

2AA4 Connect 4 : Assignment 3

Submitted by: The Group

Osarumwense Abraham Omorogbe – 1310163

Akinjisola Akinkugbe – 1130797

Tamunotonye Fiberesima – 1231043

Genevieve Okon -1315579

Alexander Jackson - 1302526

Justin Kapinski - 1305257

2AA4 Connect 4: Assignment 3

Software Engineering 2AA4, McMaster University

Submitted on April 3rd 2015

Table Contents

Table of Figures 2

Table of Tables 2

Requirements 3

MIS (4.1 & 4.2) 4

Uses Relationship (4.3) 14

Traceability (4.3) 15

MID (4.4) 16

Evaluation of the adequacy of the design (4.5) 35

Specific section on the algorithm (4.6) 36

Document the code (4.7) 36

Testing (5) 37

# Table of Figures

* Figure 1: UML uses relations 14

# Table of Tables

* Table 1: Summary of Traceability (Look at MIS for more details on class) 15
* Table 2: Task Metrics 35
* Table 3: Test Results 37

# Requirements

1. This final assignment keeps everything you had in assignment 2
2. Being able to choose between two modes of operation: 2 player Connect Four, in which 2 people can play against each other; or 1 player against the computer.
3. In 2 player mode, the behaviour is the same as in assignment 2.
4. For playing against the computer:
5. All the “rules” of the 2 player game should be the same for the game played against the computer.
6. Specifically, the same mechanism should be used to decide who plays first, once it has been decided whether the computer plays blue or red.
7. The user shall be able to choose to start a new game, store an existing unfinished game, and re-start a stored game.
8. You will need an algorithm to determine the best next move for the computer. For this assignment, this does not have to be the optimum move, just try and make it give you a reasonably good move. It obviously has to be a legal move.

**From now on we will refer the requirements as its corresponding number listed above.**

# MIS (4.1 & 4.2)

\*\*blue indicates changes added

\*\* red indicates items removed

**ConnectFourAI Class Reference**

* ConnectFourAI()
* int getBestMove()

**BoardModel Class Reference**

* BoardModel()
* int getGridWidth()
* int getGridHeight()
* void setGridPiece()
* PlayerColor getGridPiece()
* Position[] getErrorPositions()
* PlayerColor getWinner()
* Position[] getWinningPieces()
* int getPieceCount()
* PlayerColor whoHasMorePieces()
* PlayerColor getErrorColor()
* PlayerColor getStartPlayer()
* boolean dropPiece()
* boolean doMove()
* boolean doTemporaryMove()
* void undoTemporaryMove()
* void saveToFile()
* void loadFromFile()
* void reset()
* BoardModel copy()

**BoardController Class Reference**

* BoardController()
* void actionPerformed()
* void update()

**BoardView Class Reference**

* public BoardView()
* BoardComponentType lookupComponentType()
* Position lookupButtonPosition()
* void drawModel()
* void highlightPiece()
* void setCurrentMenu()
* void setTitleLabel()
* void setStatusLabel()

**GameStateModel Class Reference**

* GameStateModel()
* void setState()
* GameState getState()
* void setEditColor()
* PlayerColor getEditColor()
* void setCurrentPlayer()
* void getCurrentPlayer
* void nextTurn()
* void setAIPlayer()
* PlayerColor getAIPlayer()
* void saveToFile()
* void loadFromFile()

**Position class Reference**

* Position(int x, int y)
* int hashcode()
* boolean equals()

**BoardComponentType Enum Reference**

**BoardPanels Enum Reference**

**GameState Enum Reference**

**PlayerColor Enum Reference**

* PlayerColor opponent()

CLASS CONNECTFOURAI

Tries to decide what the best move is for a given player. Satisfies requirement 1,4d.

ConnectFourAI()

Creates a new AI object

INPUT: BoardModel board: the starting state of the board to consider.

OUTPUT: None

getBestMove()

Gets the best move.

INPUT: PlayerColor player: blue or red’s move

OUPUT: int bestMove: the column a piece should be dropped into

**CLASS BOARDMODEL**

Enables user to start the game. Also manages all data related to the game board.

Satisfies requirements  1, 3, 4a.

Uses:

java.io.DataInputStream

java.io.DataOutputStream

java.io.FileInputStream

java.io.FIleOutputStream

java.io.IOException

java.util.Observable

**ACCESS PROGRAMS**

BoardModel()

* Initializes an empty game board.
* INPUT : None.
* OUPUT : None.

getGridWidth()

* Returns the width of the board.
* INPUT : None.
* OUTPUT : int Width.

getGridHeight()

* Returns the height of the grid.
* INPUT : None.
* OUTPUT : int Height.

setGridPiece()

* Set a particular position of the board to a given color.
* INPUT : Position Position : Position on the board, PlayerColor color : color to set the position.
* OUTPUT : None.

getGridPiece()

* Finds the color on a given position and returns the color of that given position.
* INPUT : Position position : Position on the board to check color.
* OUPUT : PlayerColor : color of the position.

getErrorPosition()

* Finds all possible error positions and returns an array of error positions or null if there isn’t any.
* INPUT : None.
* OUTPUT : Position[] errorPos: errors in a 1D array.

getWinner()

* Determines which player has a winning connect four if any and returns the color of the first winner it finds(player with a connect four) or none if there’s no winner.
* INPUT: None.
* OUTPUT: PlayerColor winner: color of the winner.

getWinningPieces()

* Determines the locations of the pieces of the winning connect four, if there is one.
* INPUT: None.
* OUTPUT: Position[] winPieces: an array of the four winning pieces locations or an          empty array if all elements are null ie there’s no winner.

getPieceCount()

* Counts the number of pieces of a given player.
* INPUT : PlayerColor color : color of player to count.
* OUTPUT : int count : number of pieces a player has.

