DESIGN DOCUMENTATION

Assignment #3

Quoridor

Object-Oriented Software Principles and Design

Group Members:

Chris Mary Benson - U56085268

Nandana Shashi - U05556171

CONTENTS

1. Current Framework:

Our board game engine consists of an infrastructure that allows for various turn-based games. Our design allows for scalability and reusability through the concepts of:

I. We start with the main Games class, which acts as an abstract base for the concrete classes of other games. Our Games class is a generic one; it uses a type parameter T, where T must be a subclass of the abstract class Player. Doing this provides an advantage of providing concrete methods in the Games class that can work on any subclass of the Player Class. It is also typesafe, preventing any incompatible Player subclass types from interacting with each other.

   Further, the Games class also has a list of players of the generic type, which allows for code reusability; methods for retrieving details, such as names from the players of a game, do not need to be written in each class. A single method in the Games class achieves this with the concept of Generics. The list of type T (T extends Player) is initialized with the required specific Player subclass as soon as an instance of the Games subclass is made. This list can then be used by the corresponding class to play out the game.

   Hence, the use of generics in this infrastructure allows us to manipulate the attributes of the subclasses of Player through common methods found in the Games class, thereby showing reusability and extendibility.

II. Inheritance allows for code reusability by having common functions that can be used by all of its child classes. For example, in our framework, we have the dot and boxes game class, Quoridor, and the sliding puzzle, which all extend from the main Games class. Methods such as initializing players, which are common among these games, are declared as an abstract method in the Games class, from which the child classes provide their own implementation. There are other abstract methods declared in this class for displaying the stats for each player in the game, etc.

   Our infrastructure consists of three abstract classes - Games, Player, and Board. These three classes provide the common attributes that are shared by all games and provide common abstract method declarations as well. Each game provides its own extension for these three classes to implement game-specific methods. This represents scalability, we can create extensions to implement logic specific to the game without having to redefine the core components that are central to all games.

For example, our board class contains common methods such as retrieving the rows and columns from the user input and setting the dimensions. Each subclass just has to call this method; there is no need for the subclass to provide its own implementation for it.

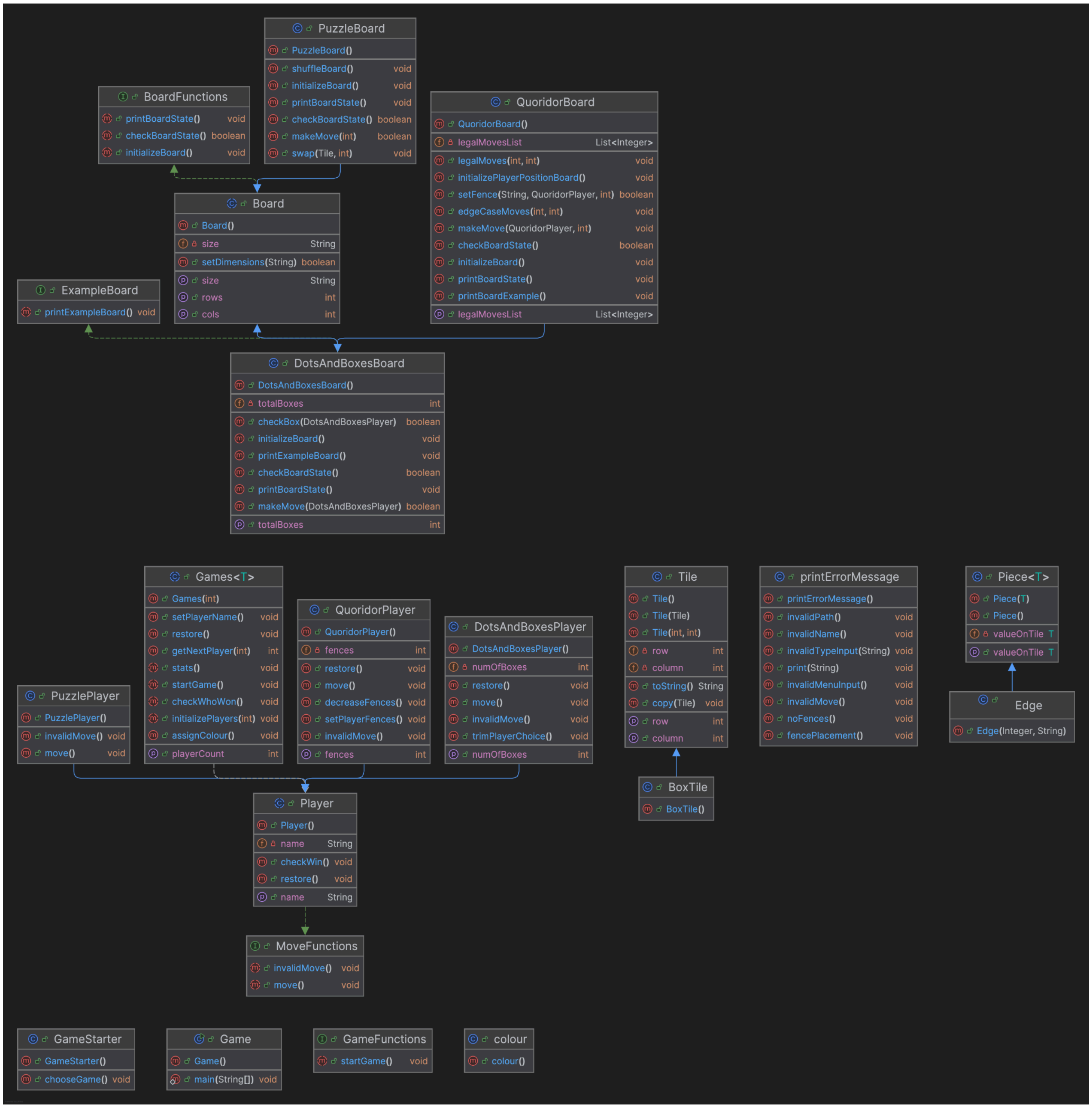
III. Interfaces enforce reusability, as there can be multiple implementations of the same interface, and at the same time, they enforce extensibility too, as there can be new implementations as well. In our design, we have interfaces that are implemented by the game, board, and player classes. For example, we have a move interface, which is implemented by the player class; now, all the child classes of player will be able to implement their own logic for the move() method.

