Labyrinth Game Documentation

Author: Saeed Fathallah KhanlooBrise

Neptun Code: F0HAXC

Task

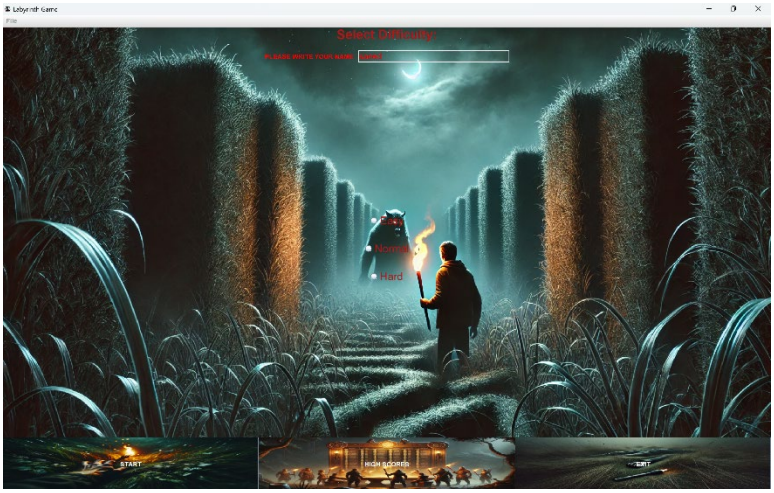
Labyrinth

The objective of the Labyrinth game is to escape the labyrinth while avoiding a dragon. The player starts in the bottom-left corner and must navigate to the top-right corner as quickly as possible. The dragon begins in a random position and moves randomly within the labyrinth.

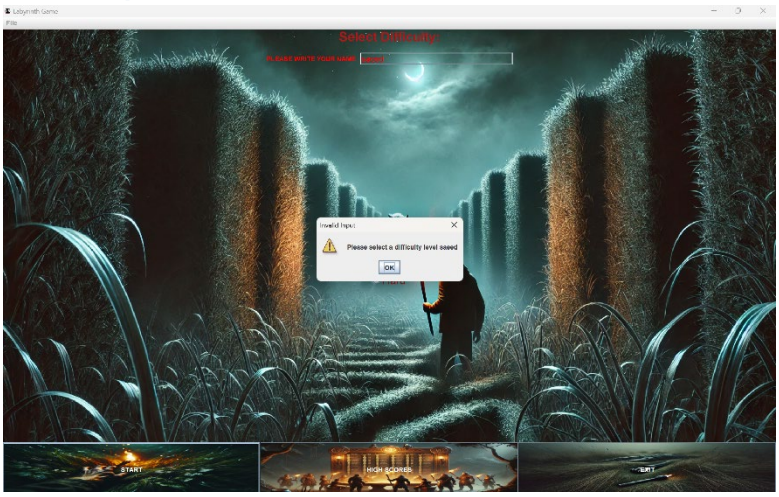
- **The player moves in four directions: left, right, up, or down.**
- **The game ends in a win if the player reaches the exit or a loss if the dragon reaches an adjacent position to the player.**
- **The player has limited visibility (radius of three fields).**
- **A high-score table records the top 10 players based on performance.**
- **Players can restart the game via a menu option.**

Statest of the Game :