whoHasMorePieces()

* Determines which player has more game pieces on the board and returns the color of that player’s pieces.
* INPUT : None.
* OUTPUT : PlayerColor : either red, blue or none.

getErrorColor()

* Determines which player has more pieces by at least 2 and returns the color of that player.
* INPUT : None.
* OUTPUT : PlayerColor : red, blue or none.

getStartPlayer()

* Determines which color and Player to start by the pieces already on the board.
* INPUT : None
* OUTPUT : PlayerColor : red, blue or NONE(if theres an error).

dropPiece()

* Drops a piece from the top of the board and lets it fall to the lowest available position.
* INPUT: int column: the column to drop the piece into, Player color: the color of the piece to drop.
* OUTPUT: boolean success: returns true if the piece fits and false if the column is full.

doMove()

* Drops a piece from the top of the board and lets it fall to the lowest available position. Observer gets notified of a new state.
* INPUT: int column: the column to drop the piece into, PlayerColor color: the color of the piece to drop.
* OUTPUT: boolean success: Returns true if the piece fits and false if the column is full.

doTemporaryMove()

* Drops a piece from the top of the board and lets it fall to the lowest available position. Observer does not get notified of a change in state.
* INPUT: int column: the column to drop the piece into, PlayerColor color: the color of the piece to drop.
* OUTPUT: boolean success: Returns true if the piece fits and false if the column is full.

undoTemporaryMove()

* Undoes a previous temporary move by removing a piece. Does not notify the observer of change.
* INPUT: int column: the column of the move to undo.
* OUTPUT: None

saveToFile()

* Throws an IOException (Input and Output interruptions).
* Captures the current state of this object and stores it in a file for later retrieval.
* INPUT: String fileName: the file to store the state in.
* OUTPUT: None.

loadFromFIle()

* Throws an IOException (Input and Output interruptions)
* Reloads the state of this object from a previously saved saved state stored in the specified file.
* INPUT: String fileName: file to load the state in.
* OUTPUT: None

reset()

* Resets the state of this object to the default state.
* INPUT: None
* UPDATE: returns to the default state where there are no colours in the grid.

**CLASS BOARDCONTROLLER**

Implements ActionListener and Observer. It is responsible for updating the models (game data) by listening to the user input events that create views(what the user sees) and vice versa.

Satisfies the requirement 1, 3, 4a.

Uses the following classes:

java.awt.event.ActionEvent

java.awt.event.ActionListener

java.util.HashMap

java.util.Observable

java.util.Observer

BoardController()

* Initializes the board controller which will listen for updates from its associated view and model after it is created.
* INPUT : None.
* OUTPUT : None.

actionPerformed()

* Is called when an event occurs and updates the right model(game data) when a user input event occurs.
* INPUT : Action e : An action event.
* UPDATE : updates the state of the game( Edit state, Start State),updates color selected (Red, Blue or None) and checks if an action is valid.

update()

* Lets the controller know that the model(game data) has been updated by a class.
* INPUT : Observable arg1 : an Observable object that called this method.

             Object arg1 : any parameter the observable object passed.

**CLASS BOARDVIEW**

Responsible for the GUI of the game. Sets up the frame and other display details of the game.

Satisfies the requirement 1, 3, 4a.

Uses:

java.awt.BasicStroke

java.awt.CardLayout

java.awt.Color

java.awt.Font

java.awt.Graphics2D

java.awt.GridBagConstraints

java.awt.GridBagLayout

java.awt.GridLayout

java.awt.Image

java.awt.Insets

java.awt.RenderingHints

java.awt.image.BufferedImage

java.util.HashMap

javax.swing.BorderFactory

javax.swing.Icon

javax.swing.ImageIcon

javax.swing.JButton

javax.swing.JFrame

javax.swing.JLabel

javax.swing.JPanel

javax.swing.SwingUtilities

BoardView()

* Initializes the view (what the user sees).
* INPUT : int boardWidth : width of the board pieces.

                        int boardHeight : height of the board pieces.

                        BoardController controller : the controller to send user input updates.

* Output : None.

lookupComponentType()

* Determines which GUI component an object is and if it is a component.
* INPUT : Object obj : an unknown object that is a GUI component.
* OUTPUT : BoardComponentType.INVALID\_COMPONENT : If it is invalid

                            The type of the component : If it is a valid GUI component .

lookupButtonPosition()

* Determines the x, y position of a button on the board.
* INPUT : Object obj : a button object.
* OUTPUT : Position : the position on the board the button occupies or null if it doesn’t.

drawModel()

* Displays the color pieces of the board to the screen.
* INPUT : BoardModel model : the model to draw.
* OUTPUT : None.

highlightPiece()

* Highlights the button with a yellow border at a certain spot.
* INPUT : Position position : the position of the button to highlight.
* OUTPUT : None.

setCurrentMenu()

* Displays the menu specified.
* INPUT : BoardPanels panelID : the id of the panel to display.
* OUTPUT : None.

setTitleLabel()

* Sets the title label.
* INPUT : String text : the text of the label.
* OUTPUT : None.

setStatusLabel()

* Sets the status label.
* INPUT: String text: the new text of the label.
* OUTPUT: None

**CLASS GAMESTATEMODEL**

Represents the current state of the game. Extends Observable.

Satisfies the requirement 1, 3, 4a, 4b, 4c.

