Inf1B – Assignment 1

S1808795

1. Overview

This report will cover my thoughts and processes when designing and implementing my solution to the specified program requirements. I will discuss the design of my solution, comment on its suitability and review possible improvements. My solution contains four major components, three of which are the model, view, and controller. The model contains the state of the game, such as the pieces the player places on the board. The view facilitates the representation of the game to the user by outputting the state, in this case, to the terminal. The controller manages the control flow of the program through allowing communication between the model and view. As an Advanced feature I have also implemented an NPC class to allow the user to play against the computer. Communication between the NPC class and the model and view is also managed by the controller.

2. Basic Features

I chose to represent the game board as a 2D integer array with empty positions initialised with its default value of zero, as I thought this would allow simple evaluation of the state of the board. Such as when checking for a win, it is sufficient to loop through the board in four directions, being horizontal, vertical, positive slope diagonal and negative slope diagonal, and checking if there are the required number of player pieces in a row for a win. Though this could potentially be improved by not checking positions where it is impossible to get a win. For example, if there is a vertical row containing four zeros, there is no need to check for a vertical win above that as the values at those positions would also be zero. When the board is outputted, a column number is displayed above each column. If the number of columns exceed nine, the column numbers are no longer aligned with the corresponding column due to more space being taken up by larger numbers. This could be improved by adding spacing to certain columns so that each column has the same width.

I defined constants PLAYER_1 and PLAYER_2 with values of one and two respectively to represent the players. This means that player one's piece is represented with the integer one, and player two's the integer two. This representation meant that it was not needed to convert data types between representing a player's piece on the board and outputting the current player through view. To keep track of which player was making their turn, I implemented a switch player method which, if the current player were one, it would change it to two and vice versa. I also defined a Boolean variable hasSurrendered in model which is passed in the isGameOver method as I thought this would prevent unnecessary complication to the controller class where the isGameOver method is called many times.

To test whether a piece would be played correctly, integers negative one to eight were entered when prompted for a free column. One to seven were accepted if there were space on the board, 0 brought up the command menu as expected, and the user was prompted for a new input for negative one and eight. If the user entered a column number associated with a full column they were also prompted for a new input. To test that the board being full would be recognised, the board was filled up with pieces and the board full message was displayed as expected. When the command for surrender was inputted, the game ended with a surrender message as expected.

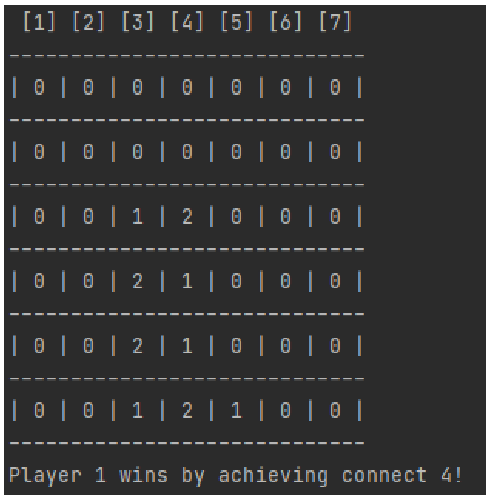
3. Intermediate Features

To allow the user to start a new game, a resetState method was implemented in model. This method sets the field values of the class back to its default constants, except for the board array where a new board is created with default zero values in each position. An alternative for resetting the board array is to loop through each position in the array and set the value to zero, this could be the slightly better or worse method depending on how efficient garbage collection is for the old board array resulting from the previous method. When testing this method, a problem was initially encountered due to not resetting the value of boolean variable hasSurrendered. I realised this was the issue by printing out the values of the state in the model constructor after the resetState method was run. When the user is prompted with whether they would like to start a new game all integer inputs give the expected output.

To allow the user to enter variable game settings, I implemented a changeGameSettings method in model. This method gets the new number of rows, columns, and pieces to connect from the view via the controller and changes the state of the corresponding fields. To ensure that the winning the game is possible, I implemented a areSettingsValid method also in model. This method checks that the number of pieces to connect is larger than 1 as its not possible to connect 1 piece or less, and that the dimensions of the board are larger or equal to the number of pieces to connect so that the game can be won horizontally, vertically and diagonally. During testing, a set of settings not adhering to these criteria were entered and an error message containing the criteria prompting the user to try again was outputted, as expected.

To make sure the game does not crash or make invalid moves, in addition to the testing mentioned above, normal, extreme and exceptional data were inputted, and expected result received, when prompted at the: make move prompt; surrender prompt; start new game prompt; and change game settings prompt. I decided to make all input required by the program to have integer data type as this allows reuse of code for input validation.

