Deliverable # 3

Group # 13

Daniel, James, Dylan and Zsanett

*Design Documentation*

For this deliverable our goal was to have a functional game that is playable through the console, as well as packaging our code for better maintainability and readability. We have refactored much of our code that we had written from the previous deliverable for maintainability.

Since last deliverable we have added a few new implementations; An AIPlayer class, a HumanPlayer, a Game and GameLauncher class.

The purpose of the AI player is to allow the user to play a single player game. The AIPlayer will randomly select a piece that it can move, and then randomly select a move for that piece. If an opponent's piece can be taken by the AI, it will prefer to do so over a regular move. The HumanPlayer class is a Scanner input controlled player. This class, and the AIPlayer class, are child classes of the Player class, from deliverable two.

The Game class will create a game and two players of either type. The players will be chosen depending on the users’ choice upon launching the game. This class will also hold the methods for each player's turn, which will end only when the player's turn is over, or if the player loses. The losing player is determined when a player has no remaining pieces, or none that can move on their turn.

The GameLauncher class is a simple launcher for the Game. It is currently a placeholder and will most likely be replaced by a more full-featured launcher when the GUI is incorporated.

The game, as stated previously, is played through the terminal. Upon launching the game, you will be prompted if you wish to play single player or multiplayer. Once you choose, you are assigned as red; at the moment it is represented as “R” on the terminal. The human user will be asked to pick the location of a piece and move to a valid location. For location coordinates we use a (x,y) coordinate system.

All the other classes have been refactored to fit our new structure. This includes a large amount of refactoring to the Board class- its checks are now self-contained and its methods are fully encapsulated. Only the board is capable of modifying its own state, other classes must call its public methods if they wish to move a piece. It will perform all of the necessary checks to ensure that illegal moves are never made.

Class #1 – AIPlayer

By James

Design: The AIPlayer, child of Player, is a class that contains relevant data for an NPC. The AIPlayer will first attempt to find an opponent's piece to jump, but if that is not possible, it will make a random movement.

|  |
| --- |
| Game |
| - generator: Random |
| + RandomStart(): Location  + RandomEnd(): Location |

Class #2 – HumanPlayer

By James

Design: The HumanPlayer, child of Player, is a class that contains relevant data for a human opponent. The HumanPlayer has the appropriate data relevant to an opponent, such as color and move validations.

|  |
| --- |
| HumanPlayer |
| - scanner: Scanner |
| + takeInput(Boolean pieceSelection): Location |

Class #3 – Game

By James

Design: The Game class, which takes on the singleton pattern, will initiate a game. It contains methods that will set the players depending on the input of the user. Also it will initialize a board and determine turns, based on whether the user is a human or an AI.

|  |
| --- |
| Game |
| - instance: Game  + blackPlayer : HumanPlayer  + redPlayer : HumanPlayer  - board : Board  - mode : int  - gameOver : Boolean |
| + getInstance(): Game  + getMode() : Integer  + initialize() : void  + turn(HumanPlayer aPlayer) : void  + turn(AIPlayer aPlayer) : void  - canMove(Player aPlayer) : Boolean  - gameOver(Player aPlayer) : void  + gameOver() : boolean |

Class #4 – GameLauncher

By James

Design: The GameLauncher class will retrieve the single instance of Game and initialize it. This is the main method that is called to begin playing the game, and only contains a 'while' loop to continue playing until a player loses.

|  |
| --- |
| GameLauncher |
|  |
| No Methods or variables. |

Class #5 (REFACTORED) – Board

Design: The Board class underwent significant changes, for a few reasons. The first being readability (methods were divided up as needed), the second being encapsulation (the board's state is only changeable by the board itself), and the last being removing redundant code.

|  |
| --- |
| Board |
| + BOARD\_ROWS: int  + BOARD\_COLUMNS: int  - boardArray: Piece[][]  - turnComplete: int |
| + initializeBoard(): void  + checkSquare(Location square): Piece  - setSquare(Location square, Piece piece): void  + turnComplete(): int  + resetTurn(): void  + endTurn(): void  - inBounds(Location square): boolean  - isJump(Location start, Location end): boolean  + movePiece(Player player, Location start, Location end): void  + checkMove(Player player, Location start, Location end, Boolean isJump, Boolean silent): boolean  + checkMove(Player player, Location start, Location end, Boolean silent): boolean  - deltaX(Location start, Location end): int  - deltaY(Location start, Location end): int  + printArray(): void |
|  |

*To-Do*

For the next Deliverable, we have a few priorities. Our first and foremost priority is to get a basic GUI operating. Zsanett has prepared a large library of images to use for our GUI, so all that remains is the implementation of the GUI itself in Java. We will be using the Swing library to accomplish this.

Our second priority is to rework the test cases to ensure that all possibilities are checked. Because our Board class was reworked to ensure encapsulation, it is now more difficult to check movements on the board, as it is not possible to create custom arrangements of pieces. A possible solution we are considering would be to create a subclass of the Board that is a TestBoard, used only to test valid movements on the board with a fully modifiable board to work with.

Finally we will continue to rework code as needed. The Board class has been the subject of many iterations of changes, and we have a functional solution, but there may be some aesthetic and functional changes to be made before the final product. This may involve dividing the Board itself into smaller, and more manageable, classes.