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 *DrawingPanel* 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 player’s 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 (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 using the method *Random* in class currentPlayer. Also all the “winning positions” in the winPos array is set to empty.

Within the action listener for the “Save Game” button, *saveGame()* is called (a method within class Memory) in order to save the current array representing positions into Memory.

Within the action listener for the “Load Game” button, *loadGame()* is called (also a method within class 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 window 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 and vertical lines, as well as circles to represent the disk slots.

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

## checkWinability Class

The purpose of this class is to determine whether it is possible to win the game or not. After every move, the *check* method within checkWinability is called by class Check. It returns a Boolean value: **true** if the game is winnable or **false** if the game board no longer holds possible winning positions.

check()

This is the one and only method used in checkWinability. First, it initializes 2D array that represents the board in terms of values for each red (1), blue (-1), or none (0). These values are determined using the *duplicate* function in class checkWin, for code efficiency*.* Next, there are four different for-loops that check for types of wins:

**Horizontal Wins**

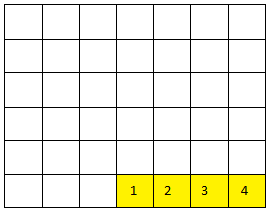


Diagram to represent what a horizontal win look like and how the positions are determined.

Pseudo code:

for the last three rows

for the first three columns

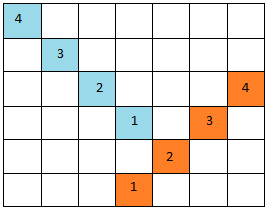
if ( value of position 1 = value of position 2 ) or (value of position 2 = 0) and if ( value of position 1 = value of position 3 ) or (value of position 3= 0)

and if ( value of position 1 = value of position 4 ) or (value of position 4 = 0)

it is possible to for a win using these positions

**Vertical Wins**

The check for vertical wins follows a similar style to horizontal wins, except the positions are defined to be on top of one another rather than beside. If the positions all hole the same value, or 0, then it is possible to win and **true** is returned.

**Diagonal Wins**

There are two diagonal checks that follow similar for-loop design as the vertical and horizontal check. One is a left leaning diagonal check; the positions are represented in blue on the side diagram. The other is a right leaning diagonal check, represented in red.

## 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. This is done 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 is to provide an easy way to quickly total up the value of four positions in a row to see if they add 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 comes 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 public only to allow the method *check* from class checkWinability to also use it as a positions to values array converted. This is chosen simply for efficiency, since duplicating code is unnecessary if it will not be altered specifically.

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.

Pseudo Code:

// 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

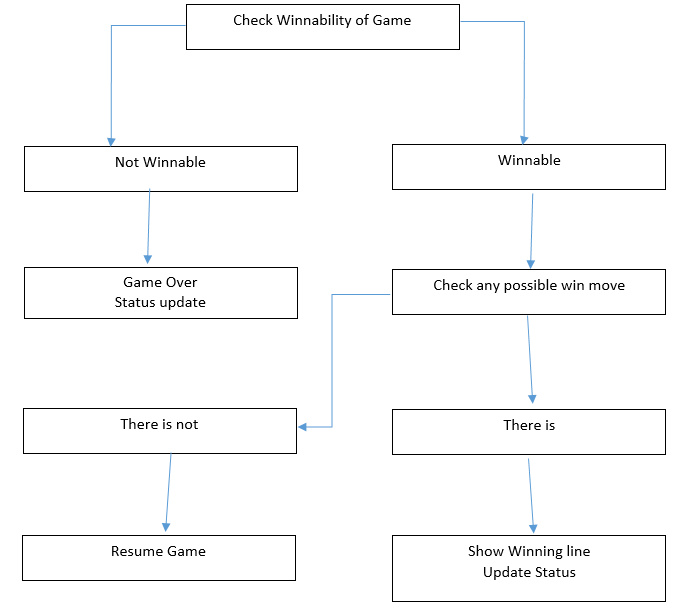
return 0 to show no win.

## Check Class

By being called on by ConnectFour every time a valid move is made on the board, this class controls the winning status of the game. First, it uses checkWinability in order to figure out whether the board will allow for any more wins. If it returns true, then checkWin is used to check for an actual win, rather than just a potential one. If yes then the game ends and the winner is displayed using showWin. If no, then the users are able to continue playing the game. If checkWinability had returned false, Check would then stop the game and change the status of the game from “Game in Progress” to “GAME OVER”.

Update()

This method is called on whenever a move is being made, so the first thing it does is implement the gravity concept to make sure all disks are supported by others below them. Then, Update() does the following:



returnPos() and returnTotal()

Return global variable pos and total which represent the winning positions of the game, if won, and the value representing which player won – respectively.

## 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.

## PlaceDisk Class

This is a simple class that is used by the class Restore, handling the GUI aspect of placing the stored disks onto the game board.

