**SFWR ENG 2AA4**

**Assignment 2**

**Group 11**

**Names:**

**Mario Machado**

**Saim Zahid**

**Khalil Rahin**

**Arka Ganguli**

**Revision History:**

* We decided to re-design the model we originally had. This is because the model we had earlier would not allow us to expand further without running into issues as we were trying to communicate between 3 different “boards”. This new design focuses on one Board and then the BoardController and CheckerPiece class work within the Board.
* The CheckerPiece class handles setting a piece, checking whether it is a king or not, and assigns the two teams.
* We renamed BoardController to Board and it handles the mode (if we are placing pieces or if we are in a game), keeps track of each player’s turn and handles loading and saving a game.
* The Board class will also handle events such as user input and pass it into a method called doAction, which then passes it to the controller and checks if it is a valid move. This method will be used even further for the AI we will need to implement in part 3 as we can have cases for user input and no-input (i.e. computer’s turn).
* We added a new BoardController class, which handles setting the board, moving the pieces, checking the legal moves available including, jump checking and valid moves.
* We renamed CheckerBoard to MainMenu as it essentially contains all the code for the main menu and initializes the game.
* We are deleting BoardButton and moving the action listeners from that class to Board, which handles user input now.

**Decomposition of Classes:**

The decomposition of the classes are still similar to last time, except we have 5 classes now and we have re-named a few classes, which were explained above. We have reduced coupling and made these classes more cohesive from the previous design choice. The parent classes are Board, BoardController, and CheckerPiece. The BoardController class handles the logic for checking if a move is legal and if a jump/ move can be made and setting the board. The Board handles events, loading, saving and modes. The CheckerPiece handles the checker piece jbutton and the king of a checker. The MainMenu and BoardView class aid the parent classes. MainMenu creates the menu, and BoardView creates the 8x8 checkerboard.

**Modular Interface Specification (MIS):**

In our MIS, we discuss the interface of each class (public entities) and the function of the methods regarding what the user sees in the interface. The uses relationship is also discussed here.

**Module: Board**

This class is used to save and load game sessions, it also decide when to start the game and the initial conditions of the game.

**Uses:**

CheckerPiece

BoardController

**Access Programs:**

**doAction(int x, int y): int**

Keeps track of what the players are clicking and moves the pieces accordingly. Also if the players choose custom mode, this places the pieces of first player on the board(the player decides where and how many pieces they want), and when player 1’s team is full lets player 2 places their pieces onto the board.

**actionPerformed(ActionEvent e)**

Gets the x-y coordinates of the checker piece clicked, and on the second click, if the move is legal, moves that piece to the desired location.

**save(int game): int**

Gives the  
players the option to save the progress of their current game.

**load(intgame): int**

Gives the players the option to load a previously  
saved game.

**Module: BoardView**

This class will be the graphical user interface for the checkers board. It will contain buttons on all legal spaces, and display all the checker pieces currently on the board.

**Uses:**

JFrame

**Access Programs:**

*makeBoard*

Creates an empty board with no checker pieces.

*displayMesg*

Display the given string to the user.

*addButtons*

Add the buttons to the board, and add the given input to each button when added.

**Module MainMenu**

This class creates the Menu to let users select a custom or classic mode checkers game.

**Uses:**

JFrame

BoardView

Board

**Access Programs:**

*MainMenu*

Displays the menu

*actionPerformed*

A method within the constructor that places the checkers pieces in a certain pattern (custom or classic) depending on what the user chose in the menu.

**Module: CheckerPiece**

This class is responsible for what changing the look of the board when certain conditions are met or a player makes a move. For example because our board is made up of a collection of organized images, when a player moves a piece it replaces the original tile with a blank and sets the other tile to be the image of the checker piece. It also changes the image if a piece has been kinged or a piece has to be removed from the board.

**Uses:**

**Access Programs:**

**JButton getButton()**

Gets the tile that’s has been clicked.

**setPiece(int piece): int**

This method is responsible for deciding whether a piece is black or white, if a tile is just blank, if the picee is a king, and whether it’s a white or black king.

**Module: BoardController**

This class is responsible for controlling the pieces on the board. It calculates which moves are legal and which aren’t.

**Uses:**

CheckerPiece

**Access Programs:**

**CheckerPiece[][] getBoard()**

Returns the array of the checkers pieces on the board.

**addTeam(int team, int x, int y): int**

Adds up the number of pieces of black and white checkers as they are added to the board.

**Modular Internal Design (MID):**

The MID includes a description of the implementation (private entities) including class variables

**Module: Board**

This class is used to save and load game sessions, it also decide when to start the game and the initial conditions of the game.

**Variables:**

int: turn, definition: Global

Keeps track of whose turn it is.

int: lastx, definition: Global

Keeps track of the x-coordinate of the last mouse click.

int: x, definition: Global

Stores the x coordinates of the second click, after the lastx lasty click.

int: y, definition: Global

Stores the y coordinates of the second click, after the lastx lasty click.

int: lasty, definition: Global

Keeps track of the y-coordinate of the last mouse click.

boolean: play, definition: Global

Decides when to start the game.

int: game, definition: Global

Stores the value of the game that has to saved/loaded. For example the first game to be saved will be game1 the second game to be saved will be game2, and so on. The same goes for loading.

