**Othello—Design Document**

Taylor Spooner, Chandler Underwood, Stephen Robinson, Jordan White

October 22, 2013

This document describes the design of a computerized Othello game. The goal is to create an Othello game that follows the requirements given in “Othello—Requirements Document.”

**Overview of the class structure**

The game will be comprised of the following classes and interfaces, in additions to the classes created by Dr. Vegdahl and Dr. Nuxoll in the game framework.

* Othello game
  + OthelloGame—interface that defines an Othello game.
  + OthelloLocalGame—class that communicates with the players to check and process moves.
  + OthelloMainActivity—creates the game and initializes the players.
* Othello player
  + OthelloPlayer—interface that defines an Othello player.
  + OthelloHumanPlayer—player with a visible GUI and listeners that allows a human user to play.
  + OthelloComputerPlayer—abstract class that defines a computerized Othello player.
  + SmartOthelloComputerPlayer—player that plays with a set strategy.
  + DumbComputerPlayer—player that makes random legal moves.
* Othello game state
  + OthelloState—holds the current state of the game, includes the board, pieces, which players are playing, whose turn it is.
  + CellState—an enumeration class that stores potential states of each cell.
* Othello actions
  + OthelloMoveAction—places a piece of the player’s color on the selected cell, if it is a legal move.
  + OthelloPassAction—does not change the board, pass the turn to the other player.

The following UML class diagram illustrates the relationships between the classes, includes relationships to classes in the game framework.