Uses:

java.io.DataInputStream

java.io.DataOutputStream

java.io.FileInputStream

java.io.FileOutputStream

java.io.IOException

java.util.Observable

GameStateModel()

* Initializes the game state.
* INPUT : None.
* OUTPUT : None.

setState()

* Sets a new state and notifies the observers.
* INPUT : GameState state : state to change to.
* OUTPUT : None.

getState()

* Gets the current state.
* INPUT : None.
* OUTPUT : state of the game.

setEditColor()

* Updates the current edit color for the edit mode.
* INPUT : PlayerColor Color : the color to set the current edit color to.
* OUTPUT : NONE.

getEditColor()

* Gets the current edit color for the edit mode.
* INPUT : None.
* OUTPUT : editColor : the current edit color.

setCurrentPlayer()

* Sets the current player of the game.
* INPUT : PlayerColor player : the new current player.
* OUTPUT : None.

getCurrentPlayer()

* Gets whose turn it is.
* INPUT : None.
* OUTPUT : currentPlayer : the current player.

nextTurn()

* Alternates players. Goes to the other player’s turn.
* INPUT : None.
* OUTPUT : None.

setAIPlayer()

* Set which player is the AI player.
* INPUT: PlayerColor player: player
* OUPUT: None

getAIPlayer()

* Gets an AI player if there’s any.
* INPUT: None
* OUTPUT: PlayerColor aiPlayer: Returns the AI player or NONE if there isn’t any.

saveToFile()

* Throws an IOException (Input/Output interruptions).
* Captures the current state of this object and stores it in a file for retrieval later.
* INPUT : String fileName : the file to store the state in.
* OUTPUT : NONE

loadFromFile()

* Throws an IOException (Input/Output interruptions).
* Reloads the state of this project from a previously saved state stored in the specified file.
* INPUT: String fileName : the file to load the state in.
* OUTPUT : None.

**CLASS POSITION**

A simple record class used for convenience to store x, y pairs together. Used by other classes. Satisfies requirements 1, 3, 4a.

Position()

* Record for  x and y.
* INPUT : int x, int y.
* OUTPUT : None.

hashCode():

Returns the hash code value of the object. Overridden.

* INPUT: None
* OUTPUT: int result:

equals():

Indicates whether an object is equal to this object. Overridden.

* INPUT: Object obj: object being compared.
* OUTPUT: boolean true or false: returns true if equals and returns false otherwise.

**ENUM BOARDCOMPONENTTYPE**

To identify components. Satisfies requirements 1, 2, 3, 4a, 4c.

* INVALID\_COMPONENT : invalid component.
* BOARD\_BUTTON : all the buttons on the board.
* NEW\_GAME\_BUTTON : the new game button.
* NEW\_GAME\_2P\_BUTTON: the new game 2 player button.
* NEW\_GAME\_AI\_BUTTON: the new game AI button.
* EDIT\_BUTTON : the edit button.
* RED\_BUTTON : the red button.
* BLUE\_BUTTON: the blue button.
* NONE\_BUTTON : the none button.
* DONE\_BUTTON : the done button.
* LOAD\_BUTTON: the load button.
* SAVE\_BUTTON: the save button.
* MAIN\_MENU\_BUTTON: the main menu button.

Updates from Assignment 1:

* Added Load\_Button, Save\_button and Main\_Menu\_Button.

**ENUM BOARDPANELS**

To identify panels of the menu to be displayed. Satisfies requirements 1, 3, 4a.

* EMPTY: to empty .
* START : to start.
* EDIT : to edit.
* PLAY : to play.
* TO\_MAIN\_MENU: to main menu.

**ENUM GAMESTATE**

To identify the different states of the game. Satisfies requirements 1, 3, 4a.

* START\_STATE : game in the start state.
* EDIT\_STATE : game in the edit state.
* PLAY\_STATE : game in the play state.
* WIN\_STATE : game in the win state.
* DRAW\_STATE : game in the draw state.

**ENUM PLAYERCOLOR**

Represents a player color. Satisfies requirements 1, 3, 4a.

* NONE - signals an error or a situation that makes no sense or an empty spot.
* BLUE - blue player color.
* RED - red player color.

opponent():

Get the opposite player

* INPUT: None
* OUTPUT: PlayerColor Blue or Red: Returns red if blue and blue if red.

# Uses Relationship (4.3)

Please refer to the USES MVC.pdf in the documents folder for a better picture.

