Design Documentation

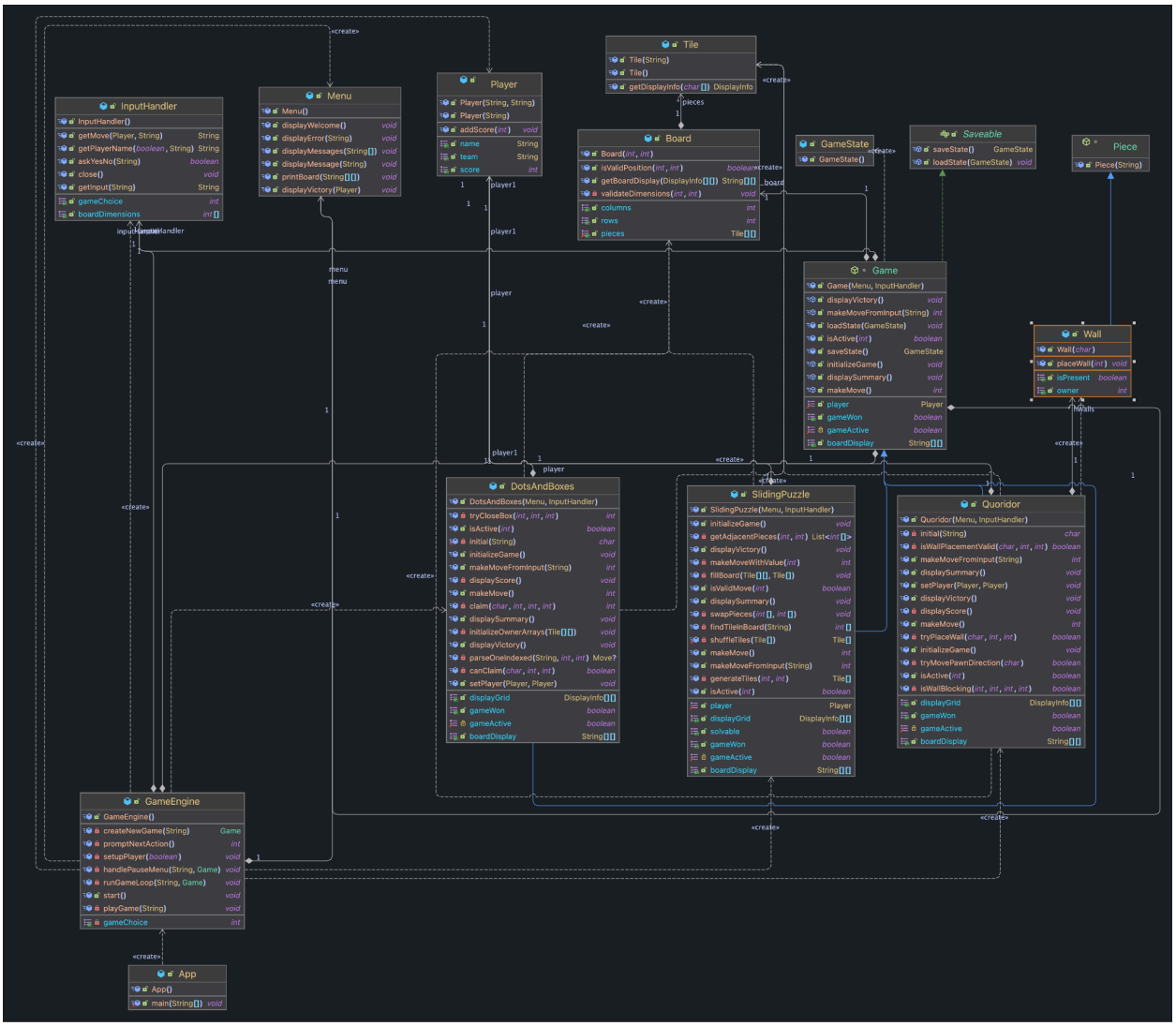
Assignment 3

Quoridor

Sanjana | U33564045

Alessandro | U63309114

UML Diagram


InputHandler
InputHandler()
getMove(Player, String) String
getPlayerName(boolean, String) String
askYesNo(String) boolean
close() void
getInput(String) String
gameChoice int
boardDimensions int[]
Menu
Menu()
displayWelcome() void
displayError(String) void
displayMessages(String[]) void
displayMessage(String) void
printBoard(String[][]) void
displayVictory(Player) void
Player
Player(String, String)
Player(String)
addScore(int) void
name String
team String
score int
Tile
Tile(String)
Tile()
getDisplayInfo(char[]) DisplayInfo
Board
Board(int, int)
isValidPosition(int, int) boolean
getBoardDisplay(DisplayInfo[][]) String[][]
validateDimensions(int, int) void
columns int
rows int
pieces Tile[][]
GameState
GameState()
Saveable
saveState() GameState
loadState(GameState) void
Piece
Piece(String)
Game
Game(Menu, InputHandler)
displayVictory() void
makeMoveFromInput(String) int
loadState(GameState) void
isActive(int) boolean
saveState() GameState
initializeGame() void
displaySummary() void
makeMove() int
player Player
gameWon boolean
gameActive boolean
boardDisplay String[][]
Wall
Wall(char)
placeWall(int) void
isPresent boolean
owner int
DotsAndBoxes
DotsAndBoxes(Menu, InputHandler)
tryCloseBox(int, int, int) int
isActive(int) boolean
initial(String) char
initializeGame() void
makeMoveFromInput(String) int
displayScore() void
makeMove() int
claim(char, int, int, int) int
displaySummary() void
initializeOwnerArrays(Tile[][]) void
displayVictory() void
parseOneIndexed(String, int, int) Move?
canClaim(char, int, int) boolean
setPlayer(Player, Player) void
displayGrid DisplayInfo[][]
gameWon boolean
gameActive boolean
boardDisplay String[][]
SlidingPuzzle
SlidingPuzzle(Menu, InputHandler)
initializeGame() void
getAdjacentPieces(int, int) List<int[]>
displayVictory() void
makeMoveWithValue(int) int
fillBoard(Tile[][], Tile[]) void
isValidMove(int) boolean
displaySummary() void
swapPieces(int[], int[]) void
findTileInBoard(String) int[]
shuffleTiles(Tile[]) Tile[]
makeMove() int
makeMoveFromInput(String) int
generateTiles(int, int) Tile[]
isActive(int) boolean
player Player
displayGrid DisplayInfo[][]
solvable boolean
gameWon boolean
gameActive boolean
boardDisplay String[][]
Quoridor
Quoridor(Menu, InputHandler)
initial(String) char
isWallPlacementValid(char, int, int) boolean
makeMoveFromInput(String) int
displaySummary() void
setPlayer(Player, Player) void
displayVictory() void
displayScore() void
makeMove() int
tryPlaceWall(char, int, int) boolean
initializeGame() void
tryMovePawnDirection(char) boolean
isActive(int) boolean
isWallBlocking(int, int, int, int) boolean
displayGrid DisplayInfo[][]
gameWon boolean
gameActive boolean
boardDisplay String[][]
GameEngine
GameEngine()
createNewGame(String) Game
promptNextAction() int
setupPlayer(boolean) void
handlePauseMenu(String, Game) void
runGameLoop(String, Game) void
start() void
playGame(String) void
gameChoice int
App
App()
main(String[]) void
«create»
inputHandler
menu
player
player1
board
pieces
walls

Responsibilities

Alessandro was responsible for:

- Refactoring the project code based on the feedback given in Assignment 2
- Starting Assignment 3
- Adding the Wall class to Quoridor

Sanjana was responsible for:

- The game logic for Assignment 3
- The addition of pause/save game state