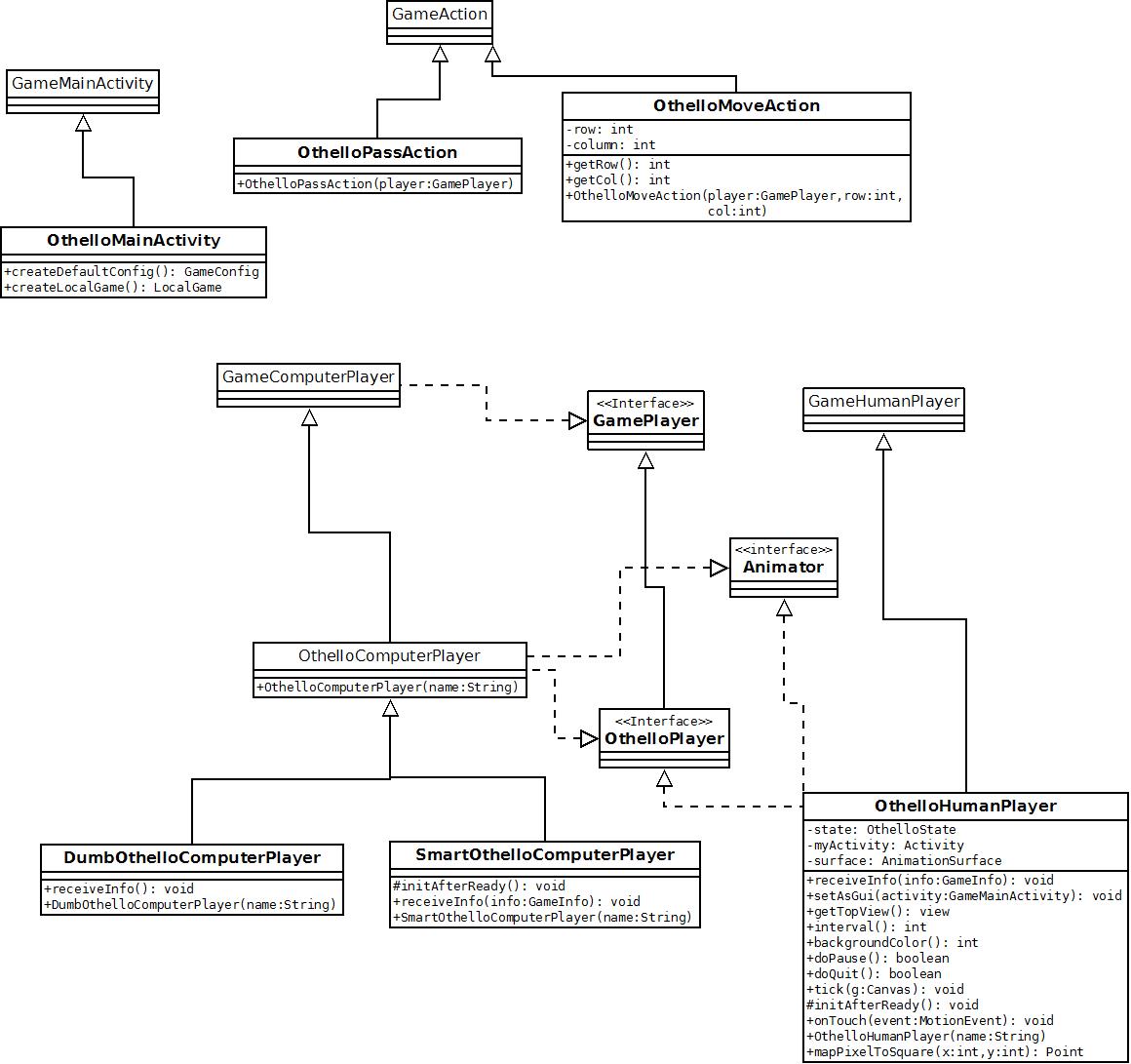
****

Figure 1: Class Diagram

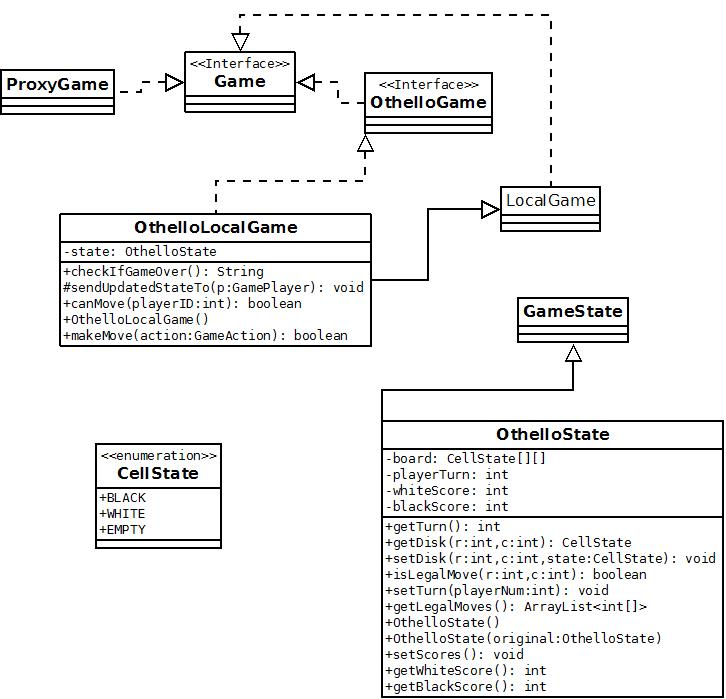
****

Figure 2: Remaining classes

**Discussion of the individual classes**

**OthelloLocalGame**

The purpose of OthelloLocalGame is to enforce the rules, prevents a player from cheating, checks if the game is over, and sends copies of the game state to players.

Within the OthelloLocalGame object, the following variables are stored:

* The current state of the game is stored. Holds all the information about the score, turn, board, pieces, and players.
  + Private OthelloState state

Within the OthelloLocalGame object, the following methods are defined:

* **OthelloLocalGame()—** Constructor for the local game, creates an instance of the state of the game.
* **String checkIfGameOver()—** If there are no available legal moves to either player , the game is over. This method returns a string with the winner’s name and the final score, which will be displayed in a dialog box.
* **void sendUpdatedStateTo(Game Player p)—**Creates a copy of the official state of the game and sends a copy to both players. Since Othello is a perfect information game, both players will receive the same copy.
* **boolean canMove(int playerID) —**Will check if it is the player’s turn using the state of the game. Returns true if there are available legal moves for the player, returns false if there are no legal moves for the player.
* **boolean makeMove(GameAction action)—**Takes in the move performed by a player. Checks if the move is a legal move or not. If it is a legal move, the method will return true, and then tell the current state of the game to update accordingly. In addition, it will call sendUpdatedStateTo(GamePlayer p). If it is an illegal move, the method will return false, the screen will flash red, and the state of the game will not be changed.

**OthelloState**

The OthelloState class has multiple purposes. The first purpose is to keep track of the official state of the game. This includes which player has the current turn, the state of each square on the board, and the score of each player. The second purpose is to send the current state of the game, in a copied form, to each player so that they are aware of the game’s current state. The player is sent a copied version of the game state so that they cannot cheat or tamper with the actual game.

