ConnectFour Design

Manaar Hyder, James Lee, Maya Ramamurthy,

Randa Mohsen, Danielle Shwed

Comp Sci 2ME3, Group 2

# Classes and Modules Used

## ConnectFour Class

The main purpose of this class is to handle the graphics component, listen to user actions (mouse clicked) as well as act as the Main class which we run to start the game. This class encompasses a Drawing Panel class where we create our drawing panel and are able to initiate buttons and control their actions/ function with the action listener. This class also includes paintComponent which controls the graphics. This means it draws the discs, the board and the “holder” for the discs. The ConnectFour class also includes mouseClicked which watches where the user clicks, then calls on placeDisc to save the position in an array, and also listens for where the user clicks in order to determine which Players turn it is.

ConnectFour()  
This is the constructor for the ConnectFour class. It sets the label of our window to be “Connect Four” as well as creates the contentPane. It then adds a mouseListener, and a DrawingPanel and BorderLayout to the contentPane. Here is also where a new object of Memory is created.

DrawingPanel()  
The DrawingPanel method is included in the ConnectFour class. It’s here that the basic parameters of the drawingPanel are set. This includes the size of the panel, whether it’s resizable, whether it is opaque or not, the background color and sets the layout.

The start button, end button, button to save the game and load the game are also created. An action listener is added to each of the buttons as well.

Within the action listener for start button (actionPerformed()) the positions array, which is the array to store where all the discs on the board are, is set to 0 at all positions. The Boolean flag for starting the game is set to true and a player to begin with is randomly selected. Also all the “winning positions” in the winPos array is set to empty.

Within the end button action listener, a nested for loop is used to check whether red and blue have not had an equal amount of turns. If not, an error appears. If there is an equal amount of turns had checkWin is called in order to determine if there is a winner. If there is, showWin is called and the winner is displayed on screen.

Within the action listener for the save button, saveGame() is called (a method within Memory) in order to save the current positions array into Memory.

Within the action listener of load Button, loadGame() is called (a method within Memory) in order to obtain the positions array from the past ConnectFour game.

There is also an action listener for the button endgame(); which when clicked, will cause the game to close.

paintComponent()  
This is where the graphics are handled. A 2D graphics object is created and the basic graphics components are drawn. Each time paintComponent is called it is almost like resetting the graphics.

This is where the board is drawn using horizontal lines, vertical lines and circles.

placeDisc()  
Based on which player is the current player, it draws the appropriate coloured disk where the user has clicked as well as stores it in the positions array.

## checkWinability Class

This class is used be Check in order to determine if it is indeed possible to win the game or not.

## Memory Class

This class is used to save the current game state into a text file, through the use of the saveGame() function, so the user could resume the game that they saved by a click of a button, through use of the loadGame() function.

saveGame()

This method uses a FileWriter and a nested for loop inside another one to write every item in the 2-d array that is the game board. This method throws the IOException in case there is a problem with the output.

loadGame()

Using a while loop, this method takes in every line in “savedGame.txt” and returns a 2-d position array that is sent back into the main class for use. Throws the FileNotFoundException in case the text file is not found.

## Move Class

The main purpose of this class is to use a mouse listener in order to determine where the user has clicked and then subsequently call on PlaceDisk to indicate the position the user has clicked as well as which player is the active player. It also uses a function called storeMem()in order to save the position the user would like their disk to be placed in the array in Memory.

mouseClicked()  
This method uses MouseEvent e in order to save where the user has clicked into 2 variables: xPos and yPos. Based on where the user clicks, it determines whether the blue or red icon has been clicked using two if statements with hardcoded values of where these icons lie on our board. If the user clicks somewhere on the board, a third if statement is used to ensure the user has clicked in bounds. If they have clicked in bounds, the point which has been clicked and current player is passed an object of PlaceDisc.

storeMem()  
The function storeMem() is used to pass the updated array into Memory. This is done by using the returnArray() method from PlaceDisk and passing it through saveGame in the Memory class.

