



## Software Testing[CS - 731]

### Symbolic execution on Source Code

Team name: Syms

#### Team Members:

Sunnidhya Roy ([Sunnidhya.Roy@iiitb.ac.in](mailto:Sunnidhya.Roy@iiitb.ac.in)) – MT2023079

Samarpita Bhaumik ([Samarpita.Bhaumik@iiitb.ac.in](mailto:Samarpita.Bhaumik@iiitb.ac.in)) – MT2023053

Github link: <https://github.com/samarpita-bhaumik/Software-Testing/tree/master>

OneDrive Link (to show all the possible paths for App class, RaceGame and SnakesandLadder): [All possible paths](#)

#### Contributions:

The exploration of the Java Path Finder and Symbolic Path finder was done by both of us but we found out that it is an open issue as only Java 8 supports JPF and SPF, hence we implemented a custom **Symbolic Test Engine** and **Symbolic Path Tester** classes to perform Symbolic execution.

As it is based on random values manual testing of the game on various corner cases have been tested by each of us on our respective modules.

1. **Sunnidhya Roy:** Implementation of the Race game with its corresponding unit test cases and the Symbolic execution to get the possible input values which will lead to maximum path coverage.
2. **Samarpita Bhaumik:** Implementation of the Snakes and Ladders game with its corresponding unit test cases and the Symbolic execution to get the possible input values which will lead to maximum path coverage.

# Abstract

This project involves the development and testing of two interactive games: *Snakes and Ladders* and *Race*. The primary focus was to ensure robust code quality and comprehensive path coverage through two methodologies: unit testing and symbolic execution. The unit testing phase systematically verified the correctness of individual components and functionality within the game logic. Symbolic execution was employed as a complementary approach to explore all possible execution paths in the source code. Given the complexity of the games' implementation, which includes rich control structures such as nested decision statements, loops, and function calls, symbolic execution proved instrumental in identifying edge cases and ensuring exhaustive testing of all logical paths.

Our Project highlights the effectiveness of combining conventional testing techniques with advanced symbolic execution to achieve thorough validation of software systems with intricate control flows.

## Introduction

Software testing is a critical phase in the development lifecycle, ensuring the reliability, functionality, and robustness of applications. This project focuses on testing two interactive games, *Snakes and Ladders* and *Race*, employing **unit testing** and **symbolic execution** to achieve thorough path coverage and maintain high-quality standards in the source code.

**Unit testing** is a software testing method where individual components or modules of the application are tested in isolation to verify that they function as expected. By systematically examining each unit, developers can identify and fix issues early in the development process, improving overall software quality and reducing downstream defects.

In addition to unit testing, this project leverages **symbolic execution** to evaluate path coverage in the source code. Symbolic execution is a program analysis technique that systematically explores all possible execution paths by treating input values as symbolic variables rather than concrete values. This enables the identification of all reachable paths within the program, ensuring comprehensive validation of complex control structures, including nested decision statements, loops, and function calls.

The combination of these approaches ensures that the intricate logic underlying the two games is thoroughly tested, uncovering edge cases and ensuring robust performance across all scenarios. This dual-method testing framework demonstrates the importance of integrating traditional and advanced techniques to achieve exhaustive software validation.

## Code Explanation

### Explanation of Each Class in the Race Game

#### Player Class

The Player class represents an individual player in the game. It encapsulates the player's name and their current position on the board, which starts at 0 when the game begins. The class provides

methods to move the player based on the dice roll, retrieve the player's name and position, and reset the position back to the starting point. The move method ensures that a player's position does not exceed the maximum position (100) by validating the sum of the current position and dice roll.

### **Dice Class**

The Dice class simulates the rolling of a six-sided die using the Random class from Java's utility package. The roll method generates a random integer between 1 and 6, representing the possible outcomes of a dice roll. This class introduces an element of chance to the game, which determines how far a player can move in their turn.

### **Board Class**

The Board class manages the overall game state, including the players, the dice, and the turn order. It is responsible for executing game logic such as rolling the dice for the current player, advancing their position, and checking if any player has reached the winning position of 100. The class also includes a method to reset the board to its initial state, enabling replayability. It ensures that turns progress sequentially among the players by updating the currentTurn index in a cyclic manner.

### **RaceGameGUI Class**

The RaceGameGUI class implements the graphical user interface (GUI) for the game using the Java Swing library. It consists of several components, such as a label to display the current player's turn, a button to roll the dice, and a text area to log game events. The LudoBoardPanel is a nested class within RaceGameGUI that provides a visual representation of the game board and the players' positions.

The updateUI method dynamically updates the GUI after every dice roll, reflecting the changes in player positions and adding a log entry. The checkForWin method determines if a player has won the game, displaying a congratulatory message and resetting the board. The game is initialized and launched from the main method, which creates an instance of the RaceGameGUI and runs it on the Swing event-dispatch thread.

### **LudoBoardPanel Class**

The LudoBoardPanel is a custom Swing JPanel that visually represents the game board. It draws a 10x10 grid using lines to form cells and renders each player's position as a colored circle on the grid. The colors represent different players, and their positions are calculated based on the player's current index on the board. This class is updated and repainted dynamically to reflect the players' movements.

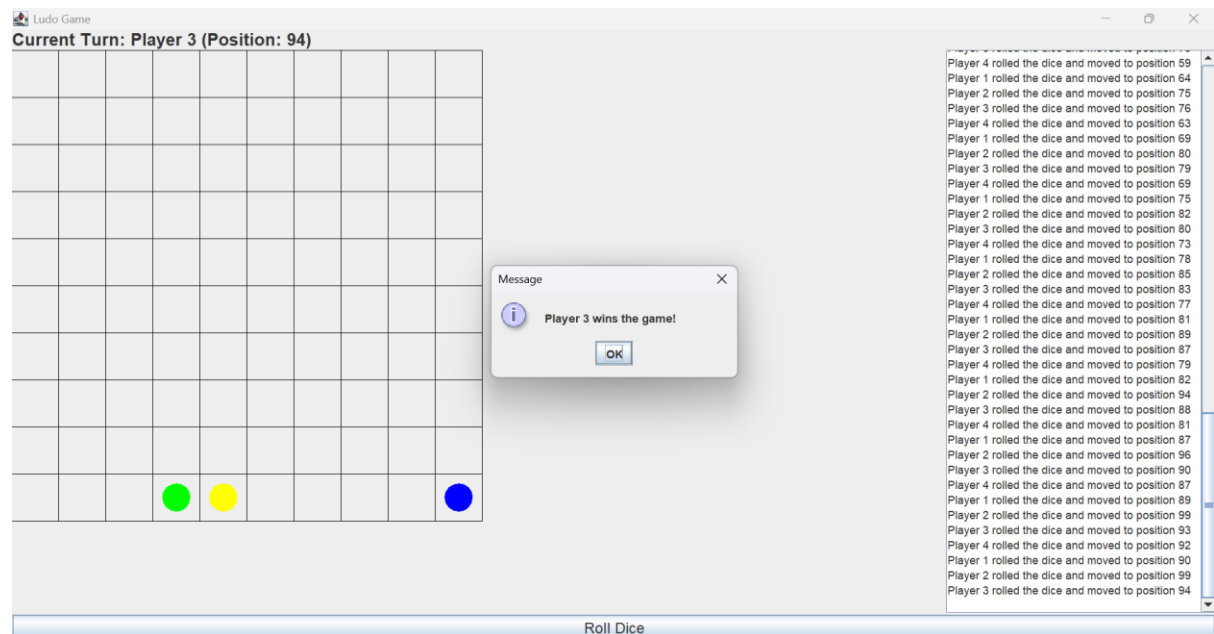
## **How the Game Works**

The game begins by initializing four players, a Board instance, and a GUI represented by the RaceGameGUI class. Each player starts at position 0. The game progresses turn by turn, with each player rolling the dice using the roll method of the Dice class. The playTurn method in the Board class moves the player forward based on the dice roll, ensuring that the position does not exceed 100.

The GUI provides real-time feedback to the players. The current turn, the rolled number, and the player's new position are displayed through labels and a game log. The LudoBoardPanel updates to visually represent the players' positions on a 10x10 grid. If a player reaches position 100, the game announces the winner through a dialog box and resets the board, allowing a new game to begin.

This seamless integration of game logic (Board, Player, and Dice classes) and user interface (RaceGameGUI and LudoBoardPanel classes) ensures an engaging and interactive experience for the players.

