COEN 6761 - Software Testing and Validation

Task 2 Report: Robot Motion Project

Submitted By

Name	Student ID
Samantha Menezes	
Muhammad Nofil	

Contents

1. TASK 1 – DEVELOPMENT AND VERIFICATION

1.1 PROJECT OVERVIEW

The robot motion project simulates a robot moving on an N x N floor while responding to user commands. The robot can move in four directions, hold a pen in up/down positions, and track its movements. The software is developed using Java and follows Agile Scrum methodology.

1.2 GITHUB REPOSITORY

Full source code and test cases are available in the following GitHub Repository:
GitHub-Repo

1.3 USER STORIES AND REQUIREMENTS

The following requirements define the behavior of the robot simulation program:

Req. ID	Title	Description
R1	System Initialization	The system sets the robot at position (0, 0) with the pen up and facing north when the robot is created.
R2	Move Functionality	The robot shall be able to move forward by a specified number of spaces in its current facing direction, regardless of whether the pen is up or down.
R3	Pen Control	The system shall allow the user to toggle the pen between up and down states, which controls whether the robot marks the floor while moving.
R4	Turn Functionality	The robot shall be able to turn 90 degrees to the right or to the left, changing its facing direction.
R5	Marking the floor	When the robot moves with the pen down, the system shall mark each position it passes, indicating the path taken by the robot.
R6	Replay History	The system shall record all commands (e.g., movement, pen state, turns) in a history list and

		provide a method to replay the commands in sequence, outputting each action taken by the robot.
R7	Quit Simulation	The system shall allow the user to quit the simulation at any time by inputting the "Q" command, which will stop further actions and exit the program.
R8	Display current state	The system shall provide a command ("C") to display the current position of the robot, its pen state (up/down), and its facing direction.
R9	Print the floor	The system shall allow the user to print the current floor grid, displaying any marks made by the robot during its movement.

1.4 IMPLEMENTATION

The system processes the following commands:

- I n| i n – Initialize floor size to n x n.
- U | u– Pen up.
- D | d – Pen down.
- R | r – Turn right.
- L | l – Turn left.
- M s | m s– Move forward s spaces.
- P | p – Print the floor.
- C | c – Display the robot's position and pen status.
- Q | q – Quit program.
- H | h – Replay all executed commands.

The program uses a 2D integer array (floor[][]) to track movement and pen markings. Movement is restricted within bounds.

1.4.1 SYSTEM INITIALIZATION

The RobotSimulation is a constructor of class RobotSimulation. It initializes the robot on a floor of size n x n. The robot's position is set to [0, 0], its pen is up, and it faces north.

```
public RobotSimulation(int n) {
    floor = new int[n][n];
    x = 0;
    y = 0;
    penDown = false;
    facing = Direction.NORTH;
    history = new ArrayList<>();
}
```

1.4.2 MOVE FUNCTIONALITY

Moves the robot a specified number of steps (steps) in the direction it is currently facing. The method is designed to move the robot step by step.The loop runs for the number of times specified by the steps argument, allowing the robot to move multiple steps. For example: If the step count is 3, the loop runs 3 times (i.e., the robot moves 3 times).

For every iteration, we create local variables newX and newY to store the new position of the robot before updating its position. The variables x and y represent the robot's current position on the floor grid. These values are copied to newX and newY to compute the new position based on the direction the robot is facing.

The new position is calculated based on the robot's current direction:

- NORTH: If the robot is facing north, it moves up the grid, so newY is incremented (newY++).
- SOUTH: If the robot is facing south, it moves down the grid, so newY is decremented (newY--).
- EAST: If the robot is facing east, it moves to the right of the grid, so newX is incremented (newX++).
- WEST: If the robot is facing west, it moves to the left of the grid, so newX is decremented (newX--).

The conditions newX >= 0 && newX < floor.length ensure that the new x-coordinate remains within the valid grid range.Similarly, newY >= 0 && newY < floor.length ensures that the new y-coordinate is within bounds.

This check prevents the robot from attempting to move outside the floor grid. If the new position is out of bounds, the robot does not move, and its position remains unchanged.

Once the new position is validated, the robot’s actual position (x, y) is updated to the newly calculated position (newX, newY).

If the pen is down, the robot marks the floor at its new position.

After the robot completes its movement, the action is recorded in the history list. The string "M <steps>" is added to the history list, where <steps> represents the number of steps the robot moved. This allows the robot's actions to be replayed later or tracked.

```
public void move(int steps) {
    for (int i = 0; i < steps; i++) {
        int newX = x, newY = y;
        switch (facing) {
            case NORTH: newY++; break;
            case SOUTH: newY--; break;
            case EAST: newX++; break;
            case WEST: newX--; break;
        }
        if (newX >= 0 && newX < floor.length && newY >= 0 && newY < floor.length) {
            x = newX;
            y = newY;
            if (penDown) {
                floor[y][x] = 1;
            }
        }
    }
    history.add("M " + steps);
}
```

1.4.3 PEN CONTROL

The penUp method sets penDown to false, indicating that the pen is up. It also records this action in the history list by adding "U".
The penDown method sets penDown to true, indicating that the pen is down. It also records this action in the history list by adding "D".

```
public void penUp() {
    penDown = false;
    history.add("U");
}

public void penDown() {
    penDown = true;
    history.add("D");
}
```

1.4.4 TURN FUNCTIONALITY

The turnRight method changes the robot's facing direction by turning it right (clockwise). It rotates through the Direction enum in the following order:
NORTH → EAST → SOUTH → WEST → NORTH.
The method also records the action in the history list with "R".

```
public void turnRight() {
    switch (facing) {
        case NORTH: facing = Direction.EAST; break;
        case EAST: facing = Direction.SOUTH; break;
        case SOUTH: facing = Direction.WEST; break;
        case WEST: facing = Direction.NORTH; break;
    }
    history.add("R");
}
```

The turnLeft method changes the robot's facing direction by turning it left (counterclockwise). It rotates through the Direction enum in the following order: NORTH → WEST → SOUTH → EAST → NORTH.
The method records the action in the history list with "L".

```
public void turnLeft() {
    switch (facing) {
        case NORTH: facing = Direction.WEST; break;
        case WEST: facing = Direction.SOUTH; break;
        case SOUTH: facing = Direction.EAST; break;
        case EAST: facing = Direction.NORTH; break;
    }
    history.add("L");
}
```

1.4.5 MARKING THE FLOOR

If the pen is down (penDown == true), the robot will mark the floor as it moves.

```
if (newX >= 0 && newX < floor.length && newY >= 0 && newY < floor.length) {
    x = newX;
    y = newY;
    if (penDown) {
        floor[y][x] = 1;
    }
}
```

1.4.6 REPLAY HISTORY

The history list records every action taken by the robot. Each command is stored as a string.
Whenever a method like move(), turnRight(), or penDown() is called, the corresponding command is added to the history.

