code will be included in this.
   Statement Coverage (%) = (Number of Executed Statements / Total Number of Statements) * 100 = (34/40)*100 = 85%
2) **Decision Coverage**: Testing for the Boolean values will be covered by decision coverage. All of the control statements will be covered by this coverage.
   Decision Coverage (%) = (Number of Executed Branches / Total Number of Branches) * 100 = 7/9 = 77.7%
3) **Condition Coverage** In this coverage all the conditions outcome are tested.
   In this coverage all the conditions outcome are tested. Condition Coverage (%) = (Number of Executed Condition Combinations / Total Number of Condition Combinations) * 100 = (2/4)* 100 = 50%
4) **Multiple Condition Coverage**: Multiple Condition Coverage is similar to condition coverage, but multiple functions are tested in this coverage.
   There are no multiple conditions in the source code of the development team and hence the multiple condition coverage cannot be calculated for their case

VII. MUTATION TESTING

We can use mutation testing method to evaluate the adequacy of a test set utilizing mutant programs. Assume that program P has passed all test cases in test set T after being put through its paces. Let's say we carry out the following: In this testing, if a certain part in the main code is changed



and tested whether test cases can find the errors in the source code. Mutation testing is done, and it has passed.

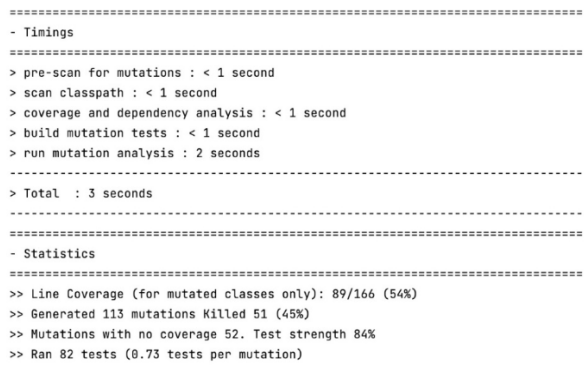


Fig. 32. **Statistics of Mutation Testing**

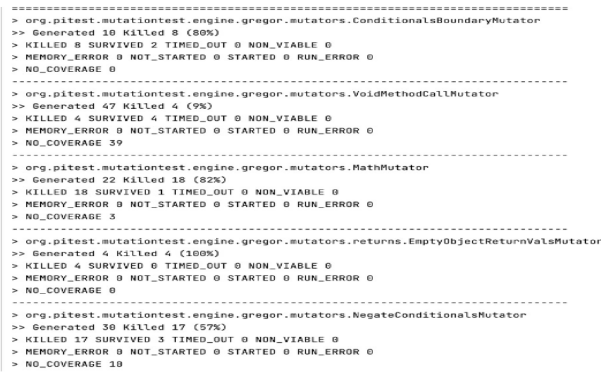


Fig. 33. **Mutation testing results for various mutators using PIT TEST plugin**

Package Summary

group.project.coen6761

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
4	62% 129/208	53% 71/134	89% 71/80

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
App.java	80% 95/119	68% 58/85	87% 58/67
Robot.java	87% 26/30	88% 7/8	100% 7/7
Room.java	100% 8/8	100% 6/6	100% 6/6
controler.java	0% 0/51	0% 0/35	0% 0/0

Fig. 34. **Pit Test Coverage Report**

VIII. DATA FLOW TESTING

Data variables and their values are the primary focus of the software testing technique known as data flow testing. The control flow graph is employed. In terms of classification, White box testing and structural types of testing can both be applied to data flow testing. It maintains an eye on the variables and its usage points at the data receiving points. The procedure is used to find faults that result from the inaccurate application of data variables or data values.

A. TEST CASES DEVELOPED FOR DATA FLOW TESTING

```
@Test
public void testInitializationOfRoom_2() {
    String choice = "i 5";
    app.runCommand(choice);
    // Assertions and other checks
}
```

Fig. 35. **UTC 1 testInitializationOfRoom 2 Data Flow**

```
@Test
public void testPrintCurrentPositionOfRobot() {
    String choice = "c";

    app.robot = new Robot();
    App.robot.setXY( x: 2,  y: 3);
    App.robot.setPenDown();
    App.robot.direction = Robot.Direction.North;

    String expectedOutput = "Position:2,3 - Pen:Down - Facing:North";
    Assertions.assertEquals(expectedOutput, app.runCommandAndGetOutput(choice));
}
```

Fig. 36. **UTC 2 testPrintCurrentPositionOfRobot – Data Flow**

```
@Test
public void testPenUp() {
    String choice = "u";

    app.robot = new Robot();
    App.robot.setPenDown();
    app.runCommand(choice);
    Assertions.assertFalse(App.robot.pen);
}
```

Fig. 37. **UTC 3 testPenUp Data Flow**

```
@Test
public void testPenDown() {
    String choice = "d";

    app.robot = new Robot();
    App.robot.setPenUp();
    app.runCommand(choice);
    Assertions.assertTrue(App.robot.pen);
}
```

Fig. 38. **UTC 4 testPenDown Data Flow**

```
@Test
public void testTurnRight() {
    String choice = "r";

    app.robot = new Robot();
    App.robot.direction = Robot.Direction.North;
    app.runCommand(choice);
    Assertions.assertEquals(Robot.Direction.East, App.robot.direction);
}
```