### GUI of the game:



## Explanation of the Snakes and Ladders Code

The Snakes and Ladders game is implemented in Java with a structured approach, featuring separate classes for the board, game logic, players, and a graphical user interface (GUI). The code models the classic board game, including snakes, ladders, and penalties, while allowing two players to compete until one wins. Here's an in-depth explanation of the code and how it works.

### Board Class

The Board class represents the playing surface of the game. It defines the size of the board and maps for storing the locations of snakes and ladders. A snake is defined by a starting and ending position where the player slides downward when landed upon. Similarly, ladders are defined by a starting and ending position where the player moves upward. Penalty positions are also introduced, where a player moves back three spaces when they land on such spots. Methods such as `addSnake`, `addLadder`, and `addPenaltyPosition` allow customization of the board, while `getFinalPosition` calculates a player's new position after considering snakes, ladders, and penalties.

### Game Class

The Game class manages the gameplay. It initializes the board and players and facilitates the turn-based flow of the game. Dice rolls determine how far a player moves, and the board rules (snakes, ladders, or penalties) are applied to adjust the player's position. The class tracks the current player and alternates turns unless a player rolls a six, which grants an additional turn. The `playNextTurn` method handles the dice roll, position updates, and checks for a win condition when a player reaches the board's last square. It also supports fixed dice rolls for testing purposes through the `setDiceRoll` method.

## Player Class

The Player class represents individual players. Each player has a name, a current position on the board, and a flag to indicate if they have earned an extra turn. Methods in this class manage player movement and state during the game.

## GameGUI class

The GameGUI class provides a graphical representation of the board and interactive gameplay using Java Swing. The board is displayed as a grid of numbered cells, and player positions are updated visually as the game progresses. A "Roll Dice" button allows players to take turns, with messages displayed to indicate dice rolls, position changes, and game events like landing on snakes, ladders, or penalties. The game ends when a player reaches the last square, at which point the interface announces the winner.

## How the Game Works

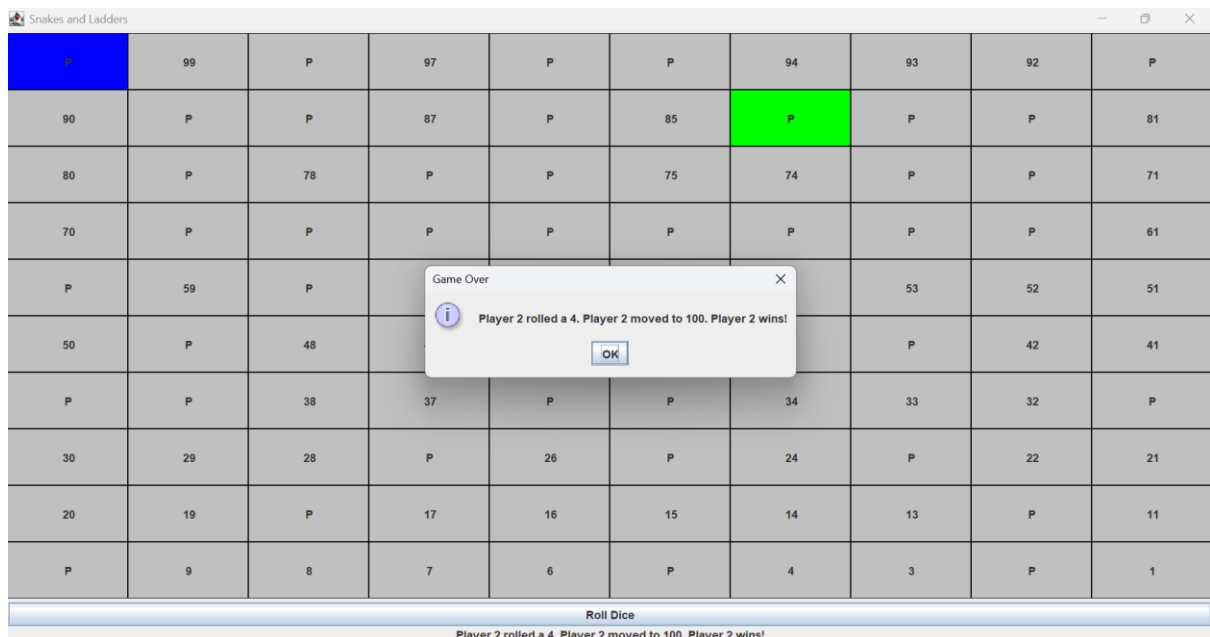
To play the Snakes and Ladders game, players take turns rolling a virtual dice by clicking the "Roll Dice" button in the graphical interface. The dice roll determines how many steps a player moves forward on the board. The game enforces the following rules:

1. **Start Position:** All players start at position 0. Each dice roll determines their movement forward.
2. **Snakes and Ladders:** If a player lands on a square with a snake, they slide down to the snake's tail. Conversely, if they land on a square with a ladder, they climb up to the top of the ladder.
3. **Penalty Positions:** Certain squares on the board are penalty spots. Landing on these positions moves the player three steps backward.
4. **Exact Roll to Win:** To win, a player must roll the exact number needed to reach the last square (e.g., position 100). If the dice roll exceeds the required number, the player stays in place.
5. **Rolling a Six:** Rolling a six grants an additional turn. Players can roll the dice again until they roll a number other than six.
6. **Turn Switching:** Turns alternate between players unless one rolls a six. The game tracks and announces whose turn it is.

The game ends when a player reaches position 100. A congratulatory message is displayed in the interface, and the "Roll Dice" button is disabled to prevent further turns. Players can restart the game manually to play again.

This implementation makes the game engaging and visually interactive, combining strategy and chance as players navigate the board while dealing with snakes, ladders, and penalties.

## GUI of the game:



# Unit Testing

## Overview of Unit Tests for the Race Game

The provided code contains a series of unit tests written using JUnit 5 to validate the functionality of the RaceGame application. The tests cover various aspects of the game, including board behavior, player mechanics, dice functionality, and GUI components. Each test ensures the integrity of the game logic and the graphical user interface.

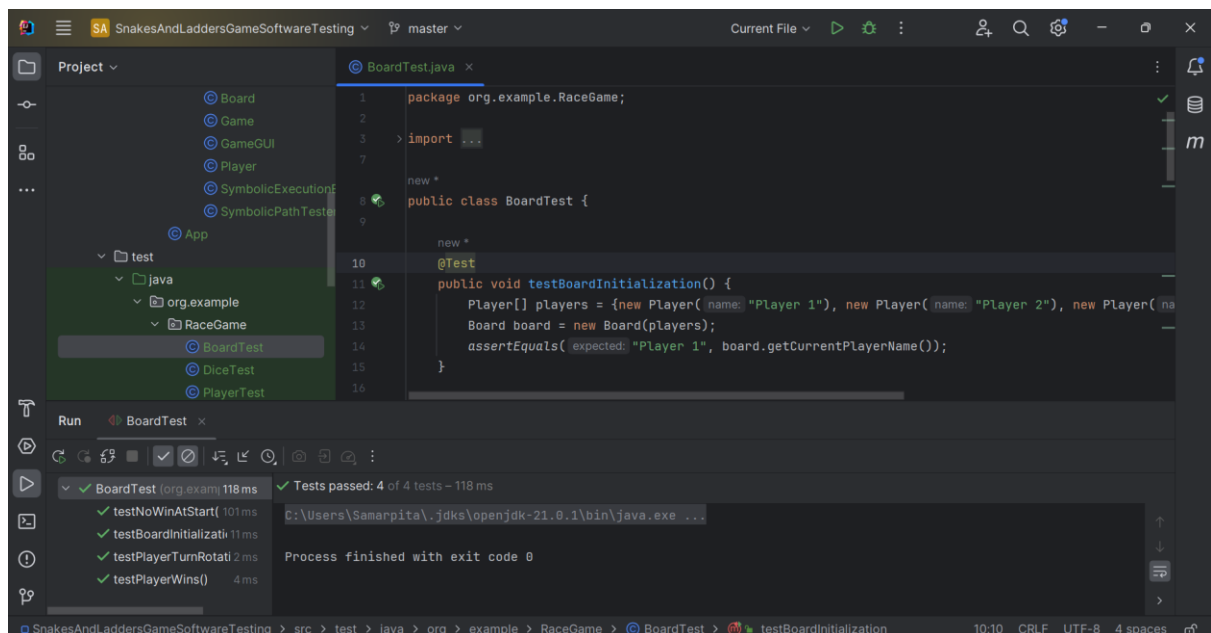
### Board Test

The BoardTest class verifies the board's initialization, player turn rotation, and win conditions.