Within the OthelloState class the following variables are stored:

* An 8 by 8 board in the form of an array of CellStates. This variable represents the actual gameboard that the game is played on. Each position in the board can either be “White,” “Black,” or “Empty.”
  + Private CellState[][] board
* An integer to keep track of which player has the current turn. This will either be 0 or 1 and will refer to the player’s index provided in the game framework.
  + Private int playerTurn
* Two integers that keep track of the players’ current scores. A white score for the white disc player, and a black score for the black disc player.
  + Private int blackScore
  + Private int whiteScore

The OthelloState class contains the following methods:

* Getter and setter methods for various instance variables related to the class.
  + Public int getTurn: returns the index of the player who has the current turn.
  + Public int getScore(player:int): returns the score of the player index is passed in.
  + Private void setTurn(playerNum:int): sets the current turn to whatever index is passed in.
  + Private void setScore(player:int): sets the score of the player that is passed in to the amount of discs that player has on the board. The score will be calculated within the method by iterating through the board and counting each disc of that color.
* A setter method to set a disc at a legal square on the board. This method is special because it will also need to flip any discs that get flanked when the disc is placed. The setDisc method will take two ints as parameters; these ints represent a row and a column in the board array at which the disc is to be placed. At the end of the setDisc method, setScore will also be called so that the score of the game is always updated as soon as the board updates.
  + Public void setDisc(r:int, c:int, state:CellState)
* A method to test whether or not a move would be legal at a particular square on the board.
  + Public Boolean isLegalMove(r:int, c:int)
* A method that returns an array list of all legal moves.
  + Public ArrayList getLegalMoves()
* A copy constructor method. This will be used to generate copies of the game state that will be sent to the players. Because this is a perfect information game, no information will be omitted.
  + Public OthelloState(original:OthelloState): Creates a copy of the OthelloState that is passed in.

**CellState**

CellState is an enumeration that will be used to describe the current state of each square on the board. It has three different values, BLACK, WHITE, and EMPTY. It has no methods or instance variables, its sole purpose is to make the code more readable.

**OthelloMoveAction and OthelloPassAction**

The purpose of the two action classes is to allow players to interact with the local game and through the local game, the game state. If the player presses a valid square on the GUI board, an OthelloMoveAction is created and sent to the local game. If the player has no legal moves available and presses the pass button, an OthelloPassAction is created and sent to the local game. The AI players will be able to generate and send these actions to the local game as well.

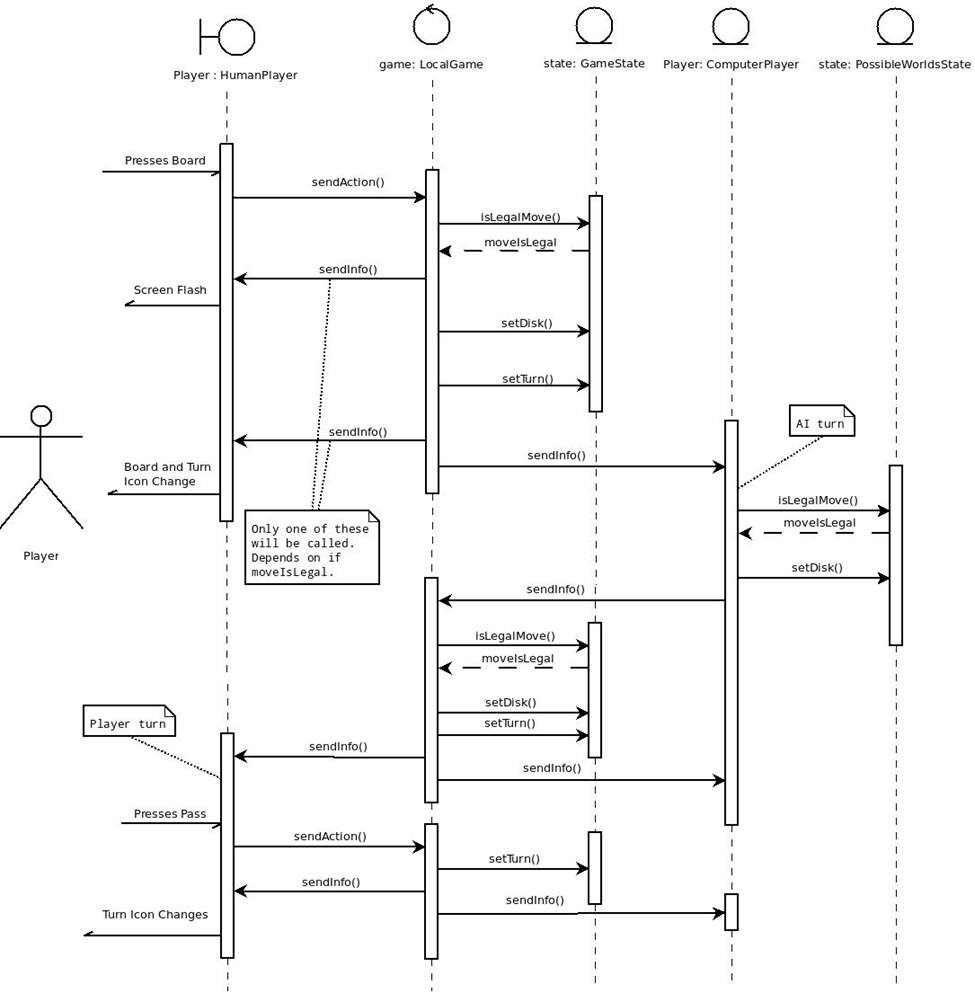
* The OthelloPassAction class has no instance variables and simply tells the game to switch the current turn to the other player.
* The OthelloMoveAction class has a row and a column int variable to represent the row and column on the game board at which the player is attempting to place a disc. It also has getter methods for the row and column so that the local game can use the action and test it against the game rules for validity.