## PlaceDisk Class

## Restore Class

The purpose of the Restore class is to load a past game from memory in order to continue to play it. This is done by simply getting the past array from Memory and setting the current array to be equal to the array from the past game.

getArrays()  
This method creates an object of the Memory class as well as an object of the PlaceDisk class. A variable called newArray is set equal to the array from the past game. The old array is obtained by the function loadGame() from the Memory class. updateArray is then called, passing the newArray through it in order to fully load the past game.

## checkWin Class

The main purpose of this class is to serve for a way for the game to check if the user has won based on the moves that have been made in the game as of yet, by using an array of given locations to figure out if there currently is four in a row through a series of tests. This was placed into a class of its own for the purpose of determining the win without having to be placed in the same class as the user interface, however due to the interaction, it is not able to be kept completely private. The methods that interact with the ConnectFour class are left public to allow for this interaction, however the remaining methods are not.

duplicate ()

This class interacts with the checkWin () method, in which it receives the input of the positions of arrays, with the format of three possible values stored in it to mark the colour of the counter if applicable: 1 if it is a red counter, and a 2 if it is a blue counter; 0 otherwise.

Using this, it will create a new array of the same size and type (integer), which will now contain a 1 if it is red, and a -1 if it is blue; 0 otherwise. This is done through two nested for-loops to easily access index positions and to allow easier access when writing in the positions, as shown below:

for 0< i< cols

for 0 < j < rows

if positions(i,j) is 1, then value(i,j) is 1

else if positions (i,j) is 2, then value(i,j) is -1

otherwise must be 0 so value(i , j) is 0

The purpose of this method was to provide an easy way to quickly total up the value of four positions in a row to see if it added up to a sum of 4 or -4, in which there would be a win, all without having to modify the original array of positions, which would be trickier to use when it came to checking the positions, as two blues in a row would equal 4, which would throw off the calculation of the game. It also would be far less efficient if we simply checked each time if the positions array at a certain index contained a 1 or a 2 and added a 1 or -1 to the array used to check the wins, thus why we made this decision.

This method is private to keep it hidden from the user, so it is not accessible from the user interface class. This will ensure the array cannot be tampered with from the outside and will remain intact due to this reasoning.

getPos()

The purpose of this is to be something to interact with the ConnectFour class once it has checked to see if a winning move has been made. If this is the case, then it will return the winning positions obtained through both the setPos() method and the checkWin() method. This will then, once called upon, deliver the array, which is later used to draw the dots on the screen to show that a win has been made. It has no input variable, but receives the array winPos which is an array of coordinates of any winning moves, which is the output for this method.

**setPos()**

The purpose of this method is to obtain the positions array found in the checkWin() method, and return it to the getPos() method which can use it to find the winning moves, as checkWin() merely returns if either the red player or blue player has won, or if the game is still in progress. There is no or output for this method, as it merely sets the variable winPos to the input variable win, an array which receives it’s input from the checkWin class after determining that either red or blue has won.

checkWin()

This is the method that determines if any winning moves have been made. This works by using the input of the array of filled positions, and in doing so, calling upon the duplicate() method to receive the array of 1 and -1 values to figure out if there is a winning move being made. There are four possible ways for there to be four in a row: by rows, columns, left diagonal, and right diagonal. We decided to divide it up this way to check systematically if a winning move has been made, and if it has, automatically return the winning player, as opposed to checking all possible combinations and returning it at the end.

This has the input of the array of positions to calculate the wining moves, and will return the variable total, which will either be a 1, if red, or a 2 if blue. If no win has been calculated at the end, it will return a 0 to show that the game is in a draw state.