The updateArray() method send the stored positions to the current positions, while place() takes in two parameters:

* Point P – the coordinate of the user’s mouse click
* String player – the current player indicated by “Red” or “Blue

## Restore Class

The purpose of the Restore class is to load a past game from memory in order to resume playing. This is done by retrieving the past array from Memory and setting the current array to be equal to it. This class also uses the class PlaceDisk in order to display the restored game onto the board 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.

## currentPlayer Class

This class handles whether the current player is Red or Blue simply by creating a private global variable which hold a string “Red”, “Blue” or “None” and manipulating/referring to that variable through methods which are called by other classes such as ConnectFour.

setPlayer()

This is a private method which takes in a parameter String string and equates currentPlayer to it. This is used when the method *Random()* determines a random player, which is needed for the start of the connect four game.

getPlayer()

This method is used a function to alternate the players. If the currentPlayer is None, *Random()* is called and a random player is assigned to the game. This player is returned and the first turn commences. If currentPlayer holds Blue, then the get method will change it to Red and return. The opposite happens if it currentPlayer is Red. This is used in the class ConnectFour where every turn is automatically set to the other player in the game, in order to prevent an uneven balance in turns between the players.

Random()

This private method randomly generates a number between 1 and 2 into an integer variable. If the result is 1, *setPlayer()* is used with the parameter being “Blue”. If 2, then the parameter is “Red”.

## showWin Class

The intended purpose behind this class is to display if there is a change in the status of the game. Position of the tiles are sent to the showWin class from the ConnectFour class. This class will then 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.

# Public Entities

## ConnectFour Class

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

## 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.

## Check Class

This class only consists three public methods: Update, returnPos and returnTotal. These methods are made for the sole purpose of sending private information from Check class to ConnectFour class.

## currentPlayer Class

This class includes only one public method.

*getPlayer()*

This method is accessed multiple times by ConnectFour in order to declare and reset the current player name as the game runs. Thus, it needs to be public.

## 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.

# 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 remain 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 its only purpose.

## currentPlayer Class

This class holds a private variable of type String which is used to represent the name of the current player. It is private simply because it only needs to be accessed by three methods of the same class. Those methods include two that are also private.

*setPlayer()*

This method simply deals with setting and resetting the global variable within the class. Since does not need to be accessed by any other class, it is private.

*Random()*

Although the game requires the first player to be randomized, this function does this privately and sends its information to the public function *getPlayer()* of the same class.

## PlaceDisk Class

This class revolves around a private variable DISC\_RADIUS of type int which represents the radius internally and does not need to be accessed by any other classes.

# The Uses Relationship

Memory

CurrentPlayer

Restore

ConnectFour

PlaceDisk

Check

CheckWin

CheckWInnability

Example

ShowWin

uses

Class A

Class B

# Review of Our Design

# Testing

Gravity

For the last version of the game, the disc was supposed to appear exactly where clicked. However, this time the game is following more rules, and a disc unsupported by one underneath must keep going down the rows until it is either supported by another disc or hits row 0, at which point it is supported by the base. The code for enabling gravity simply uses a for-loop to recursively ensure that for rows 1-5, there must be a pre-existing disc right below it. Otherwise it will go down the row until it finds support. This is very important to check, because when connect four is played outside of the virtual world gravity already exists and pulls down any unsupported pieces. It is physically impossible for a piece to float above an empty space and then for another piece to somehow be added to that empty spaces. The online game must match the pre-existing, original game, thus gravity must factor into our version of connect four. Since we are not yet expected to implement Junit Test Cases for this course, we manually tested for various cases. We clicked on various rows of an empty column and ensured that it never ‘floated’ or overrode any existing disc by ‘falling’ too far down. We clicked the disc into places where it was supported in the clicked position by a disc, or by the base. The tests all passed. We created gravity in the virtual world of connect four

Game in Progress Status

Our code has a Boolean variable ‘start’ that, when true allows the players to make their moves, thus signifying a game in progress. Therefore, when ‘start’ is true, the words “Game in Progress” should be displayed on the screen, right above the ‘board’. This alerts the players that a game is already in progress and they don’t necessarily have to start a new game because the current game does not have a result yet. The Boolean ‘start’ is false when either the game has yet to start, when someone has won the game, or the game ended in a draw. In any of these cases, since the ‘start’ Boolean is false, the “Game in Progress’ banner will not be shown on screen, as a game will not be in progress. It can be checked by playing the game, and ensuring that the banner is only on screen when it is possible to make a move – i.e. from new game until a draw is reached or a winner is decided.