### Key Tests:

1. **testBoardInitialization:** Checks if the board initializes correctly with the first player ("Player 1") as the starting player. Uses assertEquals to ensure the name of the current player matches expectations.
2. **testPlayerTurnRotation:** Simulates multiple turns and verifies the correct rotation of turns among players. After each turn, the test ensures the getCurrentPlayerName method reflects the next player.
3. **testPlayerWins:** Simulates a win condition by moving "Player 1" to position 100. Uses assertTrue to confirm the checkWin method correctly identifies a winner.
4. **testNoWinAtStart:** Ensures that no player has won the game at the start. Uses assertFalse to verify the checkWin method returns false when no player reaches the winning position.

## Output:



The screenshot shows an IDE window for a project named "SnakesAndLaddersGameSoftwareTesting". The "Project" view on the left shows a package structure: `org.example.RaceGame` containing `BoardTest`, `DiceTest`, and `PlayerTest`. The editor displays `BoardTest.java` with the following code:

```
1 package org.example.RaceGame;
2
3 > import ...
7
8 new *
9 public class BoardTest {
10
11 new *
12 @Test
13 public void testBoardInitialization() {
14     Player[] players = {new Player( name: "Player 1"), new Player( name: "Player 2"), new Player( name: "Player 3")};
15     Board board = new Board(players);
16     assertEquals( expected: "Player 1", board.getCurrentPlayerName());
17 }
```

The "Run" view at the bottom shows the test results for `BoardTest` (org.example) in 118 ms. All tests passed:

- testNoWinAtStart (101 ms)
- testBoardInitialization (11 ms)
- testPlayerTurnRotation (2 ms)
- testPlayerWins() (4 ms)

The console output shows: "Process finished with exit code 0".

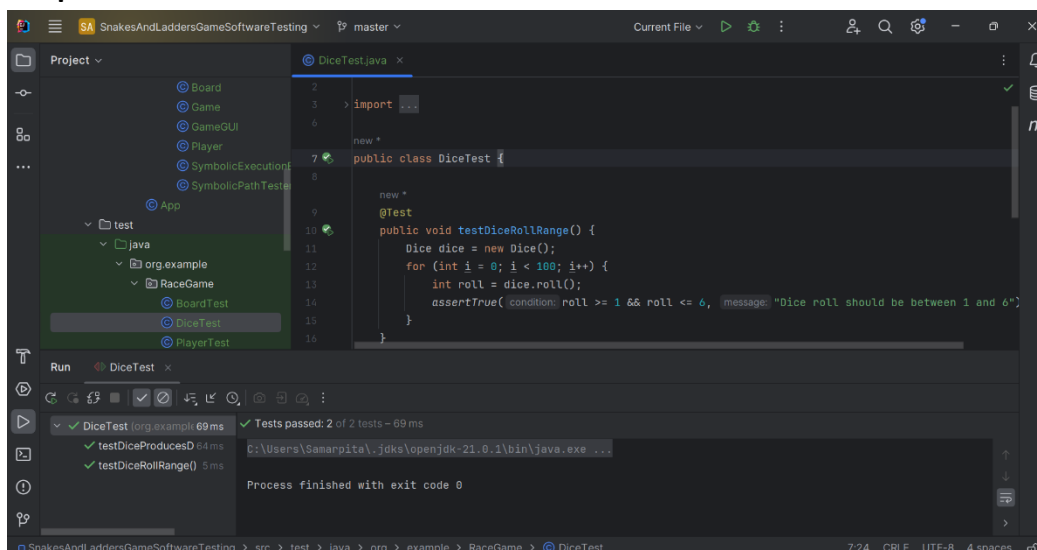
## Dice Test

The `DiceTest` class verifies the behavior of the dice used in the game.

### Key Tests:

1. **testDiceRollRange:** Rolls the dice 100 times and checks if all rolls are within the valid range (1–6). Uses `assertTrue` for validation and includes a custom error message for failure.
2. **testDiceProducesDifferentValues:** Rolls the dice twice and ensures the results are not identical. Although randomness might occasionally cause identical rolls, this test checks for general variability.

## Output:



The screenshot shows the same IDE window, but now displaying `DiceTest.java`. The code is as follows:

```
2
3 > import ...
6
7 new *
8 public class DiceTest {
9
10 new *
11 @Test
12 public void testDiceRollRange() {
13     Dice dice = new Dice();
14     for (int i = 0; i < 100; i++) {
15         int roll = dice.roll();
16         assertTrue( condition: roll >= 1 && roll <= 6, message: "Dice roll should be between 1 and 6");
17     }
18 }
```

The "Run" view shows the test results for `DiceTest` (org.example) in 69 ms. All tests passed:

- testDiceProducesDifferentValues (64 ms)
- testDiceRollRange() (5 ms)

The console output shows: "Process finished with exit code 0".

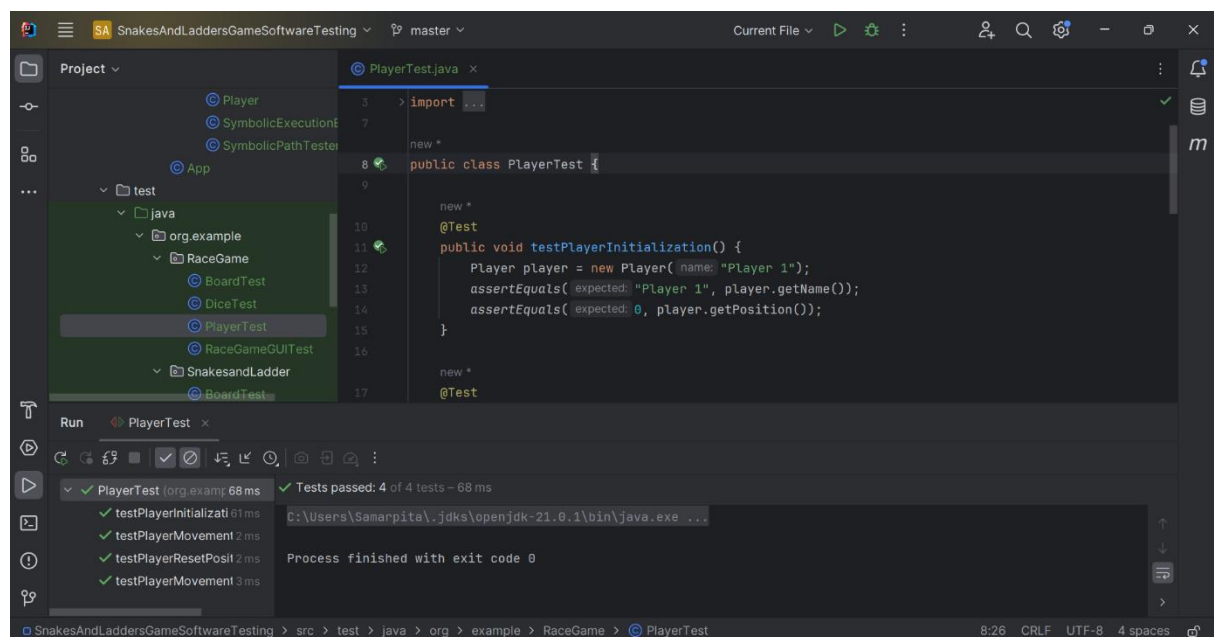
## Player Test

The PlayerTest class validates player-related functionality, such as initialization, movement, and position resets.

### Key Tests:

1. **testPlayerInitialization:** Ensures a player is correctly initialized with a name and starting position (0). Uses assertEquals to check both the player's name and position.
2. **testPlayerMovementWithinBounds:** Moves the player to a new position and verifies the updated position. Confirms that movement logic works as expected.
3. **testPlayerMovementExceeding100:** Simulates a scenario where a player moves beyond position 100. Ensures the position does not change if the movement exceeds the winning position.
4. **testPlayerResetPosition:** Moves the player to a position and then resets their position to 0. Verifies the reset functionality works correctly.

### Output:



The screenshot displays an IDE window for a project named 'SnakesAndLaddersGameSoftwareTesting'. The 'Project' view on the left shows a package structure with 'test' containing 'java' which includes 'org.example' and 'RaceGame'. The 'RaceGame' package contains 'BoardTest', 'DiceTest', 'PlayerTest', 'RaceGameGUITest', and 'SnakesandLadder'. The 'PlayerTest.java' file is open in the editor, showing the following code:

```
3  > import ...
7
8  new *
9  public class PlayerTest {
10
11  new *
12  @Test
13  public void testPlayerInitialization() {
14      Player player = new Player( name: "Player 1");
15      assertEquals( expected: "Player 1", player.getName());
16      assertEquals( expected: 0, player.getPosition());
17  }
18
19  new *
20  @Test
```

The 'Run' view at the bottom shows the test results for 'PlayerTest' (org.example) with a total time of 68 ms. All 4 tests passed:

- testPlayerInitialization 61 ms
- testPlayerMovement 2 ms
- testPlayerResetPosit 2 ms
- testPlayerMovement 3 ms

The process finished with exit code 0.