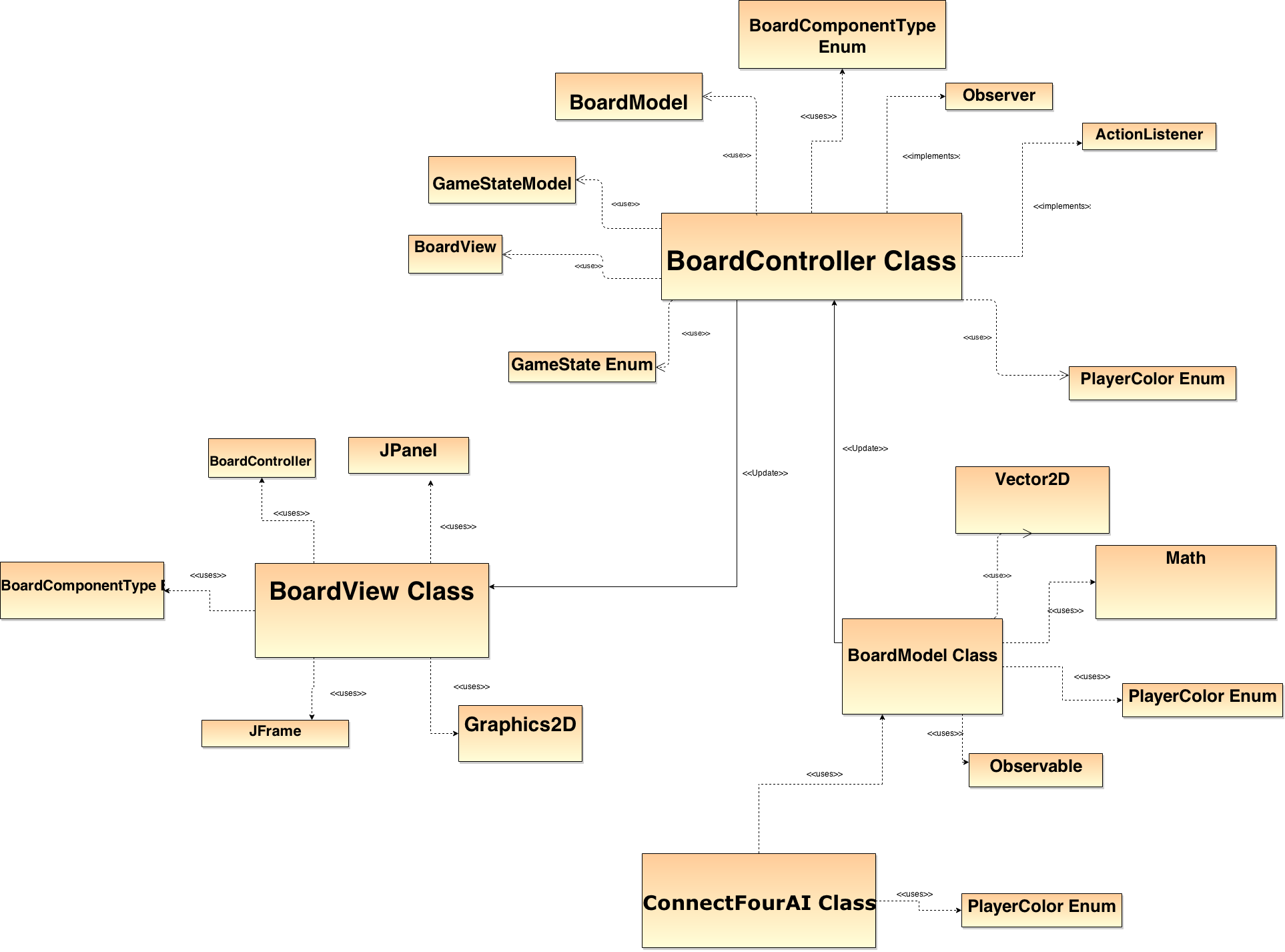


Figure : UML uses relations

# 

# Traceability (4.3)

\*\*blue indicates what was added/changed.

Table : Summary of Traceability (Look at MIS for more details on class)

|  |  |  |
| --- | --- | --- |
| Requirement | Class Used | How it achieved task |
| 1 | All | Finished assignment 2, And used all the code from the assignment. |
| 2 | BoardView  GameStateModel  PlayerColor | Added new buttons in the BoardView class to allow the user select, AI or 2 player. And new methods were added to GameStateModel (setAIPlayer(),getAIPlayer) for AI. |
| 3 | All | Added new methods and classes to assignment 2, the rules remain the same, but methods and classes were added for Artificial Intelligence. Added classes and methods such as ConnectFourAI(),negamax(). |
| 4a | BoardModel | Used the getWinner() method which determines which player has a winning connect four if any and returns the color of the first winner it finds (player with a connect four) or none if there’s no winner. These are the same rule as 2 player. |
| 4b | BoardModel  BoardController  PlayerColor | The BoardModel has a method getStartPlayer(), that the BoardController uses. Then the PlayerColor class sets the color. Just like Assignment 2. |
| 4c | BoardComponentType  BoardModel  GameStateModel | When the user presses save, it uses the saveToFile() with throws an IOException (Input and Output interruptions).  Captures the current state of this object and stores it in a file for later retrieval. When the user presses load  reloads the state of this object from a previously saved saved state stored in the specified file. Just like Assignment 2. |
| 4d | ConnectFourAI | This class used the negamax algorithm to figure out what the next move should be. |

# MID (4.4)

**Position class Reference**

* Position (int x, int y)
* int hashCode()
* Boolean equals(Object obj)

**BoardModel class Reference**

* BoardModel ()
* int getGridWidth ()
* int getGridHeight ()
* void setGridPiece (Position position, PlayerColor color)
* PlayerColor getGridPiece (Position position)
* Position [] getErrorPositions ()
* PlayerColor getWinner ()
* int getPieceCount (PlayerColor color)
* PlayerColor whoHasMorePieces ()
* PlayerColor getErrorColor ()
* PlayerColor getStartPlayer ()
* boolean dropPiece()
* boolean doMove(int column, PlayerColor color)
* boolean doTemporaryMove(int column, PlayerColor color)
* void undoTemporaryMove(int column)
* void saveToFile()
* void loadFromFile()
* void reset()
* BoardModel copy

**BoardController class Reference**

* Class BoardController ()
* Class setUpPanelMap ()
* Void actionPerformed (ActionEvent e)
* Void update (Observable arg0, Object arg1)

**ConnectFourAI class Reference**

* Class ConnectFourAI(BoardModel board)
* int getBestMove(PlayerColor player)
* int negamax(int depth, int move, PlayerColor player)
* int evaluateBoard(PlayerColor player)
* int getConnectFoursValue(int startX, int maxX, int maxY, int dx, int dy, PlayerColor player)

**GameStateModel class Reference**

* GameStateModel
* void setState
* gamestate getState
* void setEditColor
* playercolor getEditColor
* void setCurrentPlayer
* playercolor getCurrentPlayer
* void saveToFile()
* void loadFromFile()
* void nextTurn
* void setAIPlayer
* PlayerColor getAIplayer

**BoardView class Reference**

* BoardView
* void setUpFrame()
* Image createHolderImage()
* Image createPieceImage().
* BoardComponentType lookupComponentType()
* Position lookupButtonPosition()
* void drawModel()
* void highlightPiece()
* void setCurrentMenu()
* void setTitleLabel()
* void setStatusLabel()

**BoardComponentType enum Reference**

**GameState enum Reference**

**BoardPanels enum Reference**

**PlayerColor enum reference**

* PlayerColor opponent()

MODULAR INTERNAL DESIGN

CLASS DOCUMENTATION

Position class Reference

Position (int x, int y)

**IN**: int x, int y

**UPDATES**: NONE

**OUT**: NONE

Position class stores x, y pairs together. This is only used as a record. This will be used to implement future methods.

