Assignment 1

Report of Connect X

S2106632

Yifei Wang

INFR08029 Informatics 1 - Objected Oriented Programming

February 2021

Overview

The basic control flow of the whole game ConnectX is completed through the MVC (Model, View, Controller) pattern. To implement the basic and the intermediate features of the game, I didn't use any other classes than the provided codes. Following the instructions, the game is divided into 3 parts, Basic features, Intermediate features, and Advanced features, and I try all three level of the assignment. I have created the ConnectFour as the classic game which have the size of 6x7 and 4 in a row to win. Then I improved it to ConnectX to make its size and rule variable.

Also, by having trying the advanced feature, the game now can also be saved and loaded after, even AI against actual player is provided to make it more fun and enable multiple ways to interact with users.

Basic Features

First of all, to realize the game described, we have to make a skeleton of it, and the basic features happen to be the skeleton of the game.

- Model the state of the game

To model the state of the game, we first need an empty board and two players. Therefore I first have a private variable *player***(I changed it to public after though)** which is initialized as Player1. To make a grid, I believe it should be a 2D array, so the 2D array b*oard* is initialized by 2 "for loops" and each position of the board is set as " ", a blank space. Private variable *round* is also initialized to 1(because the default 0 is not a valid round) to represent which round is it now.

- Display the board

Observing the board, there are (rows + 1) rowDividers needed and in each row, (columns + 1) columnDivider needed. Therefore, to display the board, we simply need 2 for loops just as initializing the variable board. The only difference is that we have to append a rowDivider at the end to close the board.

- Allow pieces to be played

In the real game, as we choose a certain column, the piece we drop will fall along the column till it meets the bottom or some other pieces.

- First we have to make a boolean method "isMoveValid" to validate the input. And the invalid input of the move is only when the input integer is out of the bound of the board. Plus, when certain columns is full, pieces also cannot be dropped there. Any other type of inputs than integer will be dealt by the method readIntFromUser from the given Class InputUtil.
- Next, I started working on "makeMove" to make it able to make changes to the 2D Array *board*. I first observe that when the round mod 2 is 1, Player1 is making changes, and round mod 2 is 0, Player2 is making changes; after each player makes changes, the variable player get to change. As I mentioned above, the pieces either fall to the

end or they meet anything, so we have to check if the position is blank which is able to bear a piece. Then we make loops to check if there's a space to drop the piece, if yes, then we drop the piece correspond to the player(Player1 - A, Player2 - B). The break at the end of the loop is because whenever it finds a place and drops a piece, it stops to prevent it from dropping a whole column of pieces.

- Implement the game loop

As the feature name shows, we need a loop to implement it. Here I choose a **while** loop which is because I think it's more flexible here to use **while** than use **for** since we font's exactly know when to stop the loop. To make it iterate, we simply need a boolean called game and set it to true. Then I keep displaying the board and asking the two players for inputs through the provided method in the View Class askForMove.

- Game Over

- In the basic features, game is over only when the board is full or anyone concedes. I first write a method askForConcede in the Class View just as the method askForMove. I ask for integer 1 or 2 to make it simpler than ask for any String type input. At this moment, we also need winMessage for players, so I put it in the Class View. Since we get to make the loop stops as one concedes, so I then set *game* to false to stop loop.
- The other scenario is that the board is full. I then write a boolean method *isFull* to see if every blank space is taken just as initializing the *board*. Therefore, when the board is full game is over(Boolean *game* = false).

- Complete the missing features

- To show whose turn it is, we make a method getPlayer in the Class Model and at the end of display the board, we get player and print whose turn it is.
- Validation is implemented through System.err.print() and call the method itself again to tell the users they have made a mistake and not to crash the game.

Intermediate Features

- Start the New Game

- Each time players end a game, the program will ask for restart, implemented just as *askForConcede.* And if the answer is yes, which is integer 1 in the program context, it will set *game* to true and call *startSession* again to start a new game.
- To reset the game state, and wipe the board, the *startSession* will reset public variable player and round to their default values. And a new board will be created and set as 6x7 and every entry of the grid will be blank space.

- Variable Game Settings

- When It comes to the size of the game, I first try to look at the setters. Since I have to ask the users what their desired size and the rule they want, I create some methods in View Class to ask for inputs.
- Back to Model Class, Instead of simply assign constants to the *row* and *column* and the *board*, I make several methods which require inputs to set the rows and columns.

According to the set rows and columns, then we make a blank board with the input row and column. Same method used to set rule(instead of 4, how many pieces to win).

- Enhanced Input Validation

- Now that we have the customized board and rule, there are some inputs that doable people to run the basic game.
- Therefore, I list some of the invalid scenarios. For example, It have to prevent from the board is too small to win, and the customized rule should also be big enough to make the game run normally, but at the same time, the rule should also not be too large to disallow anybody to win.