Figure 3: Sequence Diagram

**OthelloHumanPlayer**

The purpose of the OthelloHumanPlayer is to allow a player to play Othello using a GUI.

The OthelloHumanPlayer class has the following instance variables:

* A copy of the most recent state of the game:
  + Private OthelloState state
* An instance of the current Activity so that the OthelloHumanPlayer is able to set up the GUI and listeners needed to play the game:
  + Private Activity myActivity
* An AnimationSurface on which the OthelloHumanPlayer is able to draw the board and any pieces:
  + Private AnimationSurface surface

The OthelloHumanPlayer has the following methods available:

* The following methods that must be implemented so that the user can interact with a GUI and play the game of othello:
  + Public void receiveInfo(info:GameInfo): receives messages from local game.
  + Public void setAsGui(activity:GameMainActivity): this method is responsible for setting up any GUI related components and listeners that the human player needs to play the game.
  + Public view getTopView(): returns the top level view from the xml layout. This allows the game to flash the screen if the player makes an invalid move.
  + Public int interval(): returns an integer that tells the animator how many times per second it should be re-drawing the canvas.
  + Public int backGroundColor(): returns an int that represents the background color the animator will be drawing. Our background will be green.
  + Public void Tick(g:Canvas): this method is where all of the animating/drawing will occur.
  + Protected void initAfterReady(): When the game first starts, this method is called to initialize certain aspects of the game for the human player.
  + Public Boolean onTouch(event:MotionEvent): Method is used to receive and handle any user touches on the game board.
  + Public void onClick(v:View): Responds to any clicks on the hint, pass, or quite buttons on the GUI.
  + Public Point mapPixelToSquare(x:int, y:int): Returns a point with the x-coordinate referring to a row and the y-coordinate referring to a column on the game board. This method is used to convert the pixel location of a user’s touch into a row and a column that the local game will understand.

**OthelloComputerPlayer:**

The purposes of the OthelloComputerPlayer classes are to provide a computer based player for another user to play Othello against. There will be two variations of OthelloComputerPlayer; a DumbOthelloComputerPlayer, and a SmartOthelloComputerPlayer.

DumbOthelloComputerPlayer, as the name implies, will be a simple AI that makes random legal moves. The DumbOthelloComputerPlayer will have the following method:

* receiveInfo(): The DumbOthelloComputerPlayer will use this method to receive messages from the local game, and then make its random legal move based on the game state it receives. Before making each move, the AI will “sleep” for one second to simulate thought and to create a realistic game pace.

The class will also have the following instance variable:

* Private OthelloState state: the class will use this OthelloState to check the board for legal moves that it can make.

SmartOthelloComputerPlayer is the difficult/intelligent AI player. This AI will operate by examining game tree of possible worlds. To do this it will create a copy of the internal game state for each possible world, and maintains a record of the moves to get to that state for analysis. The AI will look down each branch and quantify each possible path based on: Points gained/lost, corners and sides obtained, and estimated probability of occurrence. The smart AI will also include a delay to create a realistic game pace.

The following methods will be inside the SmartOthelloComputerPlayer:

* Protected initAfterReady(): performs any initialization that needs to be done at the start of each game.
* Public receiveInfo(info:GameInfo): used to receive messages from the local game concerning the current game state.

The SmartOthelloComputerPlayer will have the following instance variables:

* Private OthelloState state: this state will be examined for lists of possible moves by the AI.

**Walk-throughs of common operations**

There are three basic operations that a player may perform in this game:

* Placing a disc on the board
* Asking for help with the hint button
* Passing when there are no legal moves

To demonstrate these four scenarios we will assume a mock game between a human player (player 0) and a computer player (player 1). We will discuss the following scenarios:

* Human player makes a move.
* Computer player makes a move.
* Human player passes.
* Human player requests a hint.

