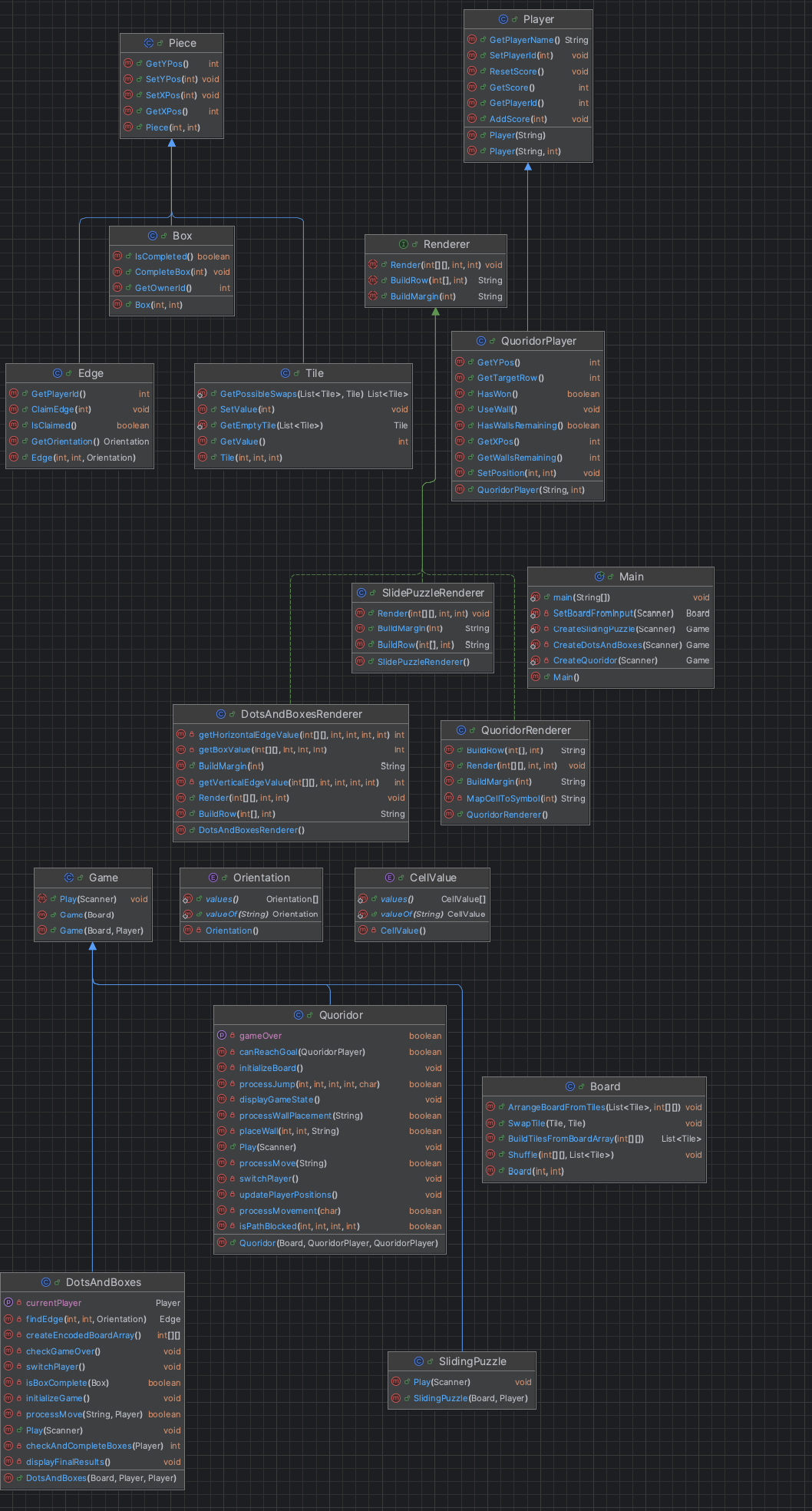
ARCADE DESIGN DOC

1. UML Diagram(after assignment 3 updated)：



1. Scalability and Extendibility

The arcade project is scalable because you can adjust the width and the height of the board to be whatever height the user enters in Slide Puzzle and Dots and Boxes. There is a minimum height and width of 2. If the user enters a height or width below the minimum height or width of 2, my program sets the height or width to a default value which is 3 for both. It also would write a friendly message to the console saying there was an invalid value for height or width and it’s setting the value to the default. In Quoridor, the height and width of the Board is always 9 by 9.

The puzzle is also extendable because many games will have a board, a player and play method. Additionally, the boards are always rendered and have rows and margins, which the Renderer interface has methods for doing. There is a SlidePuzzleRenderer, DotAndBoxesRenderer, and QuoridorRenderer which implement this interface. Additionally, I have a Piece class which has an x-position and y-position. The Tile, Edge, Box class all inherit from this. This means if I wanted to extend this project by adding more, I would simply need to add another class that inherits from Game. If I render a board, it can implement the Renderer interface. If I have anything in the game that has a specific x- or y-position, I would create another class that extends the Piece class. Additionally, I have a QuoridorPlayer class which inherits from the Player class. It has all the attributes of Player which include PlayerId and PlayerName but it also has wallsRemaining since players have a given number of walls that they can play.

Changes October 5, 2025

* Added more classes for extendibility
* Slide game now has player name and keeps track of number of moves a player makes
* Dots and Boxes game is now implemented

1. For assignment 3:
2. Overall architecture(for quoridor):

(1).QuoridorRenderer.java (game renderer)

(2).CellValue.java (Cell state enumeration)

(3).Quoridor.java (main game logic):

- Move Validation: Checks for wall obstruction and valid moves

- Jump Mechanics: Handles player interactions

- Wall Placement: Validates wall placement and prevents complete path blocking

- Path Finding: Uses a BFS algorithm to ensure game play is complete

(4).QuoridorPlayer.java (specialized player class):

- Extends `Player` to add Quoridor-specific functionality

- Manages the wall inventory (10 walls per player)

- Tracks position and target row

- Implements victory condition checks

1. Contribution (Shuxun Zhou):

- Design and implement the main game (Move players, build walls, detect victory) `Quoridor.java`

- Modify the game player class content of "QuoridorPlayer.java"

Contribution (Aidan Perez):

* Fixed bug in Main class where entering an it would cause program to crash
* Wrote code to initialize the boardArray with initial CellValue enum values in Quoridor
* Wrote code to render the Quoridor board