The pseudo code can be seen below:

// check rows first

for 0=<cols<6, col increasing

for 2<=row<0, row decreasing

total = value at (col,row) + (col + 1,row) +(col + 2,row) +(col+3,row)

if value = 4 or -4 then there is a win, so send the positions to setPositions,

return 1 or 2 depending

// check cols

for 5=<row<0, row decreasing

for 0<=col<3, col increasing

total = value at (col, row ) + (col, row + 1) + (col, row +2) + (col, row+3)

if value = 4 or -4 then there is a win, so send the positions to setPositions

return 1 or 2 depending

// check left diagonal

for 6 >= col > 2; col is decreasing

for 5 >= row > 2; row is decreasing

total = value at (col,row) + (col – 1,row – 1) + (col – 2,row – 2) + (col – 3,row – 3)

if value = 4 or -4 then there is a win, so send the positions to setPositions,

return 1 or 2 depending

// check right diagonal

for 0<=col<4, col is increasing

for 5 >= row > 2, row is decreasing

total = value at (col,row) + (col+ 1,row – 1) + (col+ 2,row – 2) + (col + 3,row – 3)

if value = 4 or -4 then there is a win, so send the positions to setPositions,

return 1 or 2 depending

return 0 to show no win.

## currentPlayer Class

## showWin Class

The intended purpose behind this class is to display if there is a change in the status of the game. When the ‘Done Turn’ button is clicked after each turn, this will send the position of the tiles to the showWin class from the ConnectFour class. In doing so, this class will determine if there is one of three possible outcomes being achieved. This is done through the use of the following three methods.

show ()

The purpose of this method is to use information from the ConnectFour class to determine if there is a change in the status of the game. The input of this method is the positions array, which contains whether there is a red, blue, or no tile in a certain position to better determine if a win has been made. This is done through the use of 3 different if statements to show the three different possibilities, as seen in the pseudo code below:

*If total is one or two*

*If one then set colour of progress text to red then display ‘Red Wins’*

*Else If two the set colour of progress text to blue then display Blue Wins’*

*Else if total is zero then draw game, and display progress text as such in purple*

*Else, game is still in progress, and keep that as status*

This method then sends the colour of the text to the setColour() method which will be called upon later from the ConnectFour class. After this, the class will return the progress of the game to ConnectFour so the text will update accordingly.

setColour()

Will get the colour from the show() method, and set the colour variable in the class to what it received. This is necessary so the colour can be sent back to the ConnectFour class so the colour of the text displayed can be updated.

getColour()

This method is what will be called upon by the ConnectFour class so it will receive the colour of the text. There is no input, but the output will be the colour. This is necessary as it was one of the simplest way to send the colour back, as all of this is done in a separate class as opposed to being done in the main class to prevent cluttering.

## checkWinability Class

## Memory Class

## PlaceDisk Class

# Public Entities

## ConnectFour Class

ConnectFour()  
This is a constructor so it must be set as public.

## Move Class

*mouseClicked()*

This is public because it must interact with the PlaceDisk class, as well as the positions clicked by the user must be accessed throughout various classes.

*storeMem()*  
This method directly interacts with the Memory class. It calls on the saveGame() method and also needs to be public in order to be used to store the current move, which will be called on from outside the Move class.

## Restore Class

*getArrays()*  
getArrays is public in order to be accessed when the user clicks “load game”. The method needs to be accessible by ConnectFour within the load button action listener. It also calls on Memory and PlaceDisk which adds to the fact why it is public.

## checkWin Class

There are no public variables in this class that interact with the entire class as a whole. While the majority of items in this class are private, there are a few public methods as they are the ones that directly interact with the ConnectFour class and need to remain public to do so.

getPos()

This method is public as it interacts directly with the ConnectFour class. If the ConnectFour class receives knowledge that a win has been obtained, it will send for this class to determine what the winning positions are to be able to draw the black dots to show the win. This method only uses the private variable winPos to return the positions, which will be discussed later on.

checkWin()

As this method calculates the winning score if applicable, it is public. However most of the variables that it uses are private. There are four loops that check the four different types of wins, by rows, columns, left diagonally, and right diagonally, and in doing so, there are two variables that are used for indexing, which are row and col. These two variables are re-declared in each set of for-loops based on the values they need to hold to properly search all possible combinations, and do not interact with any of the other methods outside the individual set of loops.