## GUI Test

The RaceGameGUITest class validates the GUI functionality, ensuring the graphical components are present and functional.

### Key Tests:

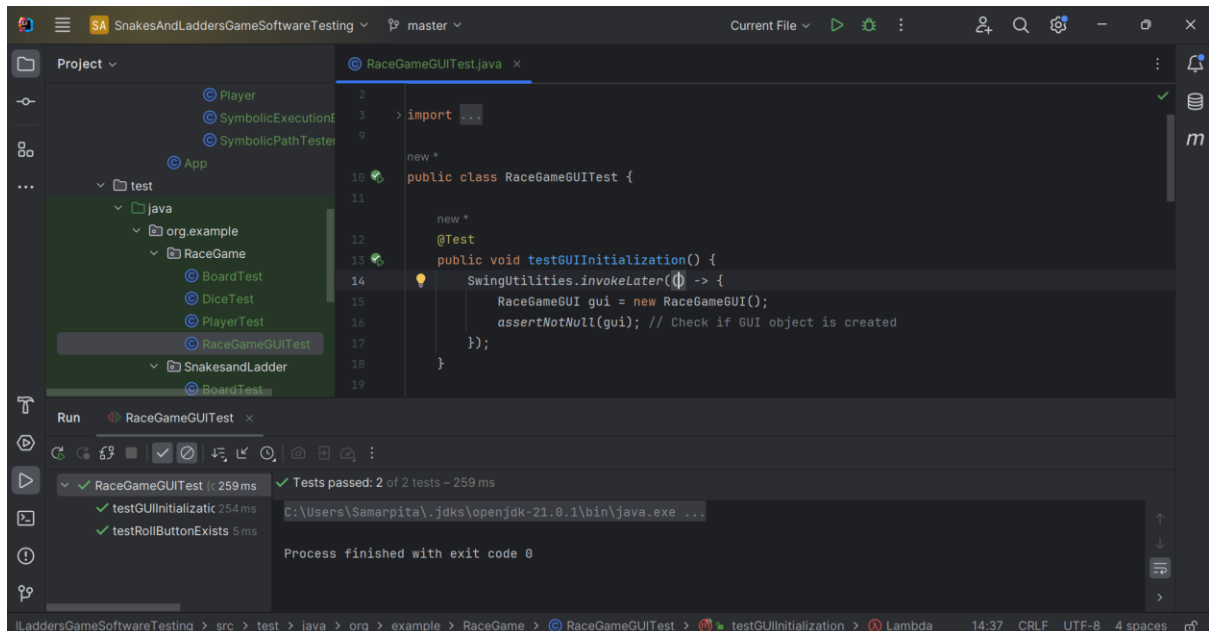
1. **testGUIInitialization:** Checks if the GUI object initializes correctly. Uses SwingUtilities.invokeLater to ensure GUI tests are run on the Event Dispatch Thread (EDT).
2. **testRollButtonExists:** Verifies that the "Roll Dice" button exists in the GUI. Uses a helper method findComponent to recursively search for the button in the GUI container.



## Helper Method:

- **findComponent:** A recursive function that searches for a component of a specific type (e.g., JButton) in the GUI hierarchy. Facilitates locating GUI components for validation.

## Output:



## How the Tests Ensure Code Quality

The tests ensure the core game logic (turns, dice rolls, movement) behaves as expected. The win condition logic is thoroughly validated to prevent incorrect results. Edge cases, such as exceeding the board's bounds or checking for a win at the start, are explicitly tested. Dice tests confirm the random behavior of the dice while ensuring valid outcomes. The presence of critical GUI components (e.g., "Roll Dice" button) is validated to ensure user interaction works correctly.

By systematically covering all major components, these unit tests provide a strong foundation for maintaining the integrity of the Race Game application.

## Overview of Unit Tests for Snakes and Ladders

The unit tests validate various functionalities of a Snakes and Ladders game implemented in Java. These tests focus on critical aspects such as the board's behavior, the GUI elements, player mechanics, and the game's logic, ensuring correctness and robustness. Below is a detailed explanation of each test class and its purpose.

### Board Tests

The BoardTest class verifies the core functionality of the game board, ensuring the proper handling of snakes, ladders, penalties, and player positions.

1. **testAddSnake:** Tests the addition of a snake on the board. For example, if a player lands on position 99, they should be moved down to position 54. This validates the correct mapping of snake positions.
2. **testAddLadder:** Tests the addition of a ladder. For instance, landing on position 3 should transport the player to position 22. This ensures ladders are mapped correctly.
3. **testAddPenaltyPosition:** Verifies penalty positions that move players backward. A penalty at position 50 should move the player to position 47.
4. **testNoSnakeLadderPenalty:** Ensures positions without snakes, ladders, or penalties do not alter the player's position.
5. **testInvalidSnake:** Prevents invalid snake configurations where the head is below the tail. For example, a snake from position 20 to 25 is invalid and should not change the player's position.
6. **testInvalidLadder:** Prevents invalid ladder configurations where the base is above the top. For instance, a ladder from position 30 to 15 is invalid.

## Output:

The screenshot shows an IDE with the following components:

- Project Explorer:** Shows a package structure with `org.example` containing `RaceGame` (with `BoardTest`, `DiceTest`, `PlayerTest`, and `RaceGameGUITest`) and `SnakesAndLadder` (with `BoardTest`, `GameGUITest`, and `GameTest`).
- Editor:** Displays `BoardTest.java` with the following code:
 

```

2
3 > import ...
7
8 new *
9 public class BoardTest {
10
11 new *
12 @Test
13 public void testAddSnake() {
14     Board board = new Board(size: 100);
15     board.addSnake(99, 54);
16 }
17
18 }
```
- Run Console:** Shows the execution of `BoardTest` with the following output:
 

```

C:\Users\Samarpita\.jdk\openjdk-21.0.1\bin\java.exe ...
Snake! You slide from 99 to 54
Penalty! Move back 3 spaces.
Ladder! You climb from 3 to 22
Process finished with exit code 0
```
- Test Results:** A table showing the results of 6 tests:
 

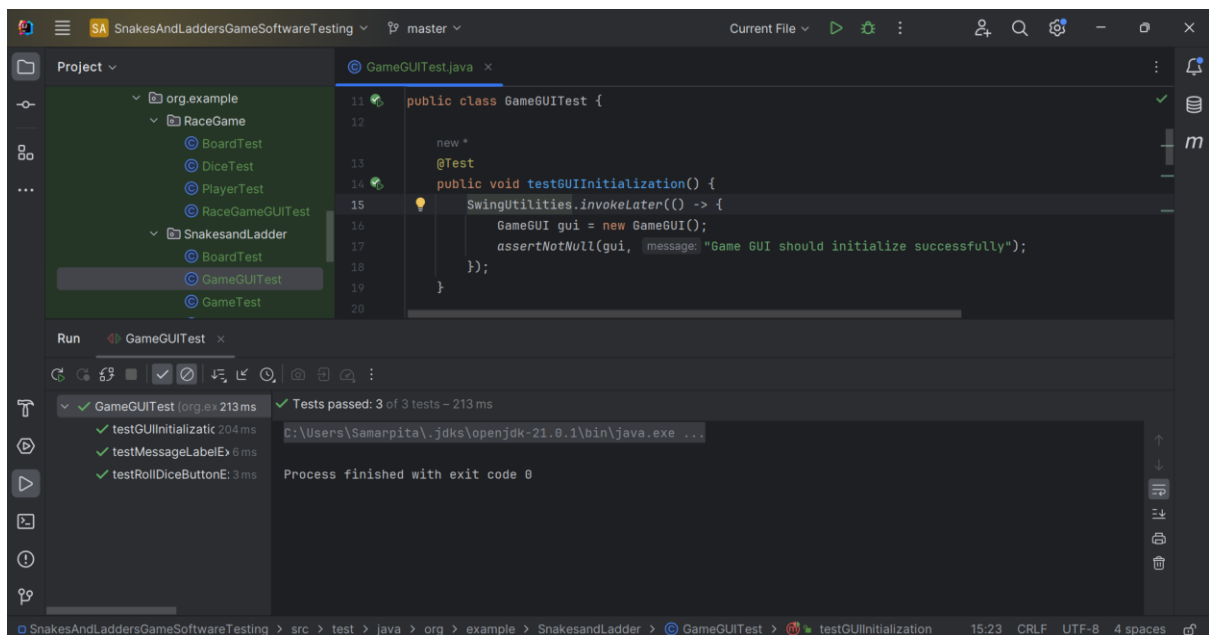
Test Name	Duration	Status
BoardTest (org.exam	108 ms	✓
testAddSnake()	93 ms	✓
testAddPenaltyPosit	3 ms	✓
testAddLadder()	4 ms	✓
testInvalidLadder()	2 ms	✓
testNoSnakeLadder()	3 ms	✓
testInvalidSnake()	3 ms	✓

## GUI Tests

The `GameGUITest` class tests the Graphical User Interface (GUI) to ensure that the necessary components for user interaction are present and functional.