HashCode()

**IN:** NONE

**UPDATES:** NONE

**OUT:** int result

This method is automatically generated by Eclipse

equals(Object obj)

**IN:** Object obj

**UPDATES:** NONE

**OUT:** boolean true/false

This method is automatically generated by Eclipse

GameState enum Reference

GameState ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

GameState enum shows the five possible states that GameState can be in. The three possible states are START\_STATE, EDIT\_STATE, PLAY\_STATE, WIN\_STATE, and DRAW\_STATE. This enum will help implement future methods.

BoardPanels enum Reference

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

Enum BoardPanels identifies panels of menu that will be displayed and this will be according to the GameState. These include EMPTY, START, EDIT, PLAY, TO\_MAIN\_MENU.

BoardModel class Reference

**USES:** java.util.Observable

java.io.DataInputStream

java.io.DataOutputStream

java.io.FileInputStream

java.io.FileOutputStream

java.io.IOException

Class Board Model has the data and methods regarding the game board displayed and the pieces on it.

BoardModel ()

**IN:** NONE

**UPDATE:** NONE

**OUT:** NONE

Class BoardModel initializes an empty game board. All the parts of the board are made to have the color none. It uses reset().

getGridWidth ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** int value of GRIDWIDTH

GetGridWidth () returns GRID\_WIDTH, which is he width of the board. The width of the board will be found in terms of spaces between game pieces.

getGridHeight ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** int value of GRIDHEIGHT

GetGridHeight () returns GRID\_HEIGHT, which is he height of the board. The height of the board will be found in terms of spaces between game pieces.

setGridPiece (Position position, PlayerColor color)

**IN:** Position position, PlayerColor color

**UPDATE:** pieceGrid [position.x][position.y] = color

**OUT:** NONE

SetGridPiece sets the given position on the board to the color that is given.

That is pieceGrid [position.x][position.y] = color

GetGridPiece (Position position)

**IN:** Position position

**UPDATE:** NONE

**OUT: returns** pieceGrid [position.x][position.y]

GetGridPiece finds out what color piece, if any, is at the given spot on the board.

getErrorPositions ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** return an array of error positions or null if there are none

GetErrorPositions () finds all error positions (i.e. game pieces that are floating in mid air). It returns the array of the error positions or it returns null if there are no error positions.

First the numbers of error positions are found

If (pieceGrid [x][y]! = PlayerColor.NONE && pieceGrid [x][y+1] == PlayerColor.NONE)

errorAmt++;

If (errorAmt == 0)

Return null

|  |  |
| --- | --- |
| ErrorAmt ==0 | Return null |
| ErrorAmt! =0 | Put all error positions in array to be returned |

getWinner ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** returns winner or NONE

GetWinner () determine which player if any has a winning connect four. It returns winner if there is a winner and NONE if there are no winners.

There are four possible types of connect four wins; horizontal, vertical, top-left diagonal, top-right diagonal connect fours. This method checks for all four possible wins using for loops and returns winner for one of the possible wins if gotten.

The are four branches in the method that correspond to the four types of connect four wins. The difference between the branches are the for loops.

The table below will explain it better.

|  |  |  |
| --- | --- | --- |
| WINS |  |  |
| HORIZONTAL | Check the leftmost spots of all possible horizontal connect fours | **for** (**int** x = 0; x < ***GRID\_WIDTH***-3; x++)  **for** (**int** y = 0; y < ***GRID\_HEIGHT***; y++) |
| VERTICAL | Check the top spots of all possible vertical connect fours | **for** (**int** x = 0; x < ***GRID\_WIDTH***; x++)  **for** (**int** y = 0; y < ***GRID\_HEIGHT***-3; y++) |
| TOP-LEFT | Check the top-left spots of all possible "\"-diagonal connect fours | **for** (**int** x = 0; x < ***GRID\_WIDTH***-3; x++)  **for** (**int** y = 0; y < ***GRID\_HEIGHT***-3; y++) |
| TOP-RIGHT | Check the top-right of all possible "/"-diagonal connect fours | **for** (**int** x = 3; x < ***GRID\_WIDTH***; x++)  **for** (**int** y = 0; y < ***GRID\_HEIGHT***-3; y++) |

After the for loops the code continues. An if statement is used to check if a red or blue piece is in that spot. It checks if other three spots are the same color. If a connect four was found it returns the color of the winner else it stops checking that spot.

getPieceCount (PlayerColor color)

**IN:** PlayerColor color

**UPDATES:** NONE

**OUT:** returns count

GetPieceCount counts number of pieces for a given player of the input color and returns number of pieces for that player.

whoHasMorePieces ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** returns PlayerColor.NONE/RED/BLUE according to which satisfies if statement.

WhoHasMorePieces determines which player has the most pieces on board (that is which color has more pieces). If they have the same it returns none.

|  |  |
| --- | --- |
| If | Returns |
| RED>BLUE | RED |
| BLUE>RED | BLUE |
| BLUE=RED | NONE |

getErrorColor ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** returns PlayerColor.None or whoHasMorePieces.