Command	Stored in History as:
Pen Up	“U”
Pen Down	“D”
Turn Right	“R”
Turn Left	“L”
Move 3 steps	“M 3”

```
public List<String> getAndReplayHistory(boolean print) {
    if (print) {
        for (String command : history) {
            System.out.println("Executing: " + command);
        }
    }
    return new ArrayList<>(history); // Return a copy to maintain encapsulation
}
```

1.4.7 QUIT SIMULATION

The Q command exits the program immediately. The System.exit(0); method is used to terminate the program.

```
case "Q":
    System.out.println("Exiting...");
    System.exit(0);
```

1.4.8 DISPLAY CURRENT STATE

The C command prints the robot’s current position, pen status, and facing direction.

```
public String getCurrentState() {
    return "Position: [" + x + ", " + y + "]\n" +
           "Pen: " + (penDown ? "Down" : "Up") + "\n" +
           "Facing: " + facing;
}
```

1.4.9 PRINT THE FLOOR

The P command prints the entire grid where the robot has moved. Marked positions (1s) are displayed as *, while unmarked positions remain blank.

```java
public void printFloor() {
    // Print grid with row numbers
    for (int i = floor.length - 1; i >= 0; i--) {
        System.out.printf("%2d ", i); // Row numbers
        for (int j = 0; j < floor.length; j++) {
            if (floor[i][j] == 1) {
                System.out.print("* "); // Print '*' for 1
            } else {
                System.out.print("  "); // Print nothing for 0
            }
        }
        System.out.println(); // Move to the next line
    }
    // Print top row numbers
    System.out.print("   "); // Offset for row numbers
    for (int j = 0; j < floor.length; j++) {
        System.out.print(j + " "); // Print each column number
    }
    System.out.println();
}
```

Functionality Breakdown

Iterates over the grid in reverse order so the top row is printed first.

Displays:

- "*" for marked positions.
- Empty space for unmarked positions.
- Row and column indices for reference.

1.5 MAPPING REQUIREMENTS AND UNIT TEST CASES

Requirement	Unit Test Case	Description
Robot is initialized with a given floor size	testInitializeSystem	Verifies that the robot is initialized at the correct position [0, 0], with the pen up, and facing north when a RobotSimulation object is created with a given size.
Pen can be lifted up and down	testPenUpAndDown	Tests the ability to lift the pen (penUp) and ensure the pen

		status is correctly reflected. Also tests the opposite action (penDown).
Robot can move forward without marking the floor	testMoveWithoutPen	Verifies that the robot moves in the correct direction but does not mark the floor when the pen is up (i.e., the position changes but the floor remains unmarked).
Robot can move forward and mark the floor	testMoveWithPen	Tests that the robot moves in the correct direction, marks the floor when the pen is down, and that the cells along the path are marked with 1.
Robot can turn right	testTurnRight	Verifies that the robot turns right correctly, changing its facing direction in the expected order (NORTH → EAST → SOUTH → WEST → NORTH).
Robot can turn left	testTurnLeft	Verifies that the robot turns left correctly, changing its facing direction in the expected order (NORTH → WEST → SOUTH → EAST → NORTH).
Robot can turn right and move	testTurnRightAndMove	Verifies that after turning right, the robot moves in the new direction, maintaining the correct position updates.
Robot's floor can be printed	testPrintFloor	Verifies that the printFloor() method works as expected without throwing any exceptions, ensuring the grid is displayed correctly with marked positions.
Current state is printed with pen up	testPrintCurrentStatePenUp	Verifies that when the pen is up, the current state is correctly printed with the robot’s position, pen status, and facing direction.

Current state is printed with pen down	testPrintCurrentStatePenDown	Verifies that when the pen is down, the current state is correctly printed with the robot's position, pen status, and facing direction.
History of commands can be replayed	testHistoryReplay	Verifies that the history of commands (e.g., pen up/down, move, turn) is stored correctly and can be replayed accurately, showing all executed commands.

1.6 TEST CASES EXECUTION RESULTS

The table below presents the execution results of the test cases designed to verify the core functionalities of the Robot Motion Project. Each test case corresponds to a specific system requirement and ensures that the robot behaves as expected. All test cases were executed successfully, confirming correct initialization, movement, pen control, direction changes, floor marking, history replay, and simulation termination. The results indicate that the system meets all functional requirements, with no failures observed during testing.

Requirement ID	Title	Description	Expected result	Pass/Fail
R1	System Initialization	The system sets the robot at position (0,0) with the pen up and facing north when created.	Robot starts at [0,0], pen is up, facing NORTH.	Pass
R2	Move Functionality	The robot moves forward a specified number of spaces in its current facing direction.	Position updates correctly based on facing direction.	Pass
R3	Pen Control	The system allows toggling the pen between up/down states to control floor marking.	penUp() sets pen state to UP, penDown() sets it to DOWN.	Pass
R4	Turn Functionality	The robot can turn 90 degrees left or right, changing its facing direction.	turnRight(): NORTH → EAST → SOUTH → WEST → NORTH. turnLeft(): NORTH → WEST → SOUTH → EAST → NORTH.	Pass

R5	Marking the Floor	The robot marks each position it moves to if the pen is down.	Floor is marked (1 or *) at all positions traveled when pen is down.	Pass
R6	Replay History	The system records all commands (move, turn, pen state) and replays them in sequence.	replayHistory() prints previously executed commands in order.	Pass
R7	Quit Simulation	Allows the user to quit the simulation by entering ‘Q’	"Exiting..." message is printed, and program terminates.	Pass
R8	Display Current State	The ‘C’ command shows the robot's position, pen state, and facing direction.	Correct position, pen state, and facing direction displayed.	Pass
R9	Print the Floor	The ‘P’ command displays the current floor grid with marks.	printFloor() outputs a grid with marked positions (*).	Pass