- Automatic Win Detection

- To win the game, players have several ways to get there. Therefore, to solve the problem, I use modularization to split *isWin* into several parts.
- First is *rowConnected.* Since it has to have rule-number of same pieces to win, I apply **for** loop again to check if the one being checked is the same as the several next in the same row. *ColConnected* is also done by the same way.
- Solving horizontal and vertical connected pieces, diagonal connected case is left. I modularize it again to *upperDiagonal*(from bottom left to the top right) and *downDiagonal*(from top left to bottom right). Here I found it's difficult to use 2 loops again, so I add another loop to iterate from the start we check to the following next pieces. The bound of *int i* and *int j*, are the bound that prevent the check from hitting out of the bound of the grid.

Advanced Features

- Save and Load

- Getting advice on Piazza, I searched method *FileWriter* and *FileReader* and try to use them to implement Save and Load.
- Also first I focus on part of it, *saveGame*. Since after saving users may want to resume the game, I have to write necessary data like customized n*rRows*, *nrCols, rule,* and the state of the game*, board.* Since it is only able to write String value of data, so I first change them to their String value and then write them in a text file *with* a name that users give.
- Now that we have *saveGame*, the only problem is how to read it. To load a game, I have to reset the variables the loaded game have. Recall the order we write in a file, the first three are *nrRows, nrCols,* and *rule*. Hence, to create a board, it only needs to call *setSize* and *setRule* I created in the *Variable Game Setting* feature. Final question is to reset the state of the board. In order to do it, I then write a method in the Model Class. Remember when I write out data in the file, I put every number and every single entry in different lines. Therefore, while loading the data, what I did is also load one line each time. Hence, by using for loop, we can simply assign every single state to its corresponding position from the output array of *loadGame*.

- Play Against The Computer

- For the Computer Player, it is very easy to make its move valid by limit it within the bound of the grid.
- For a more improved AI, I did some search online to see if there are some good ideas to start me off. I saw some essays about how a Minmax Algorithm works and as far as I have learned, I can't really apply it. However, I did really think about it on how to counter players' moves and make a winning move.
- If the AI wants to counters a move of a player, it has to check if the player is going to win, which is there's only one piece lack to make the player win. Therefore, it has to be able to check the whole board by the method revised from check win, to check if there are (rule - 1) pieces in a single line, and then put the piece in the position that will make the player win.
- Therefore, to find the if there is someone that is going to win, I wrote *CounterRowNext, CounterColNext*.etc to help check if the player has a (rule - 1) in-a-role line.
- In the contrast, *winningRowNext, winningColNext*.etc are there to help the AI to check if the AI has a (rule - 1) in-a-role line to help the AI make winning moves.
- To locate the exact position, I wrote *getRowRow, getColRow,* and *getCounterRow.* And these three methods are just like the booleans above, but they output integers which are the coordinates of start point of (rule-1) in-a-row line. And the getCounterRow method is also conducted by for loops to help get the exact column the lacking piece is in.
- Now that I have already completed every module of the plan, then what I next did is to put them together to help the AI to move. Hence, the *aiMove* is responsible for dropping pieces for AI. Note that in the if condition, I have several math formula like something mod 2 == 0, then …. And this is also responsible for the moving strategy. Since if the lacking piece is in the middle of the air and there's no pieces in that column yet, only when the row number is even, it will be safe to drop a piece there, or the opponent would win.
- For the rest of the time, when no one is going to win, just randomly drop the piece within the bound of the grid.

MVC Pattern

The whole program is written and completed by MVC pattern, and it roughly splits the code into 3 parts, Model, View, and Controller. The main goal of splitting them, from my comprehension, is that we could effectively implement and complete similar types of work in a single Class. For example, in the Model Class, I choose to put all the states variables and definitions of essential methods that control the game flow in it. In the View Class, I put all the work that needs input and interaction with users in it. In the Controller Class, I organized the game loop and the whole process of the game together to make the process easier to be understood.

Coding in this kind of style is really convenient for programmers to debug, since if there's anything wrong with the method, the programmers will know that they should check which Class corresponding the problems. Also, it may look more readable and clearer on what the programmer want to do. The Controller Class also is a good example of abstraction since people don't really need to know the contents of each method.

In conclusion, MVC Pattern is a brilliant style which is friendly since it has a high level of reusability, readability and easy to debug.

Evaluation

In the scope of the whole program, I think its functionality is good enough to carry out what the instructions said. The structure of the whole thing is also quite clear to make people understand what is going on. However, there are some limitations that could be improved. Looking at the s*tartSession* method in the Controller Class, I think it could be improved with more abstraction because it now only looks like combination of different method and there are also many repetitive stuff. In the Model Class, there are many methods written in similar ways, which I think it's also possible to be simplified by take the similar parts out and code them in a single method.

On the functional side, the program is just as what the instructions said, which may be somewhat boring and plain. A way to improve it could be to give users more freedom on the choice and customization.