GetErrorColor returns which color has more pieces than the other by at least 2, and returns none if the difference is less than 2.

getStartPlayer ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** returns NONE/RED/BLUE

GetStartPlayer determines which player should start or returns NONE if there is error regarding number of pieces of each color. This is determined on which pieces are already on the board. It uses the whoHasMorePieces method to determine how many pieces of each color are on the board and different cases to determine whom to play.

|  |  |
| --- | --- |
| Cases | Playertostart |
| RED | BLUE |
| BLUE | RED |
| NONE | Randomly return either red or blue |

dropPiece()

**IN:** Int column, PlayerColor color

**UPDATES:** NONE

**OUT:** returns success

Drops a piece from top of the board and lets it fall to the bottom and places piece in the first empty space it finds. This works by trying every spot starting from the bottom to the top and places the piece in the first empty space found.

doMove()

**IN:** int column, PlayerColor color

**UPDATES:** NONE

**OUT:** returns success

This drops a piece from the top of the board and lets it fall to the lowest available position. It returns true if there is room and notifies observers of new state

doTemporaryMove(int column, PlayerColor color)

**IN:** int column, PlayerColor color

**UPDATES:** NONE

**OUT:** return dropPiece(column, color)

This drops a piece from the top of the board and lets it fall to the lowest available position. It returns true if there was room but does not notify observers.

undoTemporaryMove()

**IN:** int column

**UPDATES:** NONE

**OUT:** NONE

This undoes a previous temporary move by removing a piece and does not notify observers of change.

saveToFile()

**IN:** String fileName

**UPDATES:** NONE

**OUT:** None

It captures the current state of this object and stores it in a file for it to be retrieved later.

for (int x = 0; x < GRID\_WIDTH; x++)

for (int y = 0; y < GRID\_HEIGHT; y++)

outStream.writeByte(pieceGrid[x][y].ordinal())

Since there are less than 255 colors for a player to be storing it in a byte is safe.

loadFromFile()

**IN:** String fileName

**UPDATES:** pieceGrid[x][y] = PlayerColor.values()[inStream.readByte()]

**OUT:** NONE

It reloads the state of this object from a previously saved state, which is stored in a specific file.

for (int x = 0; x < GRID\_WIDTH; x++)

for (int y = 0; y < GRID\_HEIGHT; y++)

pieceGrid[x][y] = PlayerColor.values()[inStream.readByte()]

It loads the stored bytes back and converts them back to enums.

setChanged() and notifyObservers().

This lets the observers know the state has change since after loading the state could be very different.

reset()

**IN:** NONE

**UPDATES:** pieceGrid[x][y] = PlayerColor.NONE

**OUT:** NONE

This resets the state of the object to a default state.

Then it uses setChanged() and notifyObservers() to let the observers know the state has change since after loading the state could be very different*.*

copy()

**IN:** NONE

**UPDATES:** newBoard.pieceGrid[x][y] = pieceGrid[x][y]

**OUT:** return newBoard

This gets a copy of this board models state but without updating the observers.

BoardController class Reference

Uses: Java.awt.event.ActionEvent

java.util.Observable

java.util.Observer

javax.swing.SwingWorker

java.awt.event.ActionListener

java.io.IOException

java.util.HashMap

This class gets user input events that view creates and reacts to them by updating models. It also listens to models and updates view when needed.

BoardController ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

BoardController Initializes the board controller and it's associated and models. The controller will listen for updates from the view and model after it is created.

setUpPanelMap ()

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

SetUpPanelMap initializes panel map. It identifies which panel of the menu should be associated with which game state.

(GameState.***START\_STATE***, BoardPanels.***START***)

(GameState.***EDIT\_STATE***, BoardPanels.***EDIT***)

(GameState.***PLAY\_STATE***, BoardPanels.***PLAY***);

(GameState.***WIN\_STATE***, BoardPanels.***TO\_MAIN\_MENU***)

(GameState.***DRAW\_STATE***, BoardPanels.***TO\_MAIN\_MENU***)

actionPerformed (ActionEvent e)

|  |  |
| --- | --- |
| CASE |  |
| BOARD\_BUTTON | If a button on the game board was pressed in the edit state that position will be the current edit color |
| NEW\_GAME\_2P\_BUTTON | PlayerColor.NONE  PLAY\_STATE |
| NEW\_GAME\_AI\_BUTTON | PlayerColor.BLUE  PLAY\_STATE |
| EDIT\_BUTTON | EDIT\_STATE, displays “edit mode” |
| BLUE\_BUTTON | Displays "Blue selected" |
| RED\_BUTTON | Displays "Red selected" |
| NONE\_BUTTON | Displays "None selected" |
| DONE\_BUTTON | If done is pressed while in the edit state this means done editing so try go to the play state |
| LOAD\_BUTTON | Load the models from files |
| SAVE\_BUTTON | Save the models to files |
| MAIN\_MENU\_BUTTON | START\_STATE |

**IN:** ActionEvent e

**UPDATES:** NONE

**OUT:** shows on the board display

ActionPerformed (ActionEvent e) is used when an event on the displayed board happens. The method reacts to the user input by updating appropriate models.

The method first decides what to do from which case happens, that is which display is pressed on the edit state, and sets the position to be the current edit color.

update (Observable arg0, Object arg1)

**IN:** Observable arg0, Object arg1

**UPDATES:** NONE

**OUT:** shows on the board display

This method is called by {@link Observable} classes to let the controller know that the model has been updated.