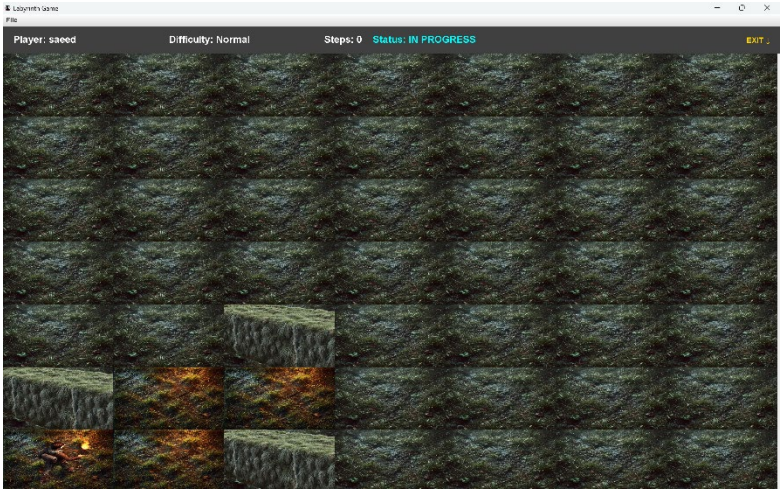
1- **MainMenu**



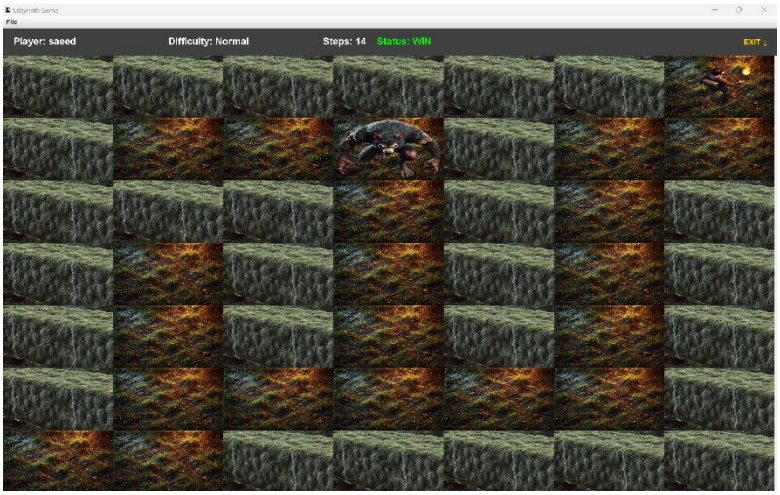
2- Invalid Input



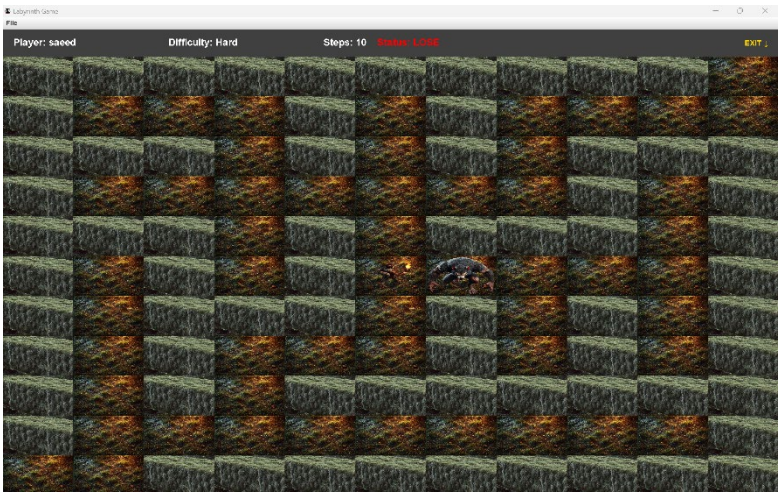
3- Beginning



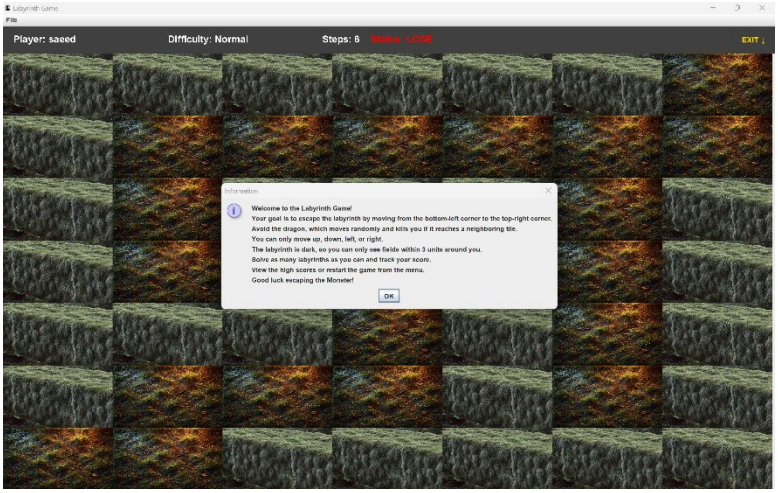
4- Win



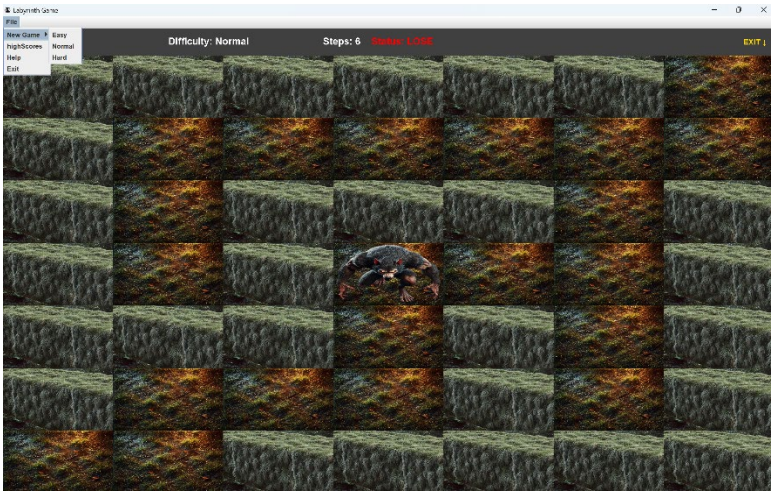
5- Lose



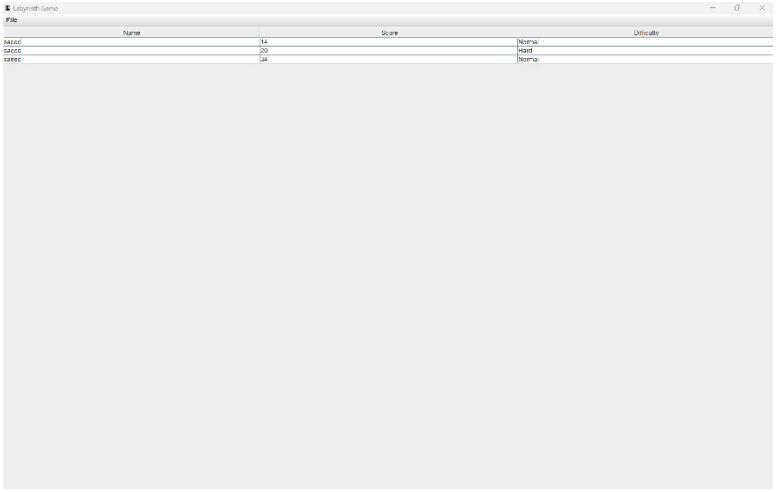
6- Help



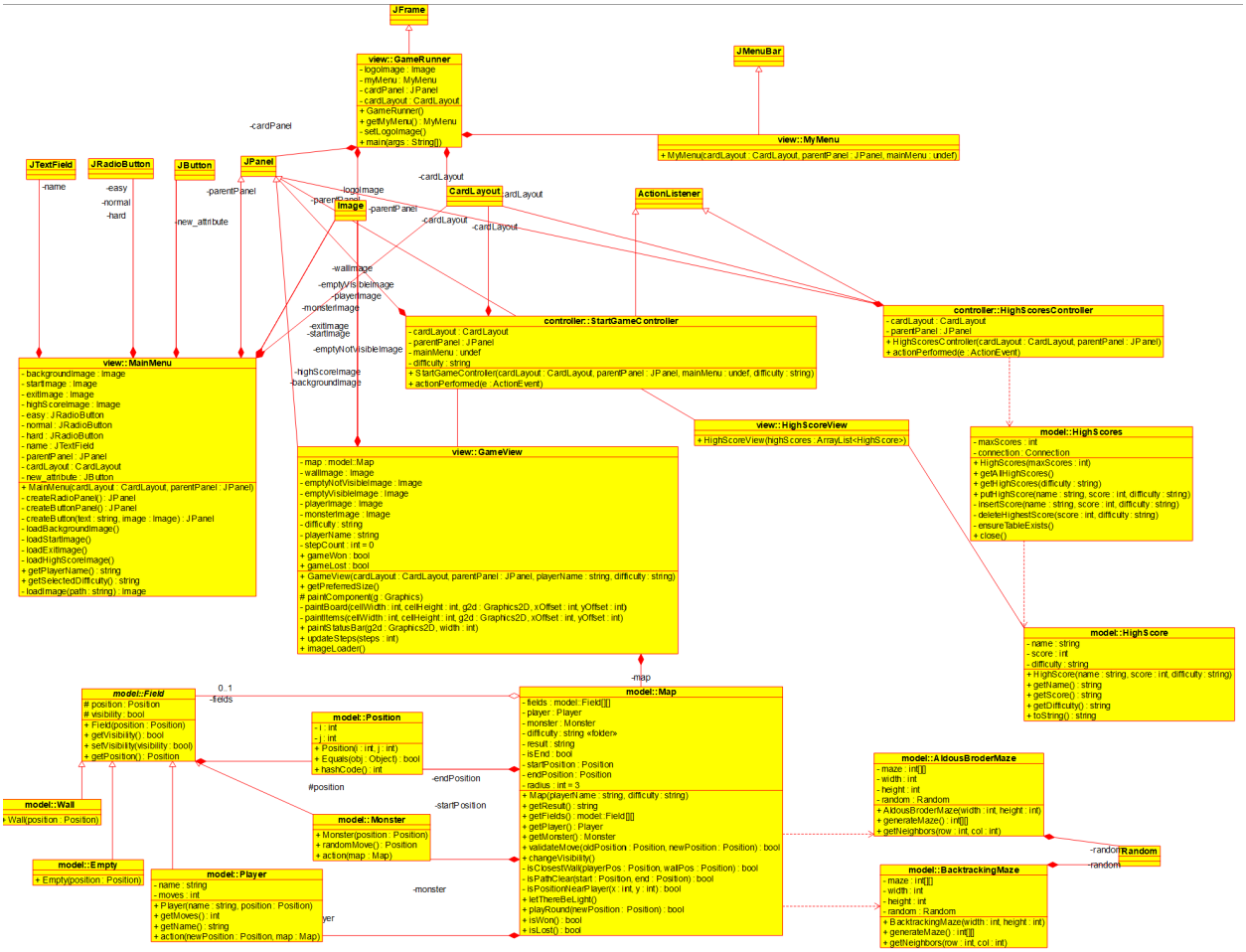
7- MenuBar



8- HighScore(LeaderBoard) :



UML :



Test Documentation for Labyrinth Game Project :

1. PositionTest.java

Purpose: To verify the correctness of the Position class, which handles coordinates in the labyrinth.
Test Case Overview:

- **ConstructorTest: Validates that positions are initialized correctly and invalid inputs are handled.**
    - **Scenario: Creating a position with negative coordinates or null values.**
    - **Expected Outcome: Constructor should throw an exception for invalid configurations.**
    - **Example: "Ensure that creating a position with -1, -1 throws an IllegalArgumentException."**
- **EqualityTest: Checks that positions with the same coordinates are equal, while those with different coordinates are not.**

---

2. PlayerTest.java

Purpose: To validate the Player class, which manages player state and movement in the labyrinth.
Test Case Overview:

- **InitializationTest: Ensures the player is initialized with correct name, moves, and position.**
    - **Example: "Verify that the player's starting position is at the bottom-left corner."**
- **MoveTest: Tests that the player moves only to valid positions.**