There is also a variable value, which is a 2D array that holds the same positions as the positions array, except instead of a 1 or 2 for red or blue, it holds a 1 or -1, so the variable total can add up what is in each position to return the according total. This is public due to the fact that it involves the original positions array from the ConnectFour class, and is sent to duplicate() to be made into the version it is now.

## currentPlayer Class

## showWin Class

There is one variable that is used publically throughout the entire class, and that is the variable *x*, that is of a type ConnectFour to allow access to this class and to its variables. For the most part, the variables in this class belong to ConnectFour, so when called upon it is done so as the following:

*x.colour = Colour.blue;*

This is done so we do not need to recreate each variable that we need, and at the end, we only need to send back those that are relevant. There are two variables that use the following manner, and those are *colour* and *progress,*  so we can send back the status of the game to be displayed with the proper text and colouring.

show()

This class is public essentially for the above reasons. We need to be able to display the correct status of the game, so using the positions array and the total score of the game, we can properly determine whether a win has happened, and if that is a red or blue win. This method also interacts with the setColour method, a private one, to send back the proper colour of the status of the game for later access by the getColour method from ConnectFour.

getColour()

The getColour() method interacts directly with ConnectFour thus the reason that it remains public. This method is rather simple in which it sends back the private variable *colour* to ConnectFour, which can be one of four colours: black, red, blue, or magenta.

## Memory Class

saveGame()

This method must be public as the main class needs to call on this method since button clicks are handled in the main class.

LoadGame()

Like the saveGame() method, the class ConnectFour must be able to call on this method in order to load the game.

# The Uses Relationship

# Private Entities

## ConnectFour Class

Since this class is the main user interface class, the majority of items in it are public, including variables and methods. One private variable is the DISC\_RADIUS variable which contains the size of the circle, so that it cannot be modified by any other classes.

DrawingPanel()

This class is private due to the reasoning that we do not want the drawing of the board to be tampered with. In addition to this, it also contains the buttons and their action listeners, which determine what happen when each button is to be clicked, and that is something we want to reamain free of outside influence.

## CheckWin Class

There are two variables that are used privately throughout this entire class. The first is *total*, a variable that helps determine if the score of the game is equal to either 4 or -4, in which a win has been obtained, however it is used in majority by the public checkWin() method. The second is the *winPos* array, which starts off as empty, but in the case that a win has been achieved, it is filled with the coordinates of the winning positions to later be accessed by ConnectFour so black dots can be drawn over these spots.

duplicate()

This method reads in the positions from checkWin, and uses a variable value to get the positions with 1, -1, or 0 depending on its status. However despite this, it does not have any private variables, as this is the only one it uses. The main reason for its private status is so that is cannot be accessed from other classes to tamper with this or allow it to be modified by others. This will help ensure the integrity of the game.

setPos()

This method mainly just sets the currently empty winPos array to the winning positions to be accessed later by getPos, to be sent back to ConnectFour. It is private to help ensure that the winning positions are properly maintained and not modified by any other classes along the line.

## showWin Class

The first thing about this class that is private is the variable colour, as it is accessed throughout the class, however we only want the colour’s value to be accessed internally due to the fact that it shows the colour of the winner on the User Interface. This variable is mainly used by the setColour method and the getColour method (previously mentioned).

setColour()

The main purpose of this private class is to set the colour to that determined by the show() method to properly display the colour of the entire progress section to prevent tampering by any other classes. There is very little else to this method, as this was it’s only purpose.

## currentPlayer Class

## checkWinability Class

## Memory Class

## PlaceDisk Class

# Review Of Our Design

# Testing