These are the different cases of if statements and what are expected from each.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **if** (arg0 == boardModel) | view.drawModel(boardModel) | Update view with new board model |
| **if** (arg0 == stateModel) | GameState currentState = stateModel.getState() | Update view with new state info |
| **if** (panelMap.containsKey(currentState)) | view.setCurrentMenu(panelMap.get(currentState)); | Display the panel |
| **if** (currentState == GameState.***PLAY\_STATE***) | PlayerColor currentPlayer = stateModel.getCurrentPlayer() | Update view |
| entPlayer != PlayerColor.***NONE***) | view.setTitleLabel(currentPlayer.toString() + "'s turn") | Ignore if player is none |
| if (currentState == GameState.WIN\_STATE) | view.setTitleLabel(currentPlayer.toString() + " won!"); | display a winning message |
| if (currentState == GameState.DRAW\_STATE) | view.setTitleLabel("Draw :("); | display the draw message |
| if (currentState == GameState.START\_STATE) | view.setTitleLabel("Welcome"); | reset the message and the board pieces |
| if (stateModel.getState()== GameState.PLAY\_STATE && stateModel.getAIPlayer() == currentPlayer) | protected Integer doInBackground()Calculates the next move | If it's the AI's turn, figure out the next move and do it Run the AI on another thread to keep the GUI responsive |

ConnectFourAI class Reference

ConnectFourAI()

**IN:** BoardModel board

**UPDATES:** GRID\_WIDTH = board.getGridWidth(), GRID\_HEIGHT = board.getGridHeight()

**OUT:** NONE

This Creates a new AI object

getBestMove()

**IN:** PlayerColor player

**UPDATES:** NONE

**OUT:** return bestMove

This gets the best move.

negamax()

**IN:** int depth, int move, PlayerColor player

**UPDATES:** NONE

**OUT:** return bestValue

|  |  |
| --- | --- |
| CASE |  |
| if (winner == player) | return MAX\_SCORE |
| if (winner == player.opponent()) | return MIN\_SCORE |
| if(board.getPieceCount(PlayerColor.NONE) == 0) | return 0 |
| if (depth == 0) | return evaluateBoard(player) |

This figures out what the next move should be using the negamax algorithm

evaluateBoard()

**IN:** PlayerColor player

**UPDATES:** NONE

**OUT:** return totalValue

|  |  |
| --- | --- |
| CASE |  |
| Horizontals | totalValue += getConnectFoursValue(0, GRID\_WIDTH-3, GRID\_HEIGHT, 1, 0, player) |
| Verticals | totalValue += getConnectFoursValue(0, GRID\_WIDTH, GRID\_HEIGHT-3, 0, 1, player) |
| \ Diagonals | totalValue += getConnectFoursValue(0, GRID\_WIDTH- 3, GRID\_HEIGHT-3, 1, 1, player) |
| / Diagonals | totalValue += getConnectFoursValue(3, GRID\_WIDTH, GRID\_HEIGHT-3, -1, 1, player); |

This evaluates the value of the board from the current player's perspective. It checks each possible connect four and how much progress the player made in getting them.

getConnectFoursValue()

**IN:** int startX, int maxX, int maxY, int dx, int dy, PlayerColor player **UPDATES:** NONE

**OUT:** return totalValue

This gets the total value of the specified connect fours from a given players perspective**.**

GameStateModel class Reference

This class is used to represent the current state of the game.It is so a class can observe it for updates, such as when the state changes.

GameStateModel

IN: NONE

UPDATES: NONE

OUT: NONE

Initialize in the state of GameState.START\_STATE.

setState

IN: GameState state

UPDATES: this.state

OUT: NONE

Set a new state and notify observers of the change.

getState

IN: NONE

UPDATES: NONE

OUT: gameState

Get the current state. Returns the current state

|  |  |  |
| --- | --- | --- |
|  |  | **Result** |
| **getState** | If Colour = Red | editColor = Red |
| if Color = Blue | editColor = Blue |
| if Color = None | editColor = None |

setCurrentPlayer

IN: player

UPDATES: currentPlayer

OUT: NONE

Sets the current player. Only accepts actual players, it does nothing is NONE is passed in. Also the current state needs to be the PLAY\_STATE.

**PseudoCode:**

**if** player != PlayerColor.NONE **AND** state = GameState.PLAY\_STATE

currentPlayer = player

getCurrentPlayer

IN: NONE

UPDATES: NONE

OUT: currentPlayer

Get the current player whose turn it is.

nextTurn

IN:NONE

UPDATES: currentPlayer

OUT:NONE

Alternate the current player. Goes to the other players turn

**PsudoCode:**

**if** currentPlayer  = blue

  setCurrentPlayer = red

**else** setCurrentPlayer = blue

setAIPlayer

IN: player

UPDATES: aiPlayer

OUT: NONE

Set which player is the AI player

getAIplayer

IN:NONE

UPDATES: NONE

OUT: aiPlayer

If there is an AI player, get which player it is.

saveToFile

IN: filename

UPDATES: NONE

OUT: NONE

Capture the current state of this object and stores it in a file for later retrival.

loadFromFile

IN: filename

UPDATES: NONE

OUT: NONE

Reloads the state of this object from a previously save state

BoardViewclass Reference

This class is responsible for the GUI of the connect four game. It sets up the frame and all other aspects of the display.

BoardView

IN: boardWidth, boardHeight, controller

UPDATES: BOARD\_WIDTH, BOARD\_HEIGHT

OUT: NONE

Get the current edit color. Return the current edit color.Initialize the view. When the constructor completes, the frame will be created and be getting user input. param boardWidth the width of the board in pieces param boardHeight the height of the board in pieces param controller the controller to send user input updates to.

setUpFrame()

Creates the frame for display and initializes all its components

INPUT : None

UPDATES: The GUI

OUTPUT : Frame with all the buttons and panels dispalyed

This method creates the frame for the display and initializes all of its components.

createHolderImage()

INPUT : Color color - red or blue

UPDATES: The GUI

OUTPUT : createHolderImage holderImage : holder image on the side of the board to hold the color piece

Dynamically creates an image of a piece in a holder using the color specified as input.

createPieceImage()

INPUT : PlayerColor color : red or blue color or none for blank, Color borderColor : color of the border

UPDATE: The GUI

OUTPUT : createPieceImage holderImage : an image representing a piece of a certain color

Create an image of a piece in a sqaure of the connect four board. This image can be tiled to create the whole board

lookupComponentType()

INPUT : Object obj : an unknown object that is a GUI component

UPDATE:NONE

OUTPUT : BoardComponentType.INVALID\_COMPONENT : If it is invalid componentMap.get(obj) : If it is a valid GUI component

Determine which GUI component an object is and if it is even one.

lookupButtonPosition()

INPUT : Object obj : a button object

UPDATE: NONE

OUTPUT : position : the position on the board the button occupies or null if it doesn’t

Determine the x,y position of a button on the board.

drawModel()

Displays the color pieces of the board to the screen

INPUT : BoardModel model : the model to draw

UPDATE: The GUI

OUTPUT : None

Display the pieces of the board to the screen.

highlightPiece()

Highlights the button with a yellow border at a certain spot.