Human player makes a move:

We will start this scenario when it is the human player’s turn. We will assume that there is at least one legal move available to the human player.

* The game sends a copy state of the game to both players.
  + The computer player will ignore this message because it is not their turn.
  + The human player will receive the info and update its GUI accordingly.
  + Because there is at least one legal move available, the pass button will be disabled.
* The user will examine the board, select a move to make, and press the GUI on the square where they wish to make a move.
  + The human player will take the pixel location of the tap, and use the mapPixelToSquare method to convert that into a row and a column on the game-board.
  + An OthelloMoveAction will be created at that row and column, and then sent to the local game.
* The local game object will process this move to make sure it is the players turn, and then check the master game state to see if this is a valid move.
  + Once it has been determined that the move was legal, the master game state will be updated and place a disc at that location.
* The updated state will now be sent back to all players.
  + The human player GUI will update with the new discs on the board.
  + The human player GUI will also update the score and turn text boxes.
* It is now player 1’s turn.

Computer player makes a move:

This scenario will start when it is the computer player’s turn and there is at least one legal move available.

* The computer player begins its turn by receiving an updated game state from the local game object.
  + If it is a dumb computer player, the computer player will simply use the getLegalMoves methods to get a list of legal moves, and then select one random move from that list. It will take that move and create an OthelloMoveAction to send to the local game object.
  + If it is a smart computer player, the computer player will evaluate the game state using an algorithm that analyzes the game tree of possible worlds. Once it has determined the move that will be most beneficial in the long-run, it will create an OthelloMoveAction and send it to the local game object.
* The local game object will receive the move from the computer player.
  + The local game object checks to make sure that it is the computer player’s turn, and then checks the master game state to see if the move is legal.
  + Once the move has been determined legal, the local game will tell the master game state to place a disc at that location.
* Next, the local game will create copies of the updated master game state and send them back to the players.
  + The computer player will ignore this message because it is no longer its turn.
  + The human player will now be free to make a move.

Human player passes

This scenario starts out when it is the human player’s turn, but there are no legal moves available.

* The human player GUI will update itself with the new game state.
* Because there are no legal moves available, the GUI will enable the pass button.
* The user will examine the board, and upon realizing that no legal moves are available, they will press the pass button.
  + This creates an OthelloPassAction and sends it to the local game.
* The local game will check to determine that this is a legal move.
  + First, it will determine that it is the human player’s turn
  + Next, it will check the master game state to confirm that there are no legal moves available for the human player.
* After determining that the pass move was legal, the master game state will be updated so it is no longer the human player’s turn.
* New copies of the updated game state will be sent to both players, and the computer player will now be free to make a move.

Human player requests a hint:

We start this scenario when it is a human player’s turn, and the user is unsure of where they are able to make a legal move.

* The user presses the hint button on the GUI.
* Any squares on the game-board that are legal moves, will be highlighted. The GUI will be re-drawn so that the user can see the legal moves.
* The user is now able to select a legal move and the game continues as described above.

**Testing Plan**

In order to implement this game correctly, we will need to test all classes as we develop them. This will require us to write “unit tests” in JUnit for all major classes. We will also test the full game with human users by actually running the game and playing it on a tablet.

The basic list of things we will test for each class is as follows:

* Test the constructors by creating objects and making sure they were constructed properly.
* Test all of the methods for each class by using the methods on a constructed object and examining the state.

We will test in the following order:

* OthelloState, OthelloLocalGame, and basic OthelloHumanPlayer.
* OthelloMoveAction, OthelloPassAction, and DumbOthelloComputerPlayer.
* SmartOthelloComputerPlayer and finished OthelloHumanPlayer.

**Division of Labor**

* **11/05:** Finish design document. Update with feedback from lab.
* **11/08:** The GUI will be created by Taylor and Chandler. The game state class will be done by Jordan. Early version of the human player will be created by Chandler and Taylor. This human player will be able to state the game and see the GUI but cannot interact. (12 hours per person)
* **11/12:** All action classes and local game will be implemented by Jordan. The easy AI will be completed by Stephen. The human player will be completed by Chandler and Taylor.(8 hours person)
* **11/16:** The smart AI will be completed by Stephen. Game should now be playable. (8 hours per person).
* **11/24:** All bugs found in the Test-to-Succeed will be addressed and fixed. (Hours depending on where the bugs were found.)
* **12/5:** All bugs found in the Test-to-Fail will be addressed and fixed. (Hours depending on where the bugs were found.)
* **12/6:** Game completed and turned in.