    - **Scenario: Move to a wall, move outside bounds, or move diagonally.**
    - **Expected Outcome: Position and moves should update only for valid moves.**
    - **Example: "Check that attempting to move diagonally leaves the player's position unchanged."**

---

3. MonsterTest.java

Purpose: To test the behavior of the Monster class, ensuring it moves randomly and validly in the labyrinth.
Test Case Overview:

- **InitializationTest: Ensures the monster is placed at a valid, non-wall position.**
    - **Example: "Verify that the monster does not start on a wall or the player's position."**
- **MoveTest: Tests that the monster moves randomly but within valid bounds.**
    - **Expected Outcome: The monster's movement should avoid walls and remain within the grid.**
    - **Example: "Ensure that after a move, the monster's new position is valid and its previous position is cleared."**

---

4. MapTest.java

Purpose: To test the Map class, which manages the labyrinth grid and interactions between the player and monster.
Test Case Overview:

- **InitializationTest: Validates that the map initializes correctly with fields, player, and monster in valid positions.**
    - **Example: "Ensure the player's position is at (4,0) and the exit is at (0,4)."**
- **ValidateMoveTest: Tests that the map correctly validates valid and invalid moves.**
    - **Scenario: Attempting diagonal movement, moving to a wall, or moving outside bounds.**