1. **testGUIInitialization:** Verifies that the game GUI initializes successfully.
2. **testRollDiceButtonExists:** Checks for the presence of the "Roll Dice" button, a critical interactive component of the GUI.
3. **testMessageLabelExists:** Ensures that a label displaying the current player's turn exists in the GUI. For example, the label should initially display "Player 1's turn."
4. **findComponent:** A helper method used to recursively locate specific components like buttons and labels within the GUI's container hierarchy.

## Output:

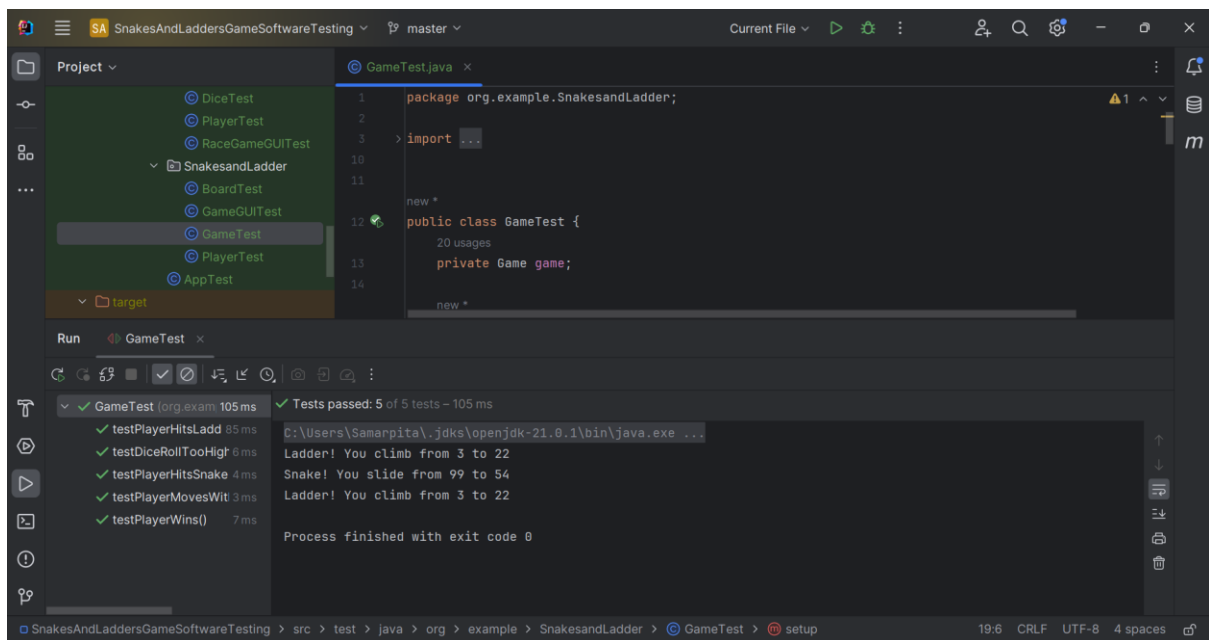


## Game Tests

The GameTest class ensures the correctness of the game's logic, including player movement, dice rolls, and winning conditions.

1. **testPlayerMovesWithoutHittingSnakeOrLadder:** Simulates a dice roll (e.g., rolling 3) to ensure that the player moves to the correct position without encountering snakes or ladders.
2. **testPlayerHitsLadder:** Verifies that landing on a ladder base moves the player to the top of the ladder. For example, moving to position 3 should transport the player to position 22.
3. **testPlayerHitsSnake:** Ensures that landing on a snake's head moves the player to its tail. For example, landing on position 99 should move the player to position 54.
4. **testPlayerWins:** Simulates a scenario where a player reaches position 100 to validate the winning condition and the appropriate message ("Player 1 wins!").
5. **testDiceRollTooHigh:** Tests a situation where a dice roll exceeds the remaining moves required to reach 100. The player's position should remain unchanged, and the message should indicate the invalid move.

## Output:



The screenshot shows an IDE with the project 'SnakesAndLaddersGameSoftwareTesting'. The 'Project' view on the left shows a package structure with 'SnakesandLadder' containing 'GameTest'. The 'Run' view at the bottom shows the execution of 'GameTest' with the following output:

```
Process finished with exit code 0
```

The 'Run' view also shows a list of tests passed:

- testPlayerHitsLadd 95 ms
- testDiceRollTooHigh 6 ms
- testPlayerHitsSnake 4 ms
- testPlayerMovesWith 3 ms
- testPlayerWins() 7 ms

The 'GameTest.java' file is open in the editor, showing the following code:

```
package org.example.SnakesandLadder;

import ...

new *

public class GameTest {
    20 usages
    private Game game;

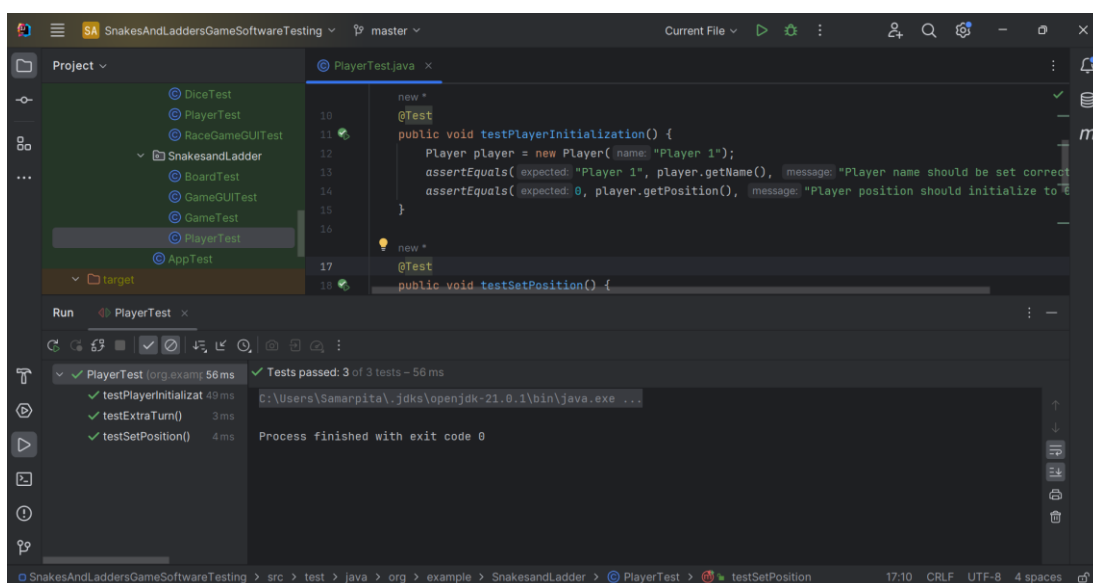
    new *
```

## Player Tests

The PlayerTest class validates the mechanics related to individual players, such as initialization, position updates, and extra turns.

1. **testPlayerInitialization:** Confirms that a new player is initialized with the correct name and a starting position of 0.
2. **testSetPosition:** Ensures that a player's position updates correctly when set programmatically.
3. **testExtraTurn:** Verifies that the extra-turn functionality is implemented correctly. A player with an extra turn should have the corresponding flag set to true.