- The UML Diagram and Readme

Architecture

The GameEngine acts as the central controller for the entire terminal arcade. It manages the overall loop of the game, player setup and the input/output flow through the InputHandler and Menu classes and most importantly transitions between different games.
Game is the abstract base class which defines the foundation for all the games. Each concrete game class extends this base class and works on it.
The concrete game implementations are as follows :

- SlidingPuzzle
- DotsAndBoxes
- Quoridor

Board provides a generic, reusable grid representation for all these games and manages a 2D array of Tile objects, where each Tile.DisplayInfo structure drives a unified ASCII-based rendering system for consistent visual output across different games.

Player stores player specific details and keeps track of scores and can be extended for multiplayer designs.
GameState and Saveable allow us to save and load game progress. Currently its saving in session only and is being handled within the GameEngine.

Scalability

The current game was made easier using the previous improvements to the scalability we had done with the last assignment. Our abstract Board, Game, Piece, and Tile class allow for games to be rendered generically while each game is responsible for its game logic and constructing the game state to be rendered by the Board class. The Board class also keeps track of generic game data that is abstracted using the now improved Tile class which stores non-game specific data for the Board to render.

Extendability

Our project's ability to incorporate new turn-based games is defined by our Abstract Game structure. The abstract Game class forms the general structure of each game, where each game class is then responsible for implementing game specific logic. The GameEngine only relies on the abstract methods defined in the abstract Game class. The Player, Board, Piece, and Tile class are all reusable components for any turn-based game that can be played on a Board.

Improvements

We refactored the Board and Tile class to be more responsible for the game state. Previously, the Board class was just responsible for outputting the board constructed by each game to the terminal. The Board class now partially keeps track of the game state, where specialized game data such as the wall or pawn position remain solely in each game class. Each game is then responsible for outputting a 2D Tile Display Information array, which is what the board class uses to render the game state to the terminal. By structuring it this way, we can use a generic Board class for all of our games while maintaining some state inside the Board class as well.

Additionally, the Tile class had previously been an abstraction for a String value. Now, the Tile class represents a single board cell that defines the standard visual information required for display (that being the edges and the center content). The games modify the state of the board cell by translating game rules into a DisplayInfo class, which the generic Board class uses to render the game state without needing any game specific logic.

Finally, we created a GameState class and Savable interface which we use to save the game state. With our new pause menu, players can pause, save their current game, quit the game, and return to the game if they've chosen to switch games or navigate to the main menu.