INPUT : Vector 2D position : the position of the button to highlight

UPDATE:The GUI

OUTPUT : None

Highlights the button with a yellow border

setCurrentMenu()

INPUT : BoardPanels panelID : the id of the panel to display

UPDATE:The GUI

OUTPUT : None

Displays the menu specified

setTitleLabel()

INPUT : String text of the label

UPDATE: The GUI

OUTPUT : None

Sets the title label

setStatusLabel()

INPUT: String text

UPDATE: titleLabel

OUTPUT: NONE

Sets the status label

PlayerColorclass Reference

This enum represents a player color in connect four. The none player can be used to signal an error in situations where it doesn't make sense.

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

opponent()

IN: NONE

UPDATES: NONE

OUT: PlayerColour

This gets the opposite player

BoardComponentTypeclass Reference

This enum is used to identify components.

**IN:** NONE

**UPDATES:** NONE

**OUT:** NONE

# Evaluation of the adequacy of the design (4.5)

We decided to make a metrics chart before designing the program to help us accurately evaluate our design. This section is very similar to assignment 2.

\*\*blue indicates what was added/changed.

Table : Task Metrics

|  |  |
| --- | --- |
| Task | Rating (1-10) |
| Follow all the requirements | 10 |
| Design using the concept of MVC | 9 |
| Efficient implementation of game logic | 9 |
| Efficient choice of data structure | 8.5 |
| Highly modular code, limiting methods in a class | 7.5 |
| Separation of concerns | 9 |
| Replication of the Connect 4 example view | 10 |
| Artificial Intelligence | 10 |

1. The first task we rated our project 10 out of 10 because we believe the design fulfills all the requirements specified, this is mentioned and described in detail in the traceability summary above.
2. We rated your 9/10 because we strongly believe our design follows the concept of a Model-View-Controller in your design, we examined how our design follows this concept in MVC explanation section above. The reason it is a 9 and not 10 is because we believe there might be a way to implement the MVC structure a bit better.
3. We were able check for a winner and errors with an algorithm that was able to check all the winning positions in 69 compares. We were able to figure out that there are only 69 different ways to win connect 4. 24 ways moving bottom to top (horizontally), 21 ways moving left to right (vertically), 12 ways diagonally left the right and another 12 ways diagonally right to left. We rate this design choice 9/10 because it an efficient algorithm but they might be a way to implement the algorithm better, maybe with recursion rather than for loops.
4. We rated our data structure implementation 8.5/10 because the data structure is a double array that represents the row and columns of the grid. Each row and column is associated with the x,y coordinate, this makes it easy for the program to know where a piece is inserted. This is a very good design choice however we believe they are a better way to do this task that we have not learned yet.
5. We rated the task of highly modular code and limiting methods in a class a 7.5/10 because we believe we have a highly modular code but some of the class had a lot of methods and variables. We think it is possible to make it more modular but we are not sure how to implement it. We can also improve design by making these methods as private as possible and using more information hiding.
6. We rated our ability to separate concerns a 9/10 because we were able to follow a MVC pattern, we believe following this pattern allowed us to separate the concerns of data structure, error checking, game logic (Model), GUI (View) and User interactions (Controller). The reason it is a 9 and not 10 is because these is room for improvement and we believe with more time we can fully separate all the concerns.
7. We rated GUI 10/10 because it is an exact replication of the Connect 4 example view.
8. The AI gets a 10/10 because nobody could beat it. It seems like it make all the right moves. Even when we thought it made the wrong move, it will still managed to win the game turns later. The computer was much smarter than us.

# Specific section on the algorithm (4.6)

\*\*blue indicates what was added/changed.

Refer to the code, ConnectFourAI.java. Uses the negamax(),evaluateboard() ,getConnectFoursValue() and getBestMove() methods to help determine the computers next move. The code for ConnectFourAI.java is well documented so it will be easy to see how the computer determines the next move.

# Document the code (4.7)

Refer to the code, it is well commented.

# Testing (5)

**\*\*\*Screen shots of test are included in Testing Screenshot Folder, all other components were tested in the Assignment 1 and 2.**

**\*\*\* Programs Document also contains a JUnit test for the BoardModel. (*BoardModelTest.java*) All the methods are tested except the save and load game, which are tested below.**

**Test Strategy:**

* Black-Box Testing:
* Tested expected functionality
* Choose between 2 players and 1 player vs A.I
* Alternate Players with A.I
* Start New Game
* Save Game
* Load Game
* Artificial Intelligence

**Test Report:**

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Test Result** | **Test Result** |
| Choose between 2 players and 1 player vs A.I | When you click 2 Player button you can all the moves are made when a spot is clicked. When you click 1 player button every other move is automatic. | When you click 2 Player button you can all the moves are made when a spot is clicked. When you click 1 player button every other move is automatic. |
| Alternate Players with A.I | The players should alternate. | The players should alternate. |
| Start New Game | Clear screen and start a new game | Clear screen and start a new game |
| Click Save | Save the state of the game to a file | Save the state of the game to a file |
| Click Load | Load the state of the game from a file | Load the state of the game from a file |
| Artificial Intelligence | The A.I. makes the correct moves and hardly loses. | The A.I. makes the correct moves and hardly loses. |

Table : Test Results