1.7 SCREEN CAPTURES

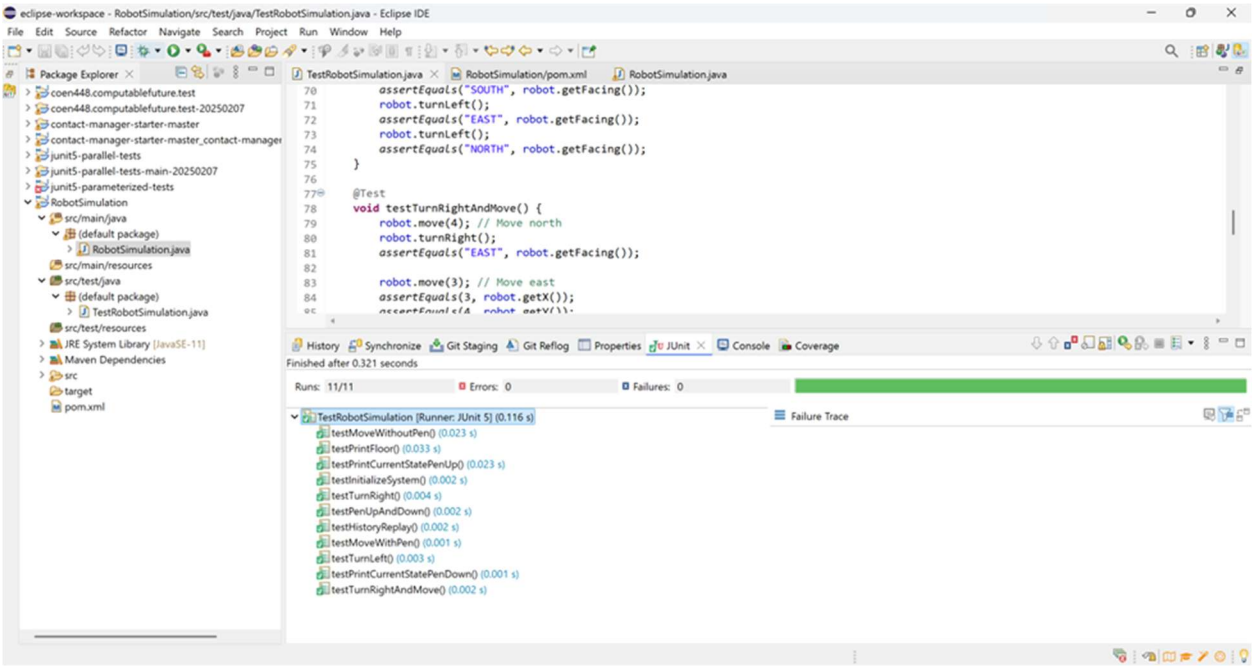


Fig 1. Junit test results

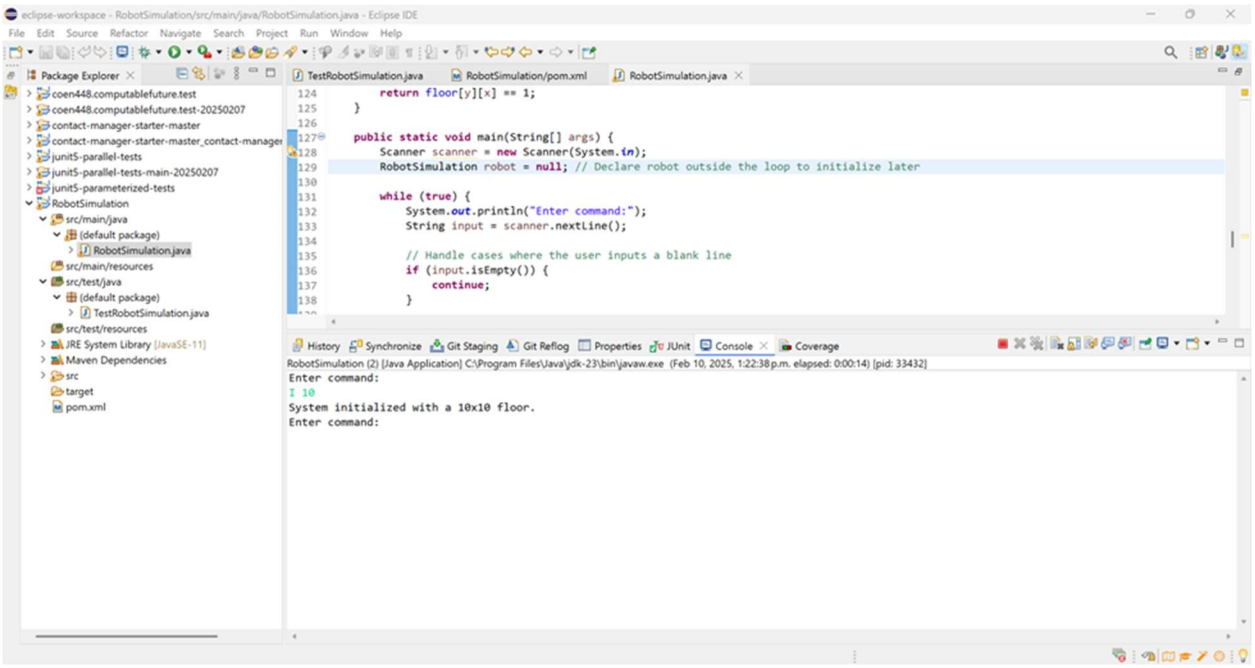


Fig 2. Initialization functionality

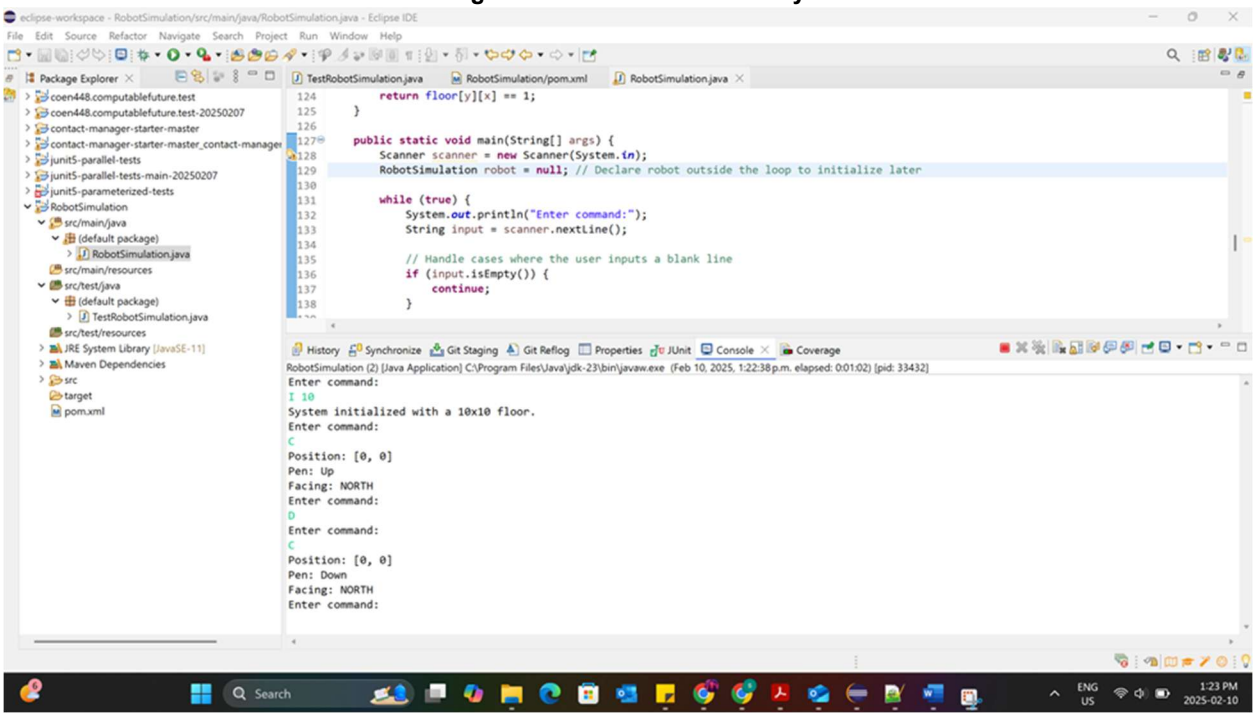


Fig 3. Pen up and pen down functionality