**Uses:**

java.awt.event.ActionEvent

java.awt.event.ActionListener

java.io.

CheckerPiece

BoardController

**Access Programs:**

**doAction(int x, int y): int**

Keeps track of what the players are clicking and moves the pieces accordingly. Also if the players choose custom mode, this places the pieces of first player on the board(the player decides where and how many pieces they want), and when player 1’s team is full lets player 2 places their pieces onto the board.

if play == true

if no previous command/click

lastx = x;

lasty = y;

if given lastx lasty and x y

make the move on the board if legal

reset lastx lasty and x y

switch players

else

Give the first player the option to set his pieces on the board.

Switch team and give player 2 the option to set his pieces on the board, after players 1 is done.

Set play == true.

**actionPerformed(ActionEvent e)**

Gets the x-y coordinates of the checker piece clicked, and on the second click, if the move is legal, moves that piece to the desired location.

CheckerPiece temp = The coordinates of the first mouse click on the desired Checker Piece.

doAction: Moves the selected piece to desired location.

**save(int game): int**

Gives the players the option to save the progress of their current game.

**load(int game): int**

Gives the players the option to load a previously saved game.

**Module: CheckerPiece**

This class is responsible for what changing the look of the board when certain conditions are met or a player makes a move. For example because our board is made up of a collection of organized images, when a player moves a piece it replaces the original tile with a blank and sets the other tile to be the image of the checker piece. It also changes the image if a piece has been kinged or a piece has to be removed from the board.

**Uses:**

java.awt.event.ActionListener

javax.swing.

JButton

**Variables:**

int team, definition: Global

Keep track of which team the pieces belong to. 1- Black, 0- White

int x, definition: Global

Gets assigned to the position of x.

int y, definition: Global

Gets assigned to the position of y.

Boolean king, definition: Global

If true that piece is kinged; If false then the piece is a regular checker piece.

int posx, definition: Global

Is the x position of a checker piece.

int posy, definition: Global

Is the y position of a checker piece.

int color, definition: Global

The value of this variable decides what color the checker pieces is.

int piece, definition: Global

Gets assigned a specific checker piece. This is mostly for the purpose of whether to king a piece or keep it as a regular piece.

**Access Programs:**

**JButton getButton()**

Gets the tile that’s has been clicked.

**setPiece(int piece): int**

This method is responsible for deciding whether a piece is black or white, if a tile is just blank, if the picee is a king, and whether it’s a white or black king.

case 0:

Blank tiles

case 1:

White Pieces, not king.

case 2:

Black Pieces, not king.

case 3:

White Piece, King.

case 4:

Black Piece, King.

default:

Blank Tiles.

**int getPiece():**

This method is responsible for returning the credentials of a tile as integers and Booleans.

If piece is a king

If its Black team

Return Case 3

If its White Team

Return case 4

Else

Return Case 0

If piece is not a king

If its Black team

Return Case 1

If its White Team

Return Case 2

Else

Return Case 0

**Boolean isKing():**

returns the variable king.

**int getTeam():**

Returns the variable team.

**String toString():**

Returns the string value of variable team.

**int getX():**

Returns the variable x.

**int getY():**

Returns the variable y.

**Module: BoardController**

This class is responsible for controlling the pieces on the board. It calculates which moves are legal and which aren’t.

**Uses:**

java.awt.event.ActionListener

javax.swing.

CheckerPiece

**Variables:**

int whiteCheckers, definition: Global

Variable for all the white pieces on the board, in other words how many white pieces are on the board.

int blackCheckers, definition: Global

Variable for all the black pieces on the board, in other words how many black pieces are on the board.

boolean newGame, definition: Global.

This variable is set to true when a player starts a new game.

int team, definition: Global

Keeps track of which team the pieces belong to. 1- Black, 0- White.

int x, definition: Global

Gets assigned to the position of x.

int y, definition: Global

Gets assigned to the position of y.

int oldx, definition: Global

Last position of the x-coordinate. Used for deciding whether move is legal or not.

int oldy, definition: Global

Last position of the y-coordinate. Used for deciding whether move is legal or not.

int newx, definition: Global

New position of the x-coordinate. Used for deciding whether move is legal or not.

int newy, definition: Global

New position of the x-coordinate. Used for deciding whether move is legal or not.

int color, definition: Global

The value of this variable decides what color the checker piece is.

int difx, definition: Global

The difference between newx and oldx

int dify, definition: Global

The difference between newy and oldy

**Access Programs:**

**CheckerPiece[][] getBoard()**

Returns the array of the checkers pieces on the board.

**addTeam(int team, int x, int y): int**

Adds up the number of pieces of black and white checkers as they are added to the board.