4 types of winning

When playing a game of connect four, there are 4 possible ways to win a game. Right diagonal, left diagonal, vertical and horizontal. If a player gets four in a row in any of these configurations, she wins the game. Our code uses a class called CheckWin to check for wins. It used a nested for-loop to go through rows and columns, thus exhausting every possible location. Starting at the bottom, the loops check for 4 types of behaviours. The vertical win check checks that for every disc, whether or not there are 3 more disc of the same colour right on top of it. The horizontal win checks that for every disc, whether or not there are 3 discs of the same colour beside it. Right diagonal checks for 3 discs, each one slot to the right and one row above. The left diagonal does the same, but checks for discs located one column to the left and one row above. The code for this is checked with counter and for-loops as mentioned. Horizontal checks that column + I in a row is the same color from I = 0 to I = 3. Vertical checks for row + I in a given column. Right diagonal checks for col + I and row – I from a given point col, row. Left diagonal checks for col - I, row – I from a given point row, col. For these checks I is always a counter from 0 to 3 since we need to check for 4 in a row. This is very important to be checked because it determines the winner of the game. If CheckWin did not happen regularly, it is possible that both teams make winning moves, and then the game is a draw, when in reality one of the 2 people should have won, thus making the output inaccurate. This can be checked by playing the game a few times, make sure that the game is played and won at least 4 times in 4 different ways. Once CheckWin determines that a game is won, the Boolean ‘start’ switches to false, the “Game in Progress” banner is removed and the players can no longer make any moves, besides clicking the ‘New Games; button for a rematch.

Show Winning Line

When playing Connect Four in the real world, there’s no advanced technology involved, so a winning connection tends to blend in with the other discs until pointed out by the player that placed the winning disc. With our version of the code, we believe that no potential wins should be ignored, as that is what the assignment requires of us. Therefore, when CheckWin finds a win, it calls on the “ShowWin” class to show that win. The showWin class then prints a black dot in the centre of the winning connection. This is implemented by collecting the values from the nested for-loops used to check for wins. The 4 coordinates are passed on to the showWin class which then prints a smaller, black circle in the four winning places. This is important to show the players that 4 discs have been connected and that the player connecting those discs has won the game. This can be checked by playing the game, and ensuring that someone actually wins and that the game does not end in a draw.

Winning Statuses

When one colour wins, it is not sufficient to simply mark the winning connection on the board. The win has to be made official with the “Blue Wins” or “Red Wins” banner above the board, replacing what used to be the “Game in Progress” banner. The code checks what color the connected discs are and prints the respective banner. This is important to reinstate the winner of the game. It can be checked by playing the game, and ensuring that the winning colour is declared as soon it wins.

Game Over

The checkWinnability class checks the game while its still running. It uses the same nested for-loop from the CheckWin class as detailed above, however the way it is used is slightly different. While CheckWin looks for 4 discs of the same colour in the right position to check for a win, checkWinnability checks for possible wins. It divides the number of remaining spaces in half to allocate some each color. It then runs through the for-loops to essentially check wins. But where It is possible to complete a connection, the loop continues to run. If it is impossible to complete any connections existing connections, or if they just don’t exist, then it is impossible to win the game, and a banner showing “Game Over” is shown above the board. This is important because if the players are playing competitively, they don’t have to keep making moves on a game in which there will be no winner. They have the option of starting a new game to try again. This, like most other functions can only be checked by playing the games. This will require a bit of mind work to set it up in a way that will definitely lead to a draw. Keep track of when it becomes impossible to win and ensure that the program gives the same result.

Saving/Restoring

The save/restore button is used to essentially pause a game and access it later. The save button saves the progress of the current game. After that the player can choose to start a new game – or 10. When the players decide to return to that game they simply have to click the new game button and then the load game button. Now they’ve essentially pressed play on the game they’d paused earlier. This is one feature that’s exclusive to the online version of the game. The real game does not have this option. This allows Player A to hold his place with Player B while playing a game with Player C. The buttons can be tested by clicking “New Game”, playing a few moves of the game, clicking “Save Game”, “New Game” and then playing as normal. When the saved game needs to be accessed, simply click “New Game” and then “Load Game”.

New Game

New game is the button at the side of the board that resets the game to the initial start position. It calls on the base functions to set up the foundation of the game, ready for one of the players to make the first move. This is important not only to initially start a game, but also to start another game at the end of the current one. Whether someone won, there was a draw, or someone managed to cheat, the new game button essentially wipes the slate clean. It is very easy to test by simply pressing the button in various stages of the game.

Random and Alternating

The current player class decides who the active player is. I.e it decides which player will be making the next move. At the very start of every game, the first player is initialized randomly. Both blue and red have an equal chance or making the first move. However, once the player is chosen and has made their move, the class also ensures that the active player is switched. This ensures that each player get to make exactly one move at a time. This is important because it is how the real game of connect four works, The players must alternatingly take turns to make a single move. In order to check that the code is doing this, simply play the game and make sure that each person makes exactly one move, before the program switches the next player to active and the current one to passive.