## Output:



The screenshot shows an IDE with the project 'SnakesAndLaddersGameSoftwareTesting'. The 'Project' view on the left shows a package structure with 'SnakesandLadder' containing 'PlayerTest'. The 'Run' view at the bottom shows the execution of 'PlayerTest' with the following output:

```
Process finished with exit code 0
```

The 'Run' view also shows a list of tests passed:

- testPlayerInitializat 49 ms
- testExtraTurn() 3 ms
- testSetPosition() 4 ms

The 'PlayerTest.java' file is open in the editor, showing the following code:

```
new *

@Test
public void testPlayerInitialization() {
    Player player = new Player( name: "Player 1");
    assertEquals( expected: "Player 1", player.getName(), message: "Player name should be set correct
    assertEquals( expected: 0, player.getPosition(), message: "Player position should initialize to 0

new *

@Test
public void testSetPosition() {
```

## Purpose of the Tests

These tests are essential to ensure the game's functionality and user experience. The board tests validate game rules (e.g., snake and ladder behavior), the GUI tests ensure a seamless user interface, the game tests confirm adherence to game logic, and the player tests verify individual player mechanics. Together, these tests create a comprehensive suite for maintaining and improving the game's quality.

## Symbolic Execution for Path Coverage

### App class:

The code provided implements a symbolic execution engine and path tester for the App class, which is the entry point for a simple console-based game menu. The App allows users to choose between playing two games: Snakes and Ladders or 4 Player Race Game. It also handles invalid inputs by displaying an error message and looping back to the menu.

### Symbolic Execution Engine

The symbolic execution engine systematically explores all possible user interactions with the App class. The goal is to generate all potential paths the program might take based on user input, including valid and invalid options. It uses recursive exploration, tracks visited states to avoid redundancy, and records paths for analysis.

### Key Components

#### Recursive Path Exploration:

The explorePaths method simulates user choices and recursively explores all possible sequences of menu interactions.

Depth is limited to avoid infinite recursion, representing a practical simulation of user interactions.

#### State Representation:

The stateKey uniquely identifies a program state using the recursion depth and the current path string.

This helps prevent revisiting previously explored states.

#### User Input Simulation:

Simulates user choices (1, 2, or invalid input).

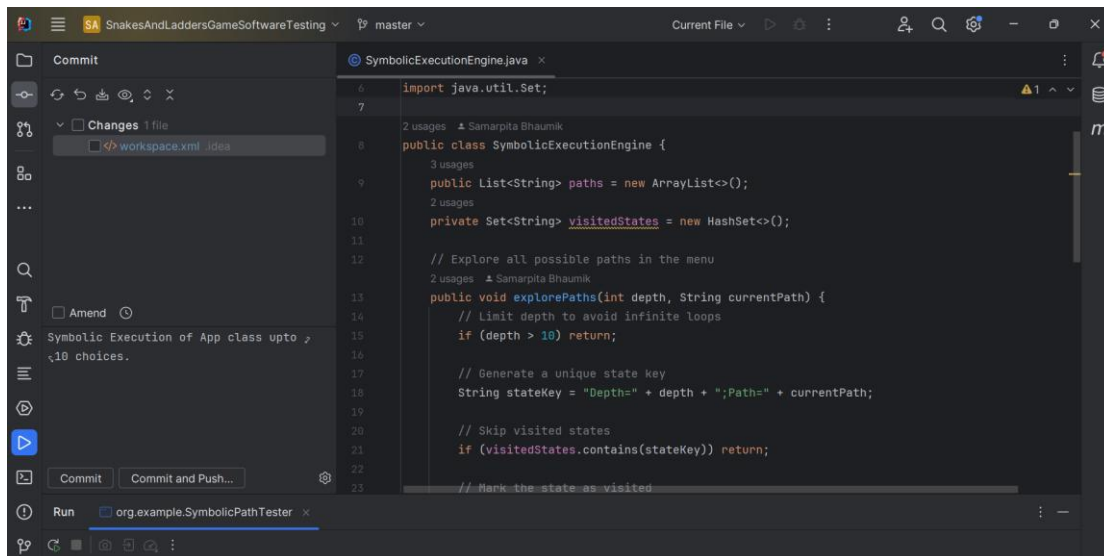
1: Represents launching the Snakes and Ladders GUI.

2: Represents launching the 4 Player Race Game GUI.

3 or higher: Represents invalid input.

**Path Logging:** Records every interaction sequence in a list (paths) for analysis and debugging.

**Backtracking:** Restores the state after each recursive step to ensure all paths are explored independently.



## SymbolicPathTester

The symbolic path tester is a driver program that initializes the engine, triggers the exploration process, and calculates metrics such as path coverage.

### Key Components

#### Engine Initialization:

Creates an instance of the SymbolicExecutionEngine and starts path exploration with depth = 0 and an initial path labeled "START".

#### Path Coverage Calculation:

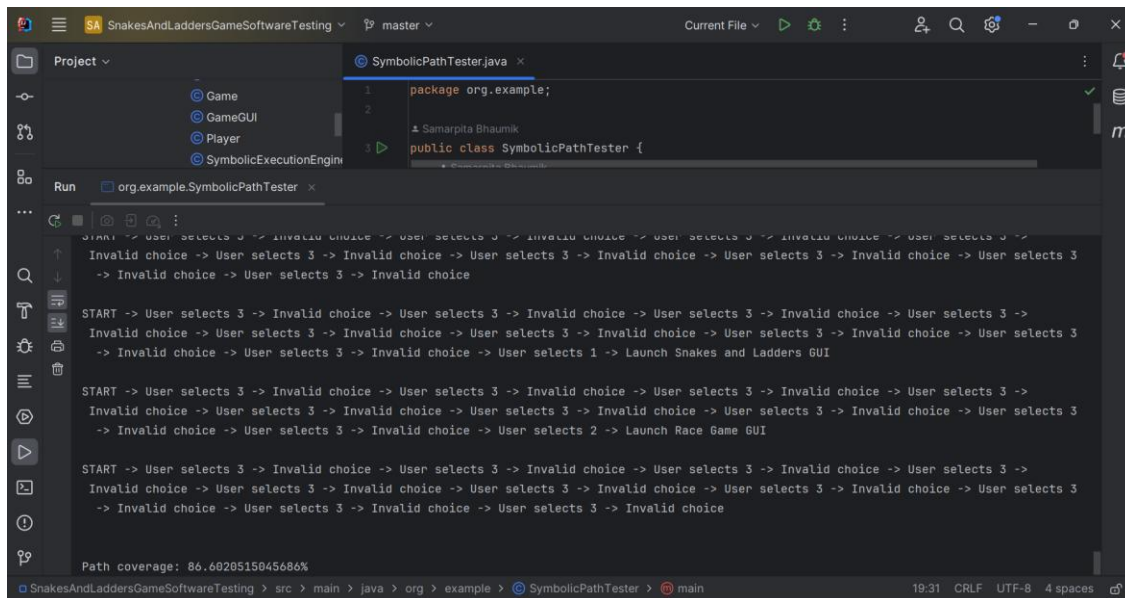
Estimates the percentage of paths covered during the exploration process.

#### Formula:

Path Coverage = ( Total Paths Explored / Estimated Possible Paths ) x 100

The number of possible paths is estimated using  $3^n$ , where  $n$  is the depth, since the menu offers 3 choices per iteration.

### Output:



### Race Game:

The provided code implements a **Symbolic Execution Engine** and a **Symbolic Path Tester** for a race game. It systematically explores various game paths by simulating dice rolls and player movements to ensure the correctness of the game's logic.

## Symbolic Execution Engine

The **SymbolicExecutionEngine** class explores all possible paths in a race game by simulating the movements of players based on dice rolls. It tracks the game's progression, checks win conditions, and ensures unique states are explored using state tracking. The engine uses recursive exploration and backtracking to simulate all possible game scenarios.

## Class Components

## Fields

- **paths:**  
A list that stores all symbolic paths explored by the engine. Each path represents a sequence of player moves and outcomes during the game.
- **visitedStates:**  
A set that tracks unique board states to prevent revisiting and redundant exploration, ensuring efficient simulation.

## Methods

1. **explorePaths(Board board, int depth, String currentPath):**
  - Recursively explores all possible paths in the game starting from the given state.
  - **Logic:** Limits recursion depth to avoid infinite loops or overly complex paths. Generates a unique state key using generateStateKey(Board) and skips previously visited states. Iterates through all possible dice rolls (1–6) and simulates the current

player's move. Updates the path string with the player's dice roll and move. Checks for a win condition using `board.checkWin()` and records the winning path. Advances to the next player, explores further moves recursively, and backtracks to restore the previous state.