Tiles[X position][Y position].setPiece(Team black or White?);

Number of White Pieces + 1 = new Number of White Pieces

Number of Black Pieces + 1 = new Number of Black Pieces

**boolean teamFull(int team): int**

This method is responsible for deciding when a team is full, in other words has a total of 12 pieces.

if white team return whiteCheckers = 12

else if black team return balckCheckers = 12

return True

**doMove(int oldx,int oldy,int newx,int newy): int**

This method checks if a move is legally allowed.

difx = newx - oldx;

dify = newy - oldy;

Set new coordinates of tile using newx and newy

Set old piece to blank

If a piece is jumped then set it to blank as well.

**canMove(int team, int oldx,int oldy,int newx,int newy):**

This method decides if a move is illegal.

**JButton[][] getButtons():**

Gets x-y coordinates of a button.

**setBoard(int[][] iBoard)**

Sets the board up depending on the positon and number of white and black pieces .

**int numberOfCheckers(int colour)**

Returns the number of checkers on the board, depending on the color of the piece.

Case 0: returns whiteCheckers

Case 1: returns blackCheckers

Default: returns whiteCheckers + blackCheckers.

**public int numberOfBlack():**

return blackCheckers

**public int numberOfWhite()**

return whiteCheckers

**public int totalNumberOfCheckers()**

return blackCheckers + whiteCheckers

**printModel()**

Prints ASCCII model of the board

**Module: BoardView**

This is the actual ‘board’ for our game. Its is the window which holds all the pieces and buttons.

**Uses:**

java.awt.

javax.swing.

JFrame

**Variables:**

int columns definition Global.

The variable that stores the number of columns on the board. It is set to 8 by default.

int rows definition Global.

The variable that stores the number of rows on the board. It is also set to 8 by default.

**Access Programs:**

**makeBoard():**

This method sets up the board. The size is set to 800x800 by default and it also separates the board into 8 square by 8 squares for easy access to buttons and to set up a coordinate system.

Set Title to “Checkers”

Set Size to 800x800

for(i, starting at 0 going to 7 by increments of 1)

for(j, starting at 0 going to 7 by increments of 1)

Panel Ith, Jth = new Jpanel

Add panel to the board

**displayMesg(String mesg):**

Displays a message.

**addButtons(JButton[][] button):**

Adds button to every panel using a for loop.

**Module MainMenu**

This class creates the Menu to let users select a custom or classic mode checkers game.

**Variables:**

BoardView view - definition: global

**Access Programs:**

**MainMenu()**

A constructor that initializes the menu.

Utilizes the window builder in Java eclipse.

**actionPerformed(event)**

A method within the constructor that places the checkers pieces in a certain pattern

(custom or classic) depending on what the user chose in the menu.

Pseudocode:

Make a new BoardView object view, with parameters of 8 rows, 8 columns, white, red

Make a new BoardModel object model, with boolean true as a parameter

Make a new BoardController object controller, and pass in view and model as a parameter.

**Traceability**:

The traceability of our program is much improved from Assignment 1. The classes interact with each other in a way that supports the MVC framework. The methods within the classes interact within the same class and coupling is also reduced.

**Evaluation of Adequacy of Design:**

Pros:

* Our design is simple and allows users to get started quickly by selecting a custom (load or place pieces) or classic mode game.
* We replaced the buttons with color as name to actual jbuttons that represent the checkers image, to improve GUI experience
* The way our board pieces are set up, reduces illegal moves as in custom mode you can only place pieces on the darker boxes.

Cons:

* Redundant variables. For instance: the panels and the arrays essentially contain the same information. But to make it robust, we had to sacrifice efficiency
* We don't show the available legal moves to a player at any time. This would be a good improvement as it allows players to see available moves and make the right choice.

**Testing:**

Open custom mode: Expected output is a blank checkers board with the ability to place all peices wherever you want. The tested result is as expected.

Place white pieces in custom mode: Tested is as expected we put 12 pieces anywhere we want and then tells us its black turns to place pieces

Place black pieces in custom mode: Expected output: able to place 12 pieces anywhere on the board then begins game. Tested result: as expected (correct)

Move a white piece or black piece: Expected result: Able to select a white piece when it is white’s turn or a black piece when it is black’s turn and do a legal move otherwise it will give illegal move error. Tested result: As expected, correct.

Move a piece over opponent’s piece: Expected result: Able to jump an opponent. Tested result: Correct, white can jump black or black can jump white.

Save game: Expected result: can save a game and then load it back. Tested result: Correct. Can save game then load it.

King piece: Expected result: when a piece gets to the end it becomes king. Tested result: correct as expected, it becomes king.

Open Classic Mode: Expected result: open a classic mode board. Tested result: a classic board is opened as expected and able to play a game.

White piece selected when its black’s turn: Expected result: Error. Tested result: as expected.

Black piece selected when it’s white turn: Expected result: Error. Tested result: as expected.

Backing up: In this test scenario, a player is trying to move a piece that is not kinged backward. The expected output is an error thrown by the program as it is not a legal move. Tested result: As expected an error.

Number of pieces: In custom mode, a player is only allowed to place 12 pieces for a colour. The expected output is an error message if you try to place more than 12 pieces of the same colour.

Backing up a kinged piece: Expected result: a played can move a kinged piece backward now. Tested Result: As expected one can move a king backwards.

Winner: Expected result: game ends when all pieces of opponents are done and winner is claimed. Tested result: Correct as expected.