220002145 kw215 11/11/2022

CS1002 WEEK 9 PRACTICAL

Overview:

The practical required us to build a fully functional copy of the game Connect Four, featuring a command line, text based user interface. The program needs to prompt the user to input the number of the column they wish to play as an integer, and then display the board, containing all the moves that have been played so far. I have completed all parts of the specification, with my program even supporting further functionality beyond the specification, as it runs a generalised form of the game, with few hard-coded features.

Design:

My implementation contains two classes, “Board” and “Game”, both situated in the “gameplay_elements” package. ”Board” is responsible for the methods and variables used for the construction and display of the board itself. “Game” makes use of a board object to instantiate the actual gameplay; it performs the win-checks, switches the player when required, detects invalid input, and runs the main gameplay