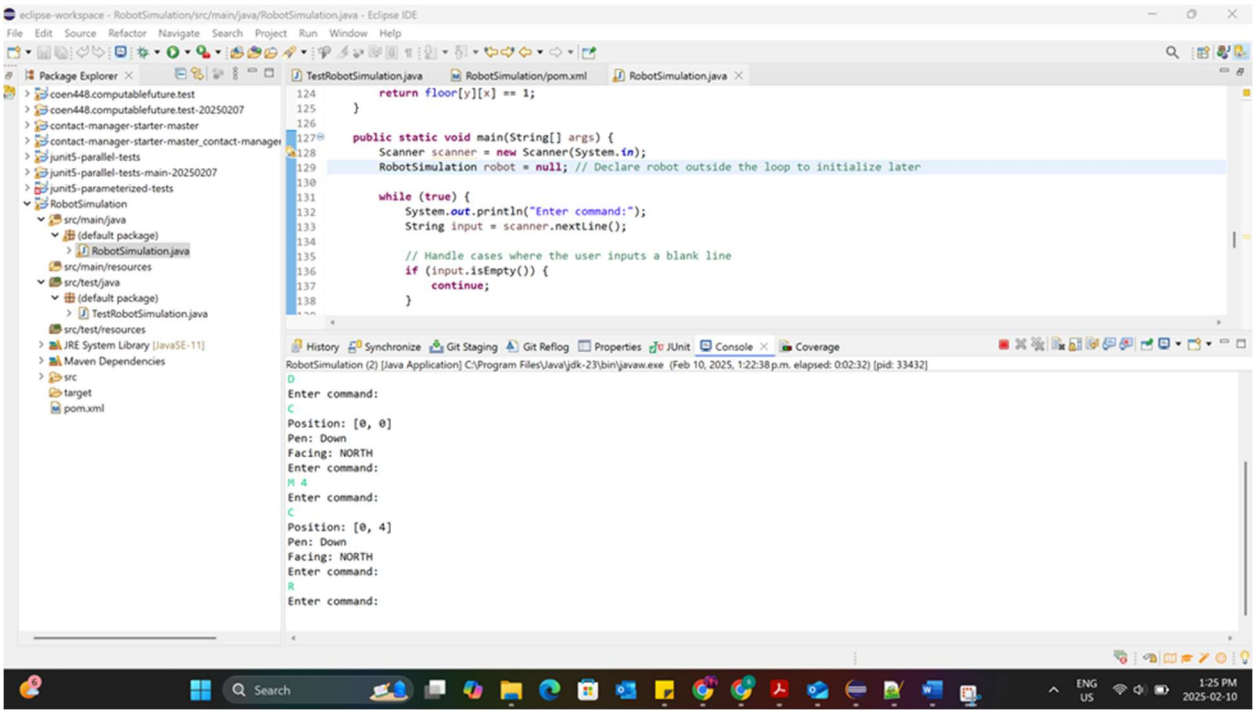


Fig 4. Move functionality

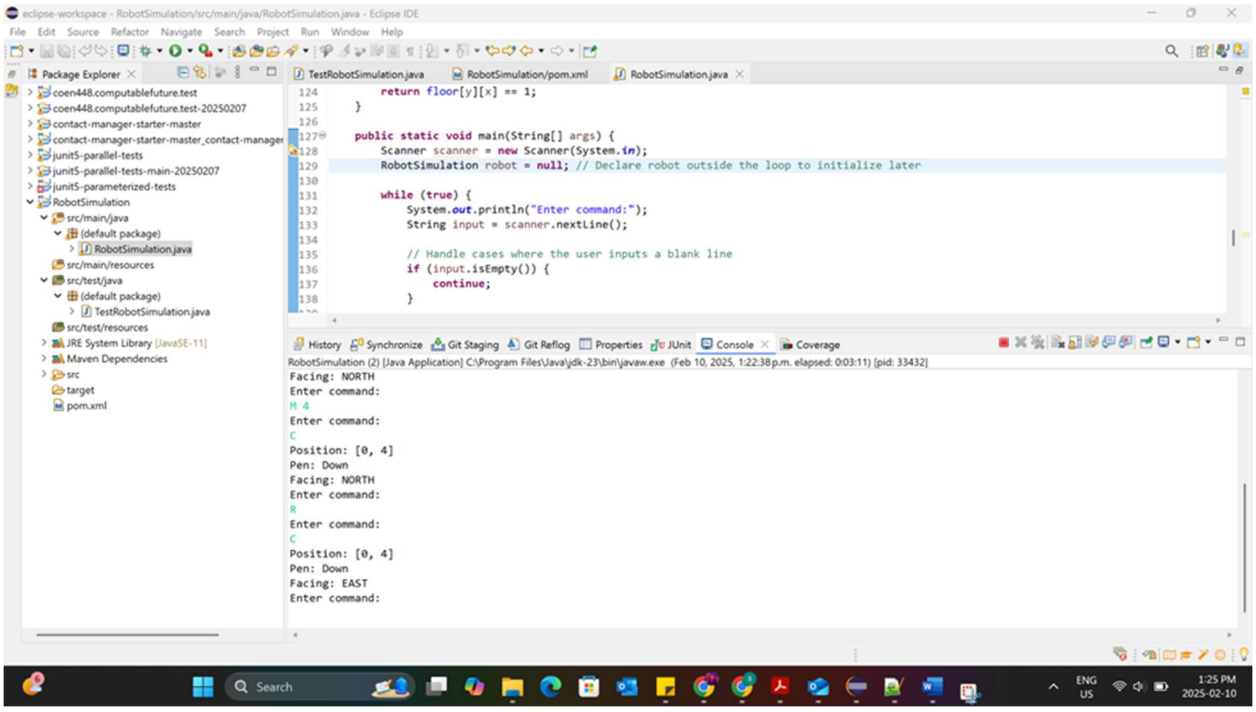


Fig 5. Turn functionality

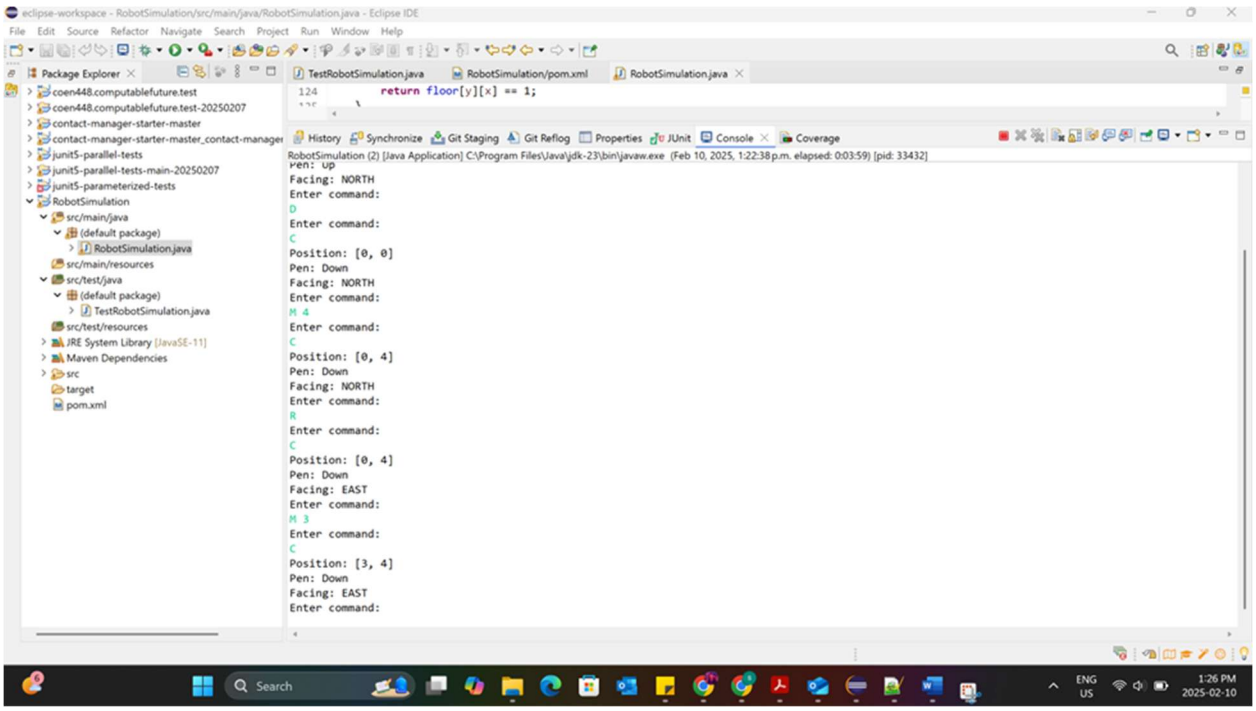


Fig 6. Turn and move functionality

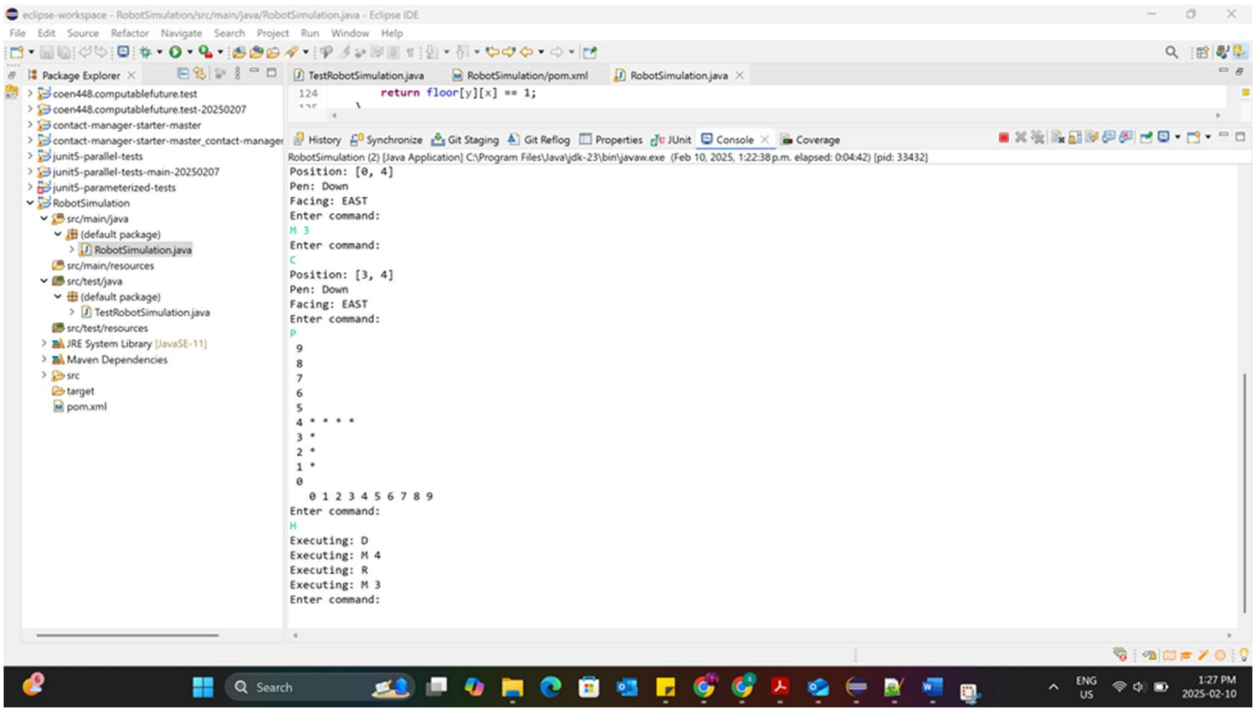


Fig 7. Floor printing functionality