An isWinConMet method was implemented in model to incorporate automatic win detection. This method has four part, the horizontal, vertical, negative slope diagonal and positive slope diagonal checks. Each check loops through the positions on the board where a win is possible for that specific direction and returns true if it finds a winning section on the board. This method was tested by placing pieces onto a new board for each of the directions until the condition was met. Although there were no errors detected at this stage, I later captured a board state which was returning true when there were not four pieces connected:

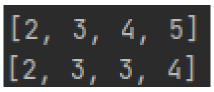


```
 [1] [2] [3] [4] [5] [6] [7]
-----------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
-----------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
-----------------------------
| 0 | 0 | 1 | 2 | 0 | 0 | 0 |
-----------------------------
| 0 | 0 | 2 | 1 | 0 | 0 | 0 |
-----------------------------
| 0 | 0 | 2 | 1 | 0 | 0 | 0 |
-----------------------------
| 0 | 0 | 1 | 2 | 1 | 0 | 0 |
-----------------------------
Player 1 wins by achieving connect 4!
```

To debug this, I created a checkID variable and for each of the directions of isWinConMet, checkID would be assigned a value corresponding to the check loop if the condition for that direction was met. This value ended up being 3 which corresponded to the negative slope diagonal check. I then created an array to store the indexes of the pieces that were returning a positive result and outputted them:

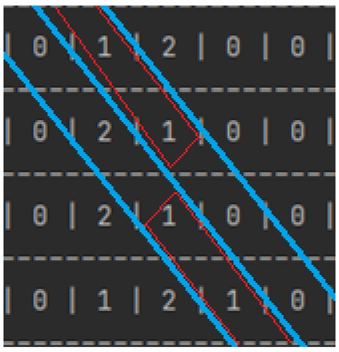
Row:

Column:



```
[2, 3, 4, 5]
[2, 3, 3, 4]
```

After illustrating the pieces that were giving a positive result in red and the four pieces the check should examine in blue, I noticed that there were four pieces connected but not in one section, which suggested that there might be an error dividing sections. This turned out to be the case as my code was not dividing the diagonal sections up often enough.



```
| 0 | 1 | 2 | 0 | 0 |
---------------------
| 0 | 2 | 1 | 0 | 0 |
---------------------
| 0 | 2 | 1 | 0 | 0 |
---------------------
| 0 | 1 | 2 | 1 | 0 |
---------------------
```

4. Advanced Features

An NPC class was created to implement an NPC that the user could play against. After doing some initial research I decided to try implement the minimax algorithm. A minimax algorithm for this game would loop through each available column where a piece can be played and calculates the score for the state of the board if a piece were to be played there. The score is calculated by evaluating individual sections of size four where the closer it is to connect four pieces within the section for one player, the higher the score and the closer it is for the other player, the lower the score. Say the first player is the maximising player, the highest score of the board state of the columns where the piece can be played is returned along with the column of the corresponding play. This process will recursively call itself for a certain value of depth, alternating between the player that chooses the highest score among the board states and the player that chooses the lowest score. One of these players will be the NPC and the other will be the user. In my attempt at implementing this algorithm I struggled with recursively calling the function as I was not certain on how to place and remove pieces as the function looped. I tried to create a copy of the board in NPC but it was shallow copying, so the board did not manage to copy the actual values. I then decided to rethink my approach and implemented an algorithm which just tried to make the next best possible move based on the score of the corresponding board states. After experimenting with different approaches, I finally managed to implement a method makeCopy to deep copy the board array. This allowed me to place pieces and evaluate their scores without affecting the actual board state. I then experimented with the values of scores to assign to specific four piece sections of the board and arrived at what I currently have. This algorithm only works for a game of connect four but could be improved to work for connect x by using for loops to go through x piece sections and evaluate their score. Another improvement could be to randomise which player goes first as currently I have set the NPC player to always go second.

To save and load the game I imported java.util.Scanner and classes from java.io. To save the game, the state of the game was first converted to a string, with each value of the state separated by a "–", this allowed them to be separated again when reading. When testing an error was discovered when the file was read first which gave a FileNotFoundException. A method was then implemented to ensure that such a file exists before trying to read it.

After completing the program, normal, extreme and exceptional data were inputted at each prompt, and expected result received.

Model View Controller

The Model View Controller (MVC) design pattern was invented by Trygve Reenskaug in the 1970’s it is very helpful for breaking large projects down into smaller more organised sections. The model contains the logic and data of the application, it has little to no interaction with the view. The view represents the user interface and is able to display the data in model. The controller contains most of the interaction between the model and view, it can process incoming requests and interact with the model based on those requests, the results of which are eventually are passed back to the view. MVC is useful for website development as it allows the data sent to client browsers to be completely separate from the data in model. There are two other similar design patterns know as Model View Presenter (MVP) and Model View View-model(MVVM) where the model and view are essentially the same as in MVC. In Model View Presenter, the presenter not only interacts with the model but also formats the data obtained to be displayed in the view. In Model View View-model, view-model allows for a clearer separation of the development of the user interface from the logic and data in the model, and view-model essentially converts the data from model to what is required to be represented by the view.

5. Evaluation

I think assignment has been a good learning opportunity as it helped me gain a better understanding of the Model View Controller design pattern and a deeper understanding of java. In my program I think that most of the code is splits into small manageable functions which allows the task in hand to be followed clearly. Some of these smaller functions also provided good reusability, such at the input validation function for commands. Though if I had more time I would try to improve on the NPC as after some testing there were some situations where it did not pick what I believe was the best next move. I would have liked to finish implementing the minimax algorithm after having found a way to make a deep copy of the board array. Furthermore, I think the code could have been written better to follow the MVC design pattern as at some points I was unsure of where to put certain methods such as input validation methods.