Fig. 39. **UTC 5 testTurnRight Data Flow**

```
@Test
public void testTurnLeft() {
    String choice = "l";

    app.robot = new Robot();
    App.robot.direction = Robot.Direction.East;
    app.runCommand(choice);
    Assertions.assertEquals(Robot.Direction.North, App.robot.direction);
}
```

Fig. 40. **UTC 6 testTurnLeft Data Flow**

```
@Test
public void testMoveEast() {

    String choice = "i 6";

    app.runCommand(choice);

    choice = "m 3";
    App.robot = new Robot();
    App.robot.setXY( x: 2,  y: 2);
    App.robot.direction = Robot.Direction.East;

    // Initialize the room with a size of 6 (or any desired size)


    app.runCommand(choice);

    Assertions.assertEquals( expected: 5, App.robot.x);
    Assertions.assertEquals( expected: 2, App.robot.y);
}
```

Fig. 41. **UTC 7 testMoveEast Data Flow**

```
@Test
public void testMoveWest() {
    String choice = "m 2";

    app.robot = new Robot();
    App.robot.setXY( x: 3,  y: 3);
    App.robot.direction = Robot.Direction.West;

    app.runCommand(choice);

    Assertions.assertEquals( expected: 1, App.robot.x);
    Assertions.assertEquals( expected: 3, App.robot.y);
}
```

Fig. 42. **UTC 8 testMoveWest Data Flow**

```
@Test
public void testMoveNorth() {

    String choice = "i 6";

    app.runCommand(choice);
    choice = "m 4";

    app.robot = new Robot();
    App.robot.setXY( x: 1,  y: 2);
    App.robot.direction = Robot.Direction.North;

    app.runCommand(choice);

    Assertions.assertEquals( expected: 1, App.robot.x);
    Assertions.assertEquals( expected: 6, App.robot.y);
}
```

Fig. 43. **UTC 9 testMoveNorth Data Flow**

```
@Test
public void testMoveSouth() {
    String choice = "m 1";

    app.robot = new Robot();
    App.robot.setXY( x: 4,  y: 4);
    App.robot.direction = Robot.Direction.South;

    app.runCommand(choice);

    Assertions.assertEquals( expected: 4, App.robot.x);
    Assertions.assertEquals( expected: 3, App.robot.y);
}
```

Fig. 44. **UTC 10 testMoveSouth Data Flow**

```
@Test
public void testPrintRoom() {
    String choice = "p";
    App.room = new Room( size: 4);
    App.room.room[2][2] = 1;
    App.room.room[3][1] = 1;

    String expectedOutput =
            "0 0 0 0 " + LINE_SEPARATOR +
                    "0 0 0 0 " + LINE_SEPARATOR +
                    "0 0 1 0 " + LINE_SEPARATOR +
                    "0 1 0 0";

    Assertions.assertEquals(expectedOutput, app.runCommandAndGetOutput(choice));
}
```

Fig. 45. **UUTC 11 testPrintRoom Data Flow**

```
@Test
public void testInvalidChoice() {
    String choice = "x";
    String expectedOutput = "Invalid choice. Please try again.";
    Assertions.assertEquals(expectedOutput, app.runCommandAndGetOutput(choice));
}
```

Fig. 46. **UTC 12 testInvalidChoice Data Flow**

B. DATA FLOW TESTING RESULTS

The results of Data Flow testing are tabulated below for reference.

R	Unit Test Cases (UTC)	Results
1	UTC1	Pass
2	UTC2	Pass
3	UTC3	Pass
4	UTC4	Pass
5	UTC5	Pass
6	UTC6	Pass
7	UTC7	Pass
8	UTC8	Pass
9	UTC9	Pass
10	UTC10	Pass
11	UTC11	Pass
12	UTC12	Pass

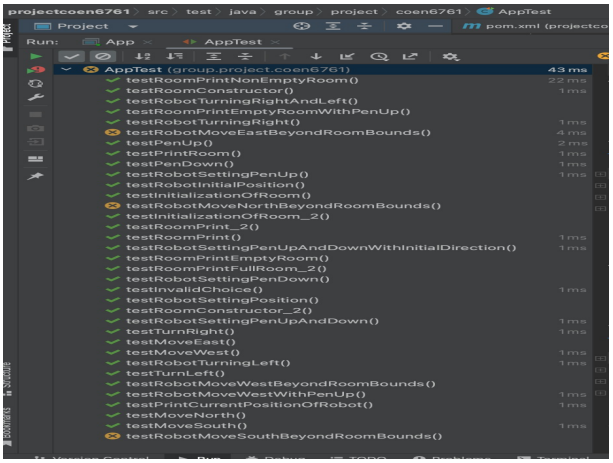


Fig. 47. **Results of all the test cases including data flow testing test cases**

IX. CONCLUSION

We have successfully produced 22 unit test cases and completed Black-box testing. We have performed code coverage using the JaCoCo testing tool and our coverage result turns out to be 10% higher than the actual coverage results of the development team. We have completed mutation testing and we have used PIT test plugin. Furthermore, we have produced 12 unit test cases to check data flow testing and all the test cases have passed successfully. Although the development team had some bugs in their source code and our team has found them, the development team has done a good job in producing their work.