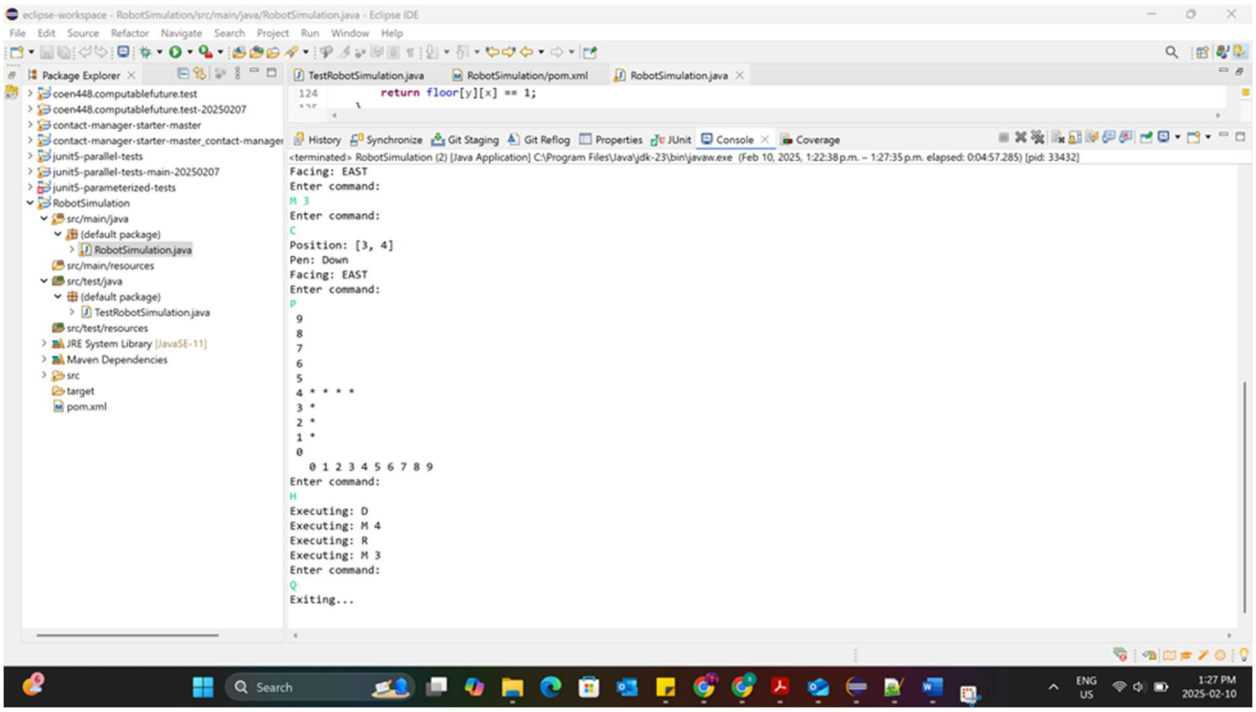


Fig 8. History of commands

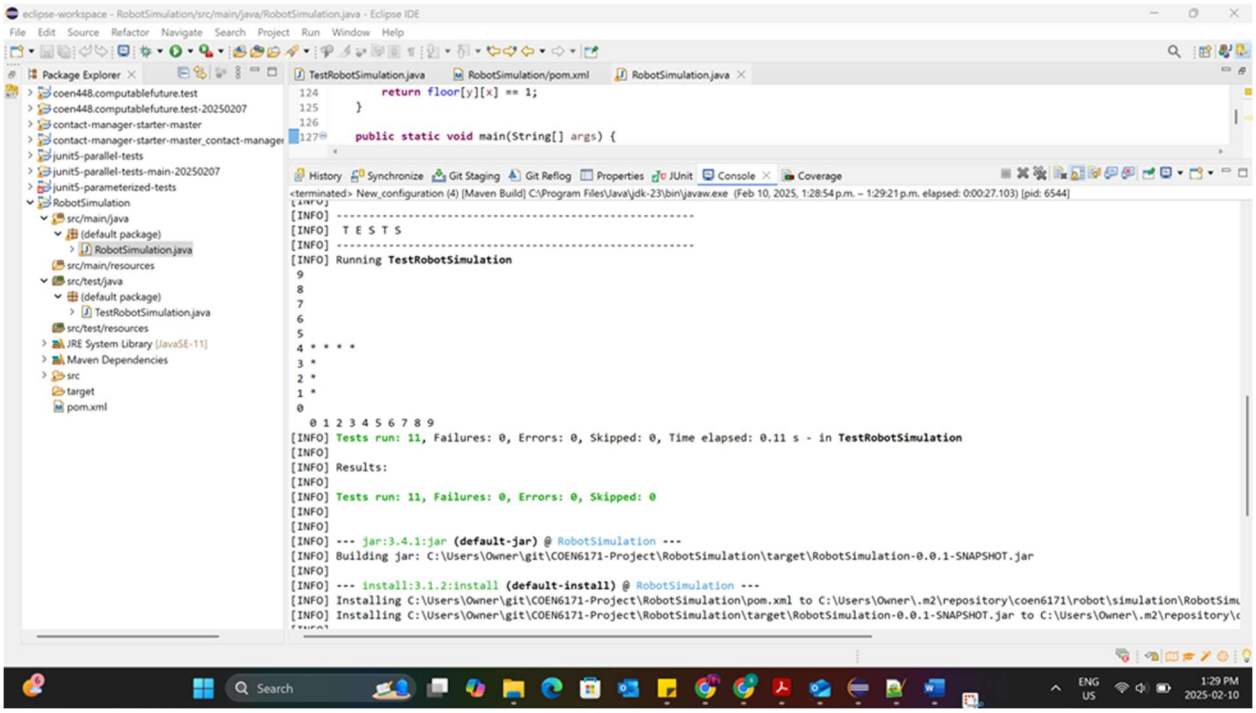


Fig 9. Unit test results and logs

1.8 DEVELOPMENT TIME TRACKING

A time-tracking tool (WakaTime) was used to compare development and testing time:

Function	Development Time (mins)	Testing Time (mins)	Total Time (mins)	% of Total
RobotSimulation(int n)	10	6	16	8.8%
penUp()	4	5	9	4.9%
penDown()	4	5	9	4.9%
turnRight()	6	7	13	7.1%
turnLeft()	6	7	13	7.1%
move(int steps)	20	24	44	24.2%
printFloor()	12	10	22	12.1%
getCurrentState()	6	9	15	8.2%
replayHistory()	10	12	22	12.1%
Subtotal (Function-Specific Time)	**88 mins**	**94 mins**	**182 mins**	**100%**

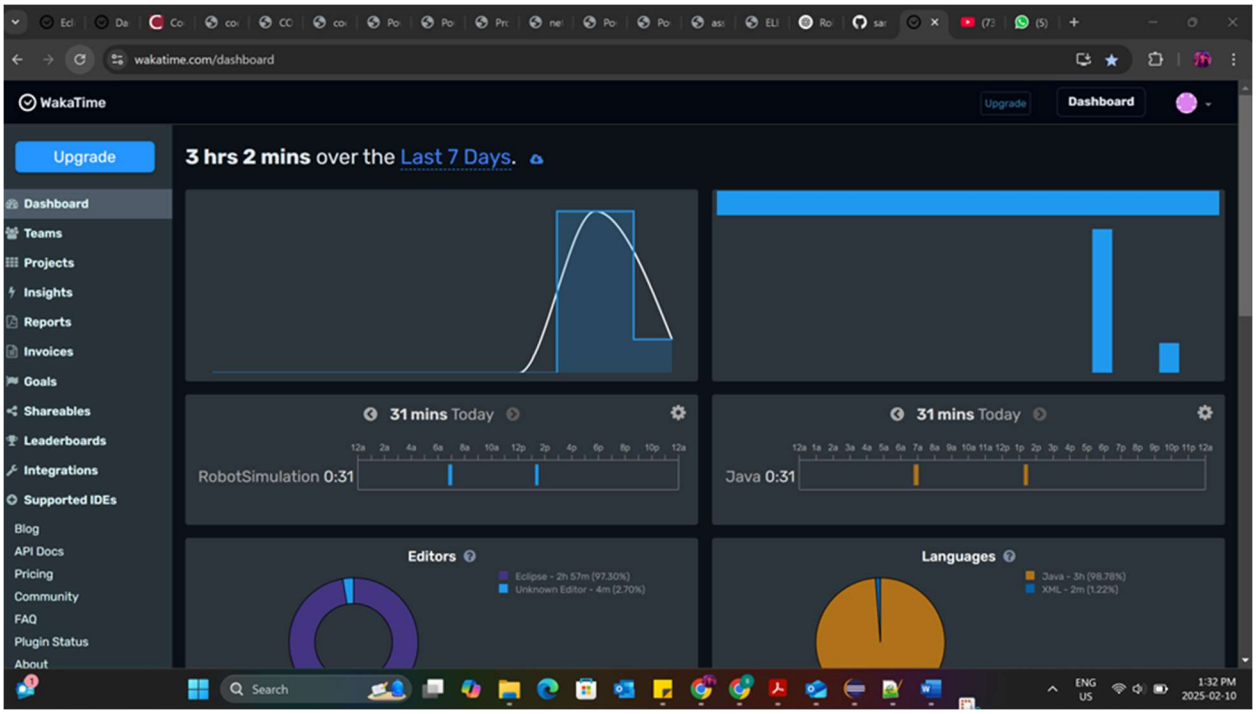


Fig 10. Time tracking through Wakatime

While the core development and testing of functions took most of the time (~182 mins), additional efforts in code flow, debugging, and refining input handling added another 30 minutes. These non-function-specific tasks are crucial in ensuring overall stability and correctness, as they help eliminate unexpected errors in execution flow before finalizing the project.

Activity	Time Spent (mins)	% of Total
Debugging issues during testing	12	6.6%
Refining switch-case logic in main function	10	5.5%
Ensuring correct input handling & flow testing	8	4.4%
Total Additional Time	30 mins	16.5%