IV. Our infrastructure also consists of utility classes that eliminate repetitive blocks of code. We have mainly three utility classes: Color class, which contains color codes, printErrorMessage class, which consists of various error statements that can be called by any class when needed, and we have the Instructions class, which is used to print the instructions for each game. All of these are declared in the Games classes, and hence can be used by any subclass when needed (Object composition).

V. With respect to turn-based variants, the Games class contains a method that can be used by any subclass to get the index of the next player in the game. If, for any reason, the player has to make another move (in the case of an invalid input or getting an extra move due to a win), the index does not get updated, and the current player makes a move in the next iteration of the game. The Games class also has an attribute called isGameDone, which is used to indicate when the iterations have to stop; this occurs whenever a player has won the game or has decided to quit the game.

Our infrastructure can be further understood from the UML diagram in the next section.

2. High-Level Overview of Infrastructure - UML Diagram



A short description of each class:

➔ Games:

Game.java : Contains the main class where an instance of the GameStarter class is initialized.
GameStarter.java : Displays the menu of games to the player to choose from.
DotsAndBoxes.java : A class specific to the DotsAndBoxes game.
PuzzleGame.java : A class specific to the Sliding Puzzle game.
Quoridor.java : A class specific to the Quoridor game.

➔ Board:

Board.java : Abstract class that serves as a base class for other boards of specific games.
DotsAndBoxesBoard.java : A Board class that consists of attributes specific to the Dots and Boxes game.
PuzzleBoard.java : A Board class that consists of attributes specific to the Sliding Puzzle.
QuoridorBoard.java : A Board class that consists of attributes specific to Quoridor.

➔ Player:

Player.java : Abstract class that serves as a base class for other players of specific games.
DotsAndBoxesPlayer.java : A player class that consists of attributes specific to the dots and boxes game, such as move(), etc.
PuzzlePlayer.java : A player class that consists of attributes specific to the sliding puzzle game, such as move(), etc.
QuoridorPlayer.java : A player class that consists of attributes specific to the Quoridor game, such as move(), etc.

➔ Tile:

Tile.java : Represents the tile on the board. Each tile will have certain characteristics - its position on the board indicated by the row and column, and also the value on the tile, which is represented by an object of the Piece class.
BoxTile.java : A class that can be used for games such as dots and boxes, which focuses on the edges of a tile. It also has an object of the Piece class, which represents the value on the tile.
Edge.java : Represents each edge of a tile on the board. It extends Piece and has an attribute edgeColour.
Piece.java : Represents a piece on a tile on the board. It is of a generic type.

➔ Utility classes:

Colour.java : A class that contains the ANSI codes for colors.
PrintErrorMessage.java : A class to print the error messages.
Instructions.Java: A class to print the instructions for each game.

3. Changes from the Previous Framework

In our first framework, before DotsAndBoxes, we only had a player and board class, with its extensions for the Sliding Puzzle game. While implementing Dots and Boxes, we decided that it would be beneficial to change our infrastructure to include a class to present the main menu to the user, from which the chosen game would be instantiated and called to start the game. We did the following:

i) Added a GameStarter Class:

The Game class serves as the main entry point of the program. It creates an instance of the GameStarter class, which is responsible for displaying the game menu and initiating the selected game.

This class handles the display of the main game menu interface. It allows players to choose which game to play, making the framework more interactive and organized by separating menu-related functionality from the main game logic.

ii) Created a Specific Game Class:

Specific game classes have been introduced for each game type. The Dots and Boxes game, Sliding Puzzle game, and Quoridor now each have their own dedicated classes that extend the central Game class. This ensures that each game can implement its unique rules and gameplay mechanics while still following a unified structure within the framework.

iii) Introduced Tile and Piece Classes:

The new framework introduces a Tile and a Piece class to represent each tile on the board and its corresponding value, respectively. Each Tile object holds its position on the board and contains a Piece object representing its value. The Piece class holds the content or symbol placed within a tile.

In the Dots and Boxes framework, the Tile class defines positions and relationships for the edges through its child class, BoxTile. BoxTile has an Edge member, which is a subclass of the Piece class. This class is also used by the Quoridor class to make use of the edge attributes.

4. Contributions:

***Chris Mary Benson*:**

- **QuoridorBoard.java** - initializeBoard(), initializaPlayerPositionBoard(), makeMove(), setFence(), edgeCaseMoves()
- **Quoridor.java** - startGame(), setFencesForPlayer(), initializePlayers(), printLegalMoves(), findValidPath(), stats(), placeFence()
- **QuoridorPlayer.java** - all methods
- **Utilities** - Color class, printErrorMessages class
- Gamestarter class, Games class

***Nandana Shashi*:**

- **QuoridorBoard.java** - printBoardState(), makeMove(), setFence(), legalMoves(), edgeCaseMoves(), printBoardExample
- **Quoridor.java** - startGame(), setCurrentPosition(), setPlayerPosition(), findValidPath(), restore(), movePosition(), checkWhoWon()
- **QuoridorPlayer.java** - all methods
- **Utilities** - printErrorMessages class, Instructions class
- GameStarter class, Games class