    - **Expected Outcome: Only valid moves are allowed.**
    - **Example: "Ensure moving diagonally from (4,0) to (3,1) is flagged as invalid."**
- **WinConditionTest: Verifies that the player wins upon reaching the top-right corner.**
    - **Example: "Check that the game state updates to 'win' when the player reaches the exit."**
- **LossConditionTest: Ensures the player loses if the monster reaches an adjacent field.**

---

5. HighScoresTest.java

Purpose: To validate the HighScores class, which tracks and manages the high-score database.
Test Case Overview:

- **InitializationTest: Ensures the database table is created successfully.**
    - **Example: "Check that the high-score table exists and is accessible after initialization."**

- **AddAndRetrieveTest: Tests that new high scores are added and retrieved correctly.**
    - **Scenario: Adding a score with name and difficulty, then retrieving it.**
    - **Expected Outcome: The new score should appear in the correct order.**
    - **Example: "Verify that adding a score of 50 places it at the top of the table."**
- **EnforceLimitTest: Validates that only the top scores remain when the limit is reached.**
    - **Scenario: Adding scores beyond the limit and ensuring the lowest score is removed.**
    - **Example: "Ensure adding a score of 10 removes the lowest score of 300 from the table."**
- **EmptyDatabaseTest: Checks the behavior of the system when no scores are present.**
    - **Expected Outcome: Retrieving high scores should return an empty list.**