2. TASK 2 - COVERAGE REPORT

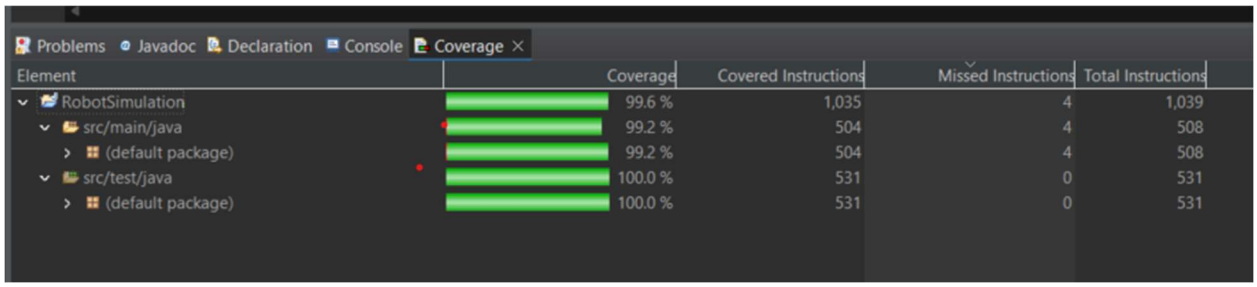
2.1 FUNCTIONAL COVERAGE

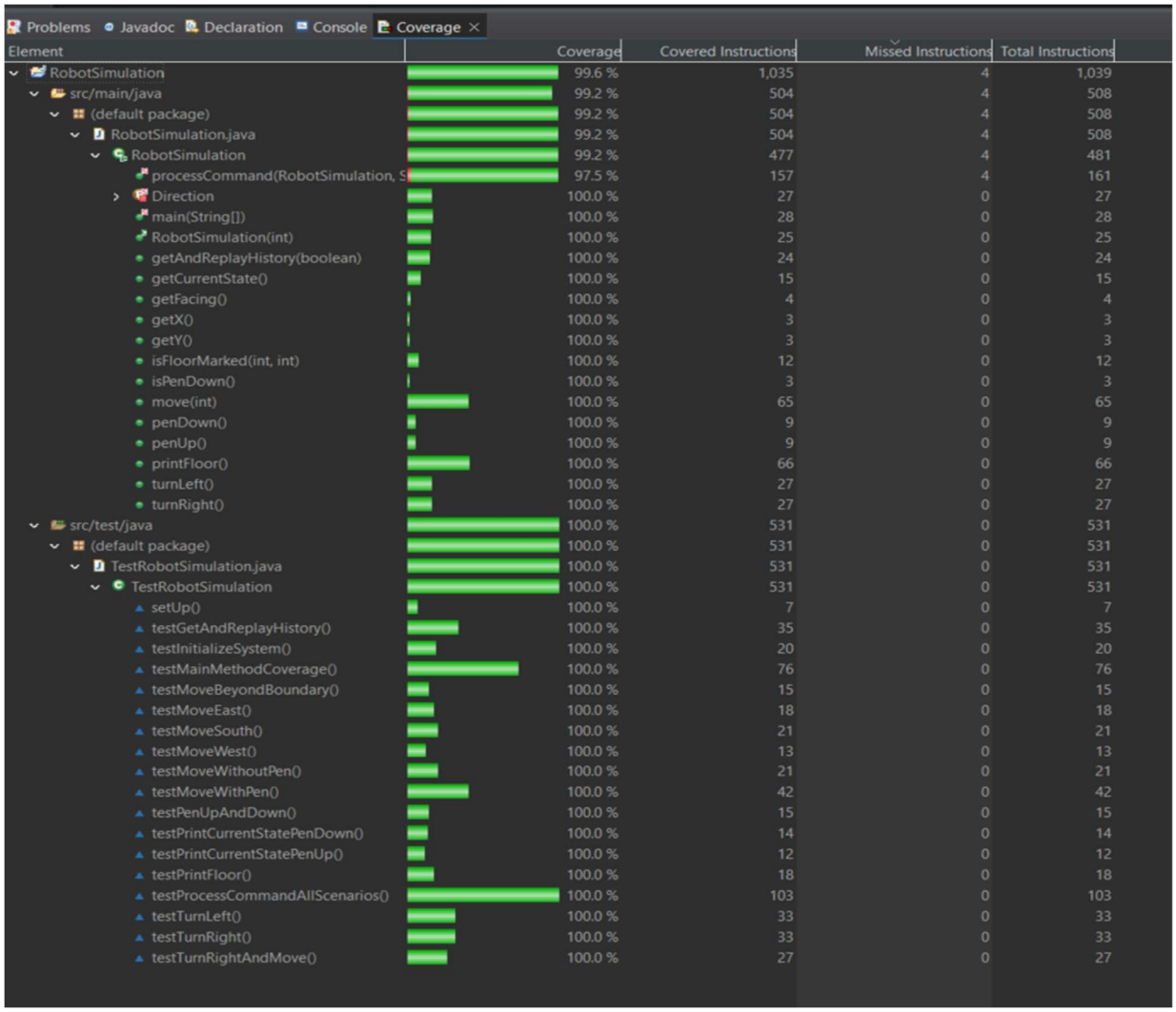
METHOD	DESCRIPTION	COVERAGE	RELEVANT TEST
RobotSimulation(int n) [Constructor]	Initializes a n x n floor grid. The robot starts at (0, 0), facing NORTH, with the pen up. A history list stores all executed commands.	100%	testInitializeSystem()
int getX()	Returns the current x-coordinate of the robot.	100%	Covered in multiple movement-related tests, e.g., testMoveWithPen()
int getY()	Returns the current y-coordinate of the robot.	100%	Covered in multiple movement related tests
void penUp()	Sets penDown = false, preventing the robot from marking the floor. Adds "U" to the history.	100%	testPenUpAndDown()
void penDown()	Sets penDown = true, allowing the robot to mark the floor. Adds "D" to the history.	100%	testPenUpAndDown()
boolean isPenDown()	Returns true if the pen is down, false otherwise.	100%	testPenUpAndDown()
String getFacing()	Returns the current direction the robot is facing.	100%	testTurnRight() testTurnLeft()
void turnRight()	Rotates the robot 90° clockwise. (NORTH → EAST	100%	testTurnRight()

	→ SOUTH → WEST → NORTH) Adds "R" to the history.		
void turnLeft()	Rotates the robot 90° counterclockwise. (NORTH → WEST → SOUTH → EAST → NORTH) Adds "L" to the history.	100%	testTurnLeft()
void move(int steps)	Moves the robot forward in the current direction, marking the floor if penDown = true. If movement exceeds grid boundaries, it stops at the edge. Adds "M <step_count>" to the history.	100%	testMoveWithoutPen() testMoveWithPen() testMoveBeyondBoundary()
void printFloor()	Prints the grid where * represents marked cells and empty spaces represent unmarked cells.	100%	testPrintFloor()
String getCurrentState()	Returns the robot's position, pen state, and facing direction in a formatted string.	100%	testPrintCurrentStatePenUp() testPrintCurrentStatePenDown()
List<String> getAndReplayHistory(boolean print)	Returns a copy of the history list. If print = true, prints the commands.	100%	testGetAndReplayHistory()

2.2 STATEMENT COVERAGE

Statements Covered	Statements Missed	Total Statements
504	4	508

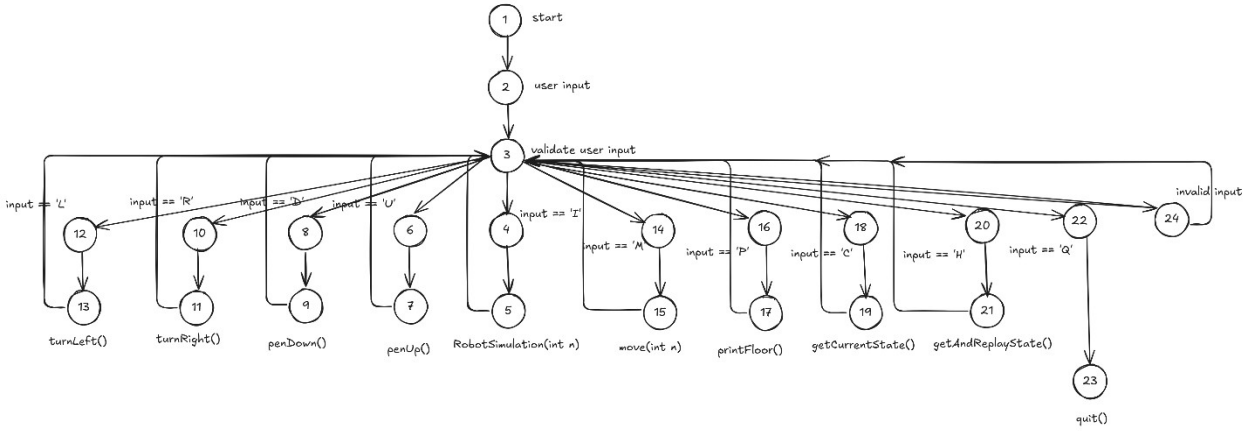




The high coverage shows that the test suite effectively verifies most execution paths. The 4 missed statements suggest some untested edge cases, likely in exception handling or specific conditions. Adding targeted test cases can help achieve 100% coverage.

2.3 PATH COVERAGE

Cyclomatic complexity **M = E – N + 2P**, where
E = number of edges in control flow graph
N = number of nodes in control flow graph
P = number of connected components



E = 34
N = 24
P = 1
For the above graphs, the cyclomatic complexity is given by
M = 34 – 24 + 2*(1)
M = 12
Hence, the cyclomatic complexity for this control-flow is **12**.

We have 12 independent paths in the scenario. To achieve efficient path coverage, we have to have a minimum of 12 test cases.

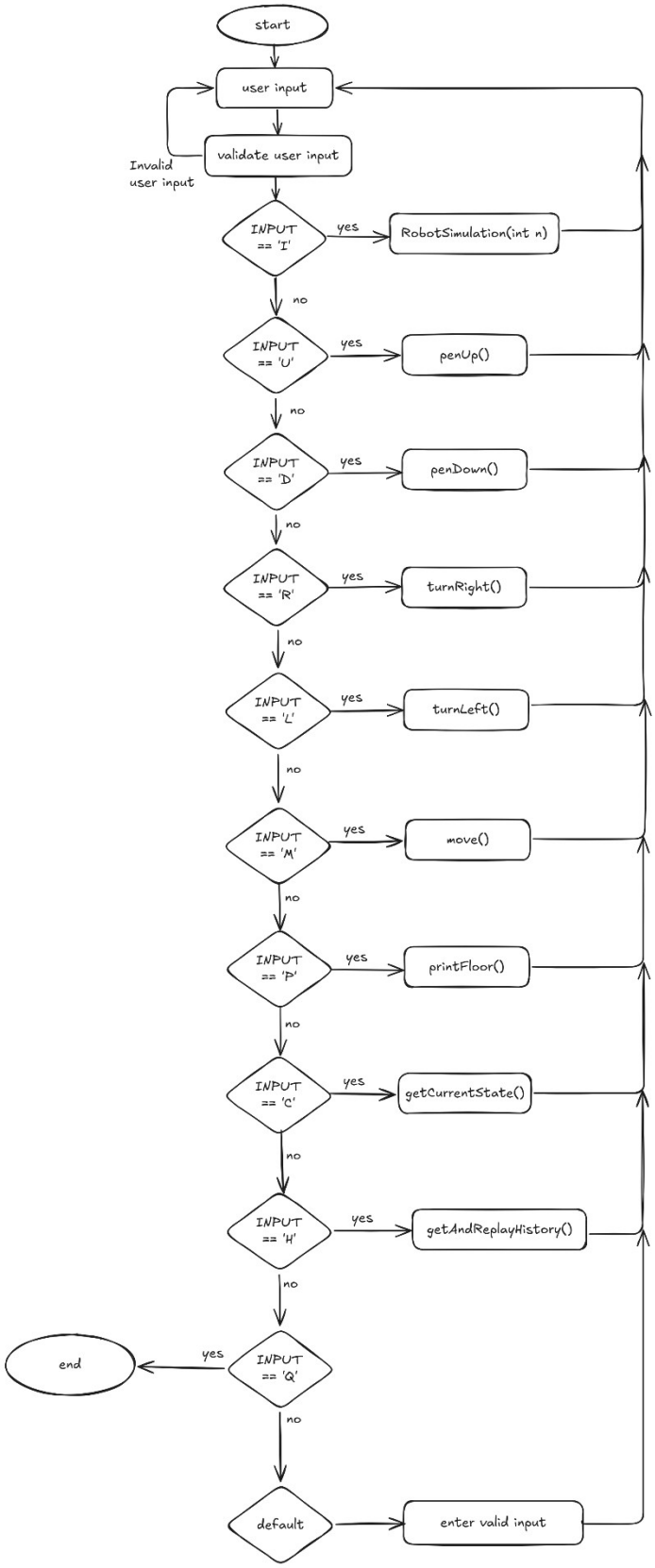


Fig: executable paths

No	Test case	Description	Path covered
1	testInitializeSystem()	Checks robot initialization with default	1 -> 2 -> 3 -> 4 -> 5
2	testPenUpAndDown()	Test pen up and down commands	1 -> 2 -> 3 -> 6 -> 7 -> 3 -> 8 -> 9
3	testMoveWithoutPen()	Moves robot without pen down and check if the floor remains unmarked	1 -> 2 -> 3 -> 14 -> 15
4	testMoveWithPen()	Moves robot with pen down and checks path is marked	1 -> 2 -> 3 -> 6 -> 7 -> 3 -> 14 -> 15
5	testMoveBeyondBoundary()	Tests movement beyond floor boundary	1 -> 2 -> 3 -> 14 -> 15
6	testTurnRight()	Test turn right in all directions	1 -> 2 -> 3 -> 10 -> 11 -> 3 -> 10 -> 11 -> 3 -> 10 -> 11 -> 3 -> 10 -> 11
7	testTurnLeft()	Test turn left in all directions	1 -> 2 -> 3 -> 12 -> 13 -> 3 -> 12 -> 13 -> 3 -> 12 -> 13 -> 3 -> 12 -> 13
8	testTurnRightAndMove()	Combines turn and move commands	1 -> 2 -> 3 -> 14 -> 15 -> 3 -> 10 -> 11 -> 3 -> 14 -> 15
9	testPrintFloor()	Verifies floor print functionality	1 -> 2 -> 3 -> 16 -> 17
10	testPrintCurrentStatePenUp()	Checks current state when pen is up	1 -> 2 -> 3 -> 14 -> 15 -> 3 -> 18 -> 19
11	testPrintCurrentStatePenDown()	Checks current state when pen is down	1 -> 2 -> 3 -> 6 -> 7 -> 3 -> 18 -> 19

12	testGetAndReplayHistory()	Tests history tracking and replaying	1 -> 2 -> 3 -> 6 -> 7 -> 3 -> 14 -> 15 -> 3 -> 10 -> 11 -> 3 -> 20 -> 21

2.4 CONDITION COVERAGE

Condition (Location in Code)	True (Test Cases)	False (Test Cases)
1. `if (input == null		input.trim().isEmpty())` (in processCommand method)
2. if (currentRobot == null) (before commands U, D, R, L, M, P, C, H)	testProcessCommandAllScenarios() before "I 10" (robot not initialized, condition is true)	testProcessCommandAllScenarios() after "I 10" (robot is now initialized, condition is false)
3. if (parts.length < 2) (checking missing arguments for "I" or "M" commands)	testProcessCommandAllScenarios() with "I" (missing argument for floor size, condition is true)	testProcessCommandAllScenarios() with "I 10" (argument provided, condition is false)
	testProcessCommandAllScenarios() with "M" (missing argument for steps, condition is true)	testProcessCommandAllScenarios() with "M 3" (argument provided, condition is false)
4. if (newX >= 0 && newX < floor.length && newY >= 0 && newY < floor.length) (in move method)	testMoveWithoutPen() when moving within the grid (newX and newY remain within bounds, all sub-conditions are true)	testMoveBeyondBoundary() (robot moves beyond grid boundary, so at least one sub-condition is false)
5. if (penDown) (in move method when marking the floor)	testMoveWithPen() (pen is down before moving, so floor gets marked, condition is true)	testMoveWithoutPen() (pen is up, so floor does not get marked, condition is false)
6. if (robot.getCurrentState() != null) (checking valid state retrieval)	testPrintCurrentStatePenUp() (robot exists and has a valid state, condition is true)	(Not applicable, as the method always returns a valid state once initialized)
7. if (robot.isFloorMarked(x, y)) (checking if a position is marked)	testMoveWithPen() (robot moved with pen down, so at least one floor cell is marked, condition is true)	testMoveWithoutPen() (robot moved with pen up, so no floor marking occurred, condition is false)
8. if (command.equalsIgnoreCase("Q")) (exit condition in processCommand method)	testMainMethodCoverage() (when "Q" is entered, the loop ends, condition is true)	testMainMethodCoverage() with "U", "D", "M 3" etc. (loop continues, condition is false)
9. if (command.equalsIgnoreCase("XYZ")) (checking unknown command case)	testProcessCommandAllScenarios() (invalid command is entered, prints "Unknown command", condition is true)	(Not applicable, as valid commands are always recognized)
10. if (robot == null && command.equals("Q")) (handling quit before initialization)	testProcessCommandAllScenarios() (robot is null, "Q" is entered, condition is true)	testProcessCommandAllScenarios() (robot is initialized before "Q", condition is false)

2.5 AI TOOL USAGE

The following AI tools were used during development:

- ChatGPT: Used for clarifying Java syntax and debugging.

2.6 LINKS AND REFERENCES

[1] GitHub URL: https://github.com/sammy-9930/COEN6171-Project/tree/main
[2] Wakatime Dashboard: https://wakatime.com/dashboard