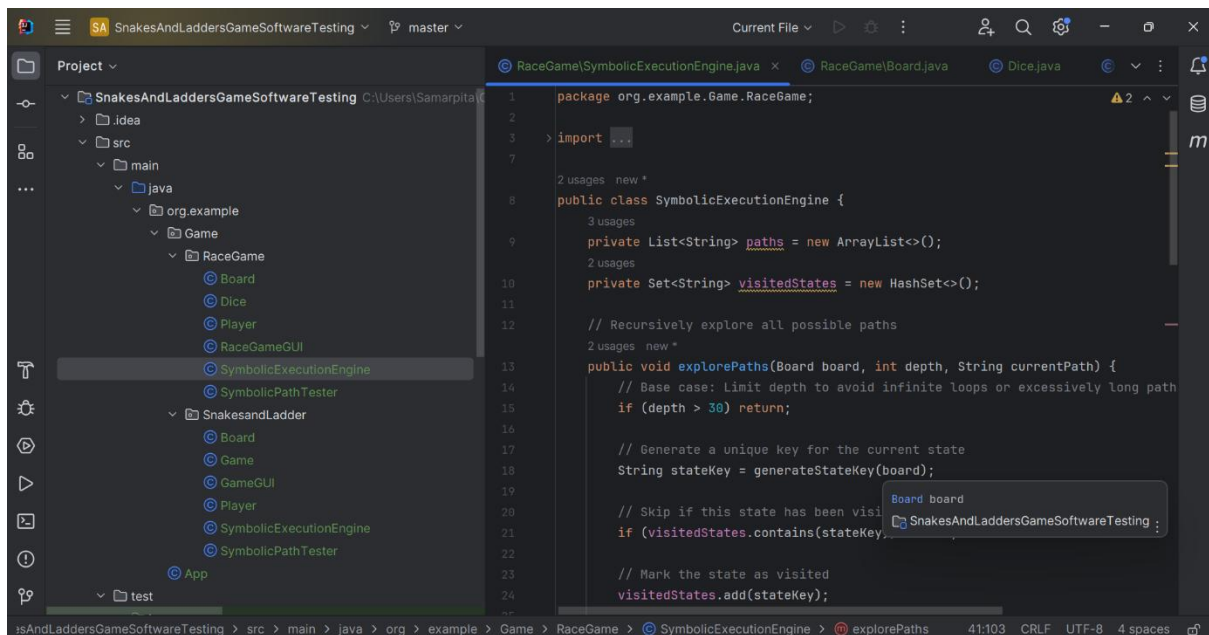
## 2. **generateStateKey(Board board):**

- Generates a unique string representation of the board's current state by encoding each player's position and the current turn.
- This key is used to track visited states and avoid revisiting the same game scenario.

## 3. **printPaths():**

- Prints all symbolic paths stored in the paths list.

Each path represents a complete sequence of moves leading to various outcomes, including wins.



## SymbolicPathTester

The SymbolicPathTester class acts as the main driver for the symbolic execution engine. It initializes the game board, players, and the symbolic execution engine, triggering the exploration of various game paths.

### Steps in the main Method

#### 1. **Create Players:**

- Four players (Player 1, Player 2, Player 3, and Player 4) are created and added to an array, representing the players in the game.

Example:

```
Player player1 = new Player("Player 1");
```

#### 2. **Initialize Board:**



- A Board object is instantiated, passing the array of players. This board manages the players' positions, the current turn, and the win condition.
- 3. **Create Symbolic Execution Engine:**
  - A SymbolicExecutionEngine instance is created to handle the exploration of different game paths.
- 4. **Explore Game Logic:**
  - The explore method of the SymbolicExecutionEngine is invoked to simulate all possible dice rolls for each player's turn, generating all paths based on the dice rolls and player movements.
- 5. **Print Results:**
  - The printPaths() method is called to output all the generated paths, allowing the user to examine the various scenarios and their outcomes. It also outputs the Path Coverage percentage.

## How the Code Works:

### 1. Symbolic Execution

The engine simulates the game by iterating through all possible dice rolls (1-6) for each player on their turn. After each simulated dice roll, the player's position is updated based on the roll, and the resulting path is recorded. The player's position is reset after each exploration to allow the engine to explore other possible scenarios. All possible paths are explored

### 2. Path Tracking

The execution engine tracks all the paths taken during the simulation. Each path is represented as a string that records the player's actions (such as dice rolls and resulting positions) during their turn. For example:

**Turn=Player 1, DiceRoll=4 -> Player 1 moved from 0 to 4**

These paths are stored in a list and can be printed later for analysis.

### 3. Board Logic

After each dice roll simulation, the current turn is updated to switch to the next player. The game continues until all players have had their turns, exploring all possible game paths.

## Output Example (Manual Testing)

For a 4-player game, the paths generated might look like this:

1. Player 1 rolls a 2, moves from 0 to 2.
2. Player 2 rolls a 5, moves from 0 to 5.
3. Player 3 rolls a 6, moves from 0 to 6.
4. Player 4 rolls a 1, moves from 0 to 1.

The paths generated might be printed as:

Turn=Player 1, DiceRoll=2 -> Player 1 moved from 0 to 2.....

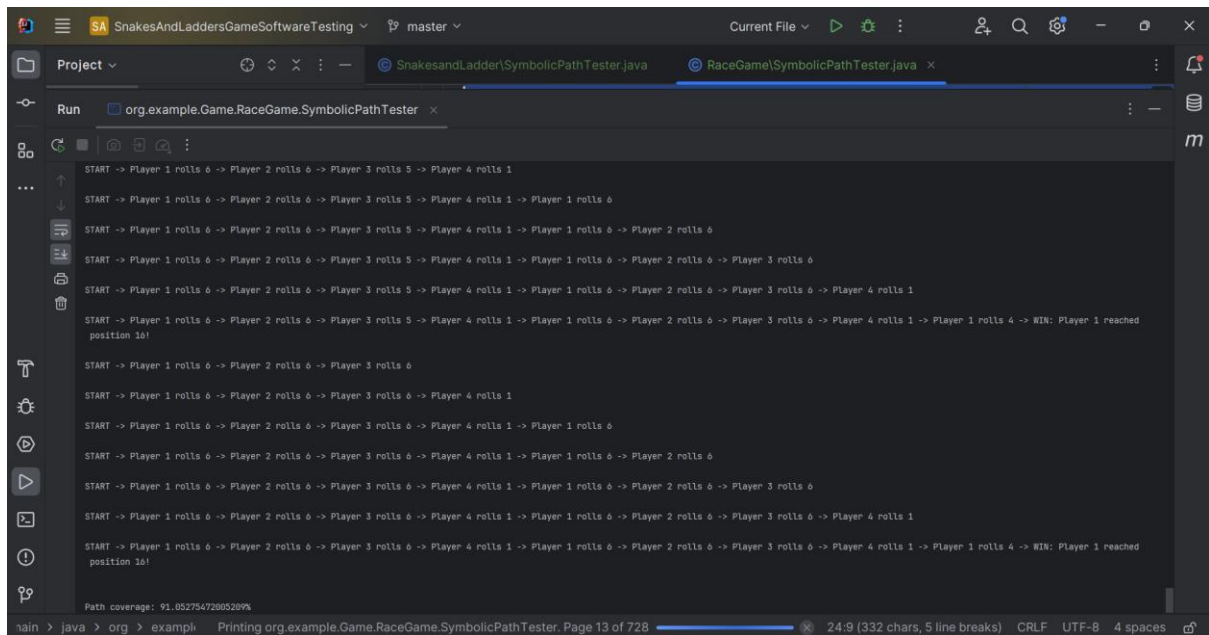
Turn=Player 2, DiceRoll=5 -> Player 2 moved from 0 to 5.....

Turn=Player 3, DiceRoll=6 -> Player 3 moved from 0 to 6.....

Turn=Player 4, DiceRoll=1 -> Player 4 moved from 0 to 1.....

## Output: (Automated Path Coverage using Symbolic Execution Engine)

These represents all the possible paths that were explored



```
Run org.example.Game.RaceGame.SymbolicPathTester
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1 -> Player 1 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 5 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 4 -> WIN: Player 1 reached position 16!
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1
START -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 0 -> Player 2 rolls 0 -> Player 3 rolls 0 -> Player 4 rolls 1 -> Player 1 rolls 4 -> WIN: Player 1 reached position 16!
Path coverage: 91.05275472805209%
```

## Snakes and Ladders Game:

The provided code implements a **Symbolic Execution Engine** and a **Symbolic Path Tester** for a Snakes and Ladders game. This system is designed to explore the game's logic by simulating various scenarios using dice rolls, tracking player movements, and ensuring correct gameplay mechanics.

### Symbolic Execution Engine

The **SymbolicExecutionEngine** class is designed to simulate and analyze all possible paths in a Snakes and Ladders game by exploring every combination of dice rolls and player movements. It ensures efficient exploration by tracking visited states and uses recursion with backtracking to evaluate all potential game outcomes.

## Class Components

### Fields

1. **paths:**  
A list that stores all symbolic paths generated during the exploration. Each path represents a sequence of moves and outcomes in the game.
2. **visitedStates:**  
A set used to track unique game states, ensuring previously visited states are not revisited, which optimizes performance and avoids infinite loops.

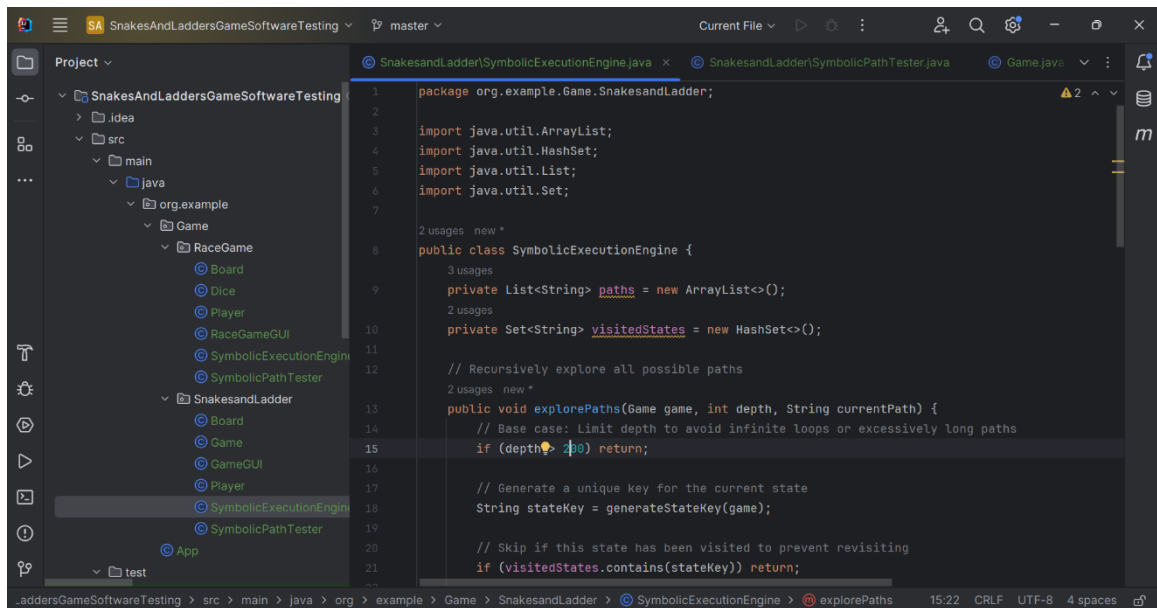
### Methods

1. **explorePaths(Game game, int depth, String currentPath):**
  - Recursively explores all possible paths in the game, starting from the given game state.
  - **Logic:** Limits recursion depth to avoid excessively long paths or infinite loops (maximum depth: 200). Generates a unique state key using `generateStateKey(Game)` to track visited states and skips previously explored states. Iterates through dice rolls (1–6) for the current player, simulating their move and updating the path. Checks for a win condition when the player's position matches the board size. Winning paths are recorded, and the player's position is reset for further exploration. Switches turns to the next player and recursively explores the subsequent moves. Backtracking is used to restore the state for other dice roll scenarios.
2. **generateStateKey(Game game):**
  - Generates a unique string representation of the game's current state by encoding each player's position and the current player's turn.
  - This key is used for efficient state tracking in `visitedStates`.
3. **printPaths():**
  - Prints all symbolic paths stored in the `paths` list.
  - Each path showcases a complete sequence of moves and outcomes, including the dice rolls, player actions, and win conditions.

### Key Features

- **Recursive Path Exploration:** Simulates all possible dice rolls and player movements.
- **State Tracking:** Ensures each unique game state is visited only once using `visitedStates`.
- **Backtracking:** Restores the game state after each recursive call to allow exploration of alternative paths.

- **Win Condition Handling:** Detects and records paths where a player wins by reaching the last square on the board.



## SymbolicPathTester

The SymbolicPathTester class serves as the main driver for the symbolic execution engine. It initializes the game and the symbolic execution engine, then triggers the exploration of game paths.

### Steps in the main Method:

#### 1. Initialize Game:

- A Game object is created with a board size of 100 (the typical size for Snakes and Ladders).
- The initializeGame() method is called to set up the game (e.g., placing the players and setting up the board with snakes and ladders).

**Example:**

```
Game game = new Game(100);
```

```
game.initializeGame();
```

#### 2. Initialize Symbolic Execution Engine:

- A SymbolicExecutionEngine object is instantiated to explore the game paths and simulate dice rolls.

#### 3. Explore Game Logic:

- The explore method of the SymbolicExecutionEngine is called to simulate all possible moves for each player. The engine explores every dice roll and updates the paths accordingly.

#### 4. Print Paths:

- The printPaths() method is called to output the paths generated during the exploration, providing a detailed account of the game scenarios.

### How the Code Works:

#### Symbolic Execution

The engine simulates the Snakes and Ladders game by iterating over all players and all possible dice rolls (1 to 6). For each combination of player and dice roll, the engine calculates the new position, checking if the player encounters a snake or ladder. If the player moves to a new position, it's recorded in the path. If the player wins (i.e., reaches the last position on the board), the win is logged. After simulating a turn, the player's position is reset to explore other potential game states.

#### Path Tracking

The symbolic execution engine tracks each turn and movement by creating a path that represents the sequence of actions taken. Each path consists of the player's name, the dice roll, and the resulting game state (e.g., a move, a win, or a message about staying at the current position due to an invalid move).

For example:

**Player=Player 1, DiceRoll=4 -> Player 1 moved to 8**

**Player=Player 2, DiceRoll=6 -> Player 2 moved to 14**

**Player=Player 1, DiceRoll=2 -> Player 1 moved to 10**

#### Handling Snakes and Ladders:

The game checks if the player's new position corresponds to the start of a snake or ladder. The getFinalPosition method adjusts the player's position if they land on a snake or ladder. For instance, if a player lands on a ladder, their position may jump forward, while landing on a snake will send them backward.

#### Resetting Player Position:

After each symbolic execution, the player's position is reset to its original state. This allows the engine to explore multiple paths without altering the board state permanently. This reset ensures that the execution of one path does not interfere with others, allowing for independent simulations of each possible scenario.

#### Output Example: (Manual Testing)

After executing the symbolic exploration, the paths might look like this:

- The code performs a comprehensive test of the game's rules, verifying that the mechanics (e.g., player movement, snake/ladder behavior, win condition) are functioning correctly under all possible game states.

#### 4. **Scalability:**

- The engine is designed to support multiple players and can be extended to handle more complex variations of the game, such as custom snakes, ladders, or additional game rules.

### **Uses and Benefits:**

#### 1. **Game Logic Verification:**

- The symbolic execution engine ensures that the game's rules are followed correctly, and all possible scenarios are tested, helping to verify that the logic is sound.

#### 2. **Bug Detection:**

- By simulating all possible paths, the engine helps to detect edge cases or logical errors in the game, such as issues with the snakes and ladders interaction, position updates, or win condition checks.

#### 3. **Efficiency:**

- The symbolic execution method provides a way to efficiently test many scenarios at once without having to manually simulate each combination of dice rolls and player turns.

### **References:**

1. "Software Testing and Analysis: Process, Principles, and Techniques"  
Authors: Mauro Pezzè, Michal Young  
Description: This book covers symbolic execution as part of advanced testing techniques. It includes discussions on its application to find bugs and ensure code correctness.
2. "Introduction to Software Testing"  
Authors: Paul Ammann, Jeff Offutt  
Description: This book includes a section on symbolic execution and its role in generating test cases and exploring program paths.
3. "Symbolic Execution for Software Testing: Three Decades Later"  
Authors: Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Willem Visser  
Journal: Communications of the ACM, 2011  
Description: A comprehensive survey of symbolic execution techniques and advancements over 30 years.
4. Symbolic Execution Explained (by MIT OpenCourseWare)  
Description: A detailed lecture explaining the principles and applications of symbolic execution in software testing.
5. "A Beginner's Guide to Symbolic Execution" (by Medium)  
Author: Flávio Cruz  
Description: A concise, beginner-friendly explanation of symbolic execution, including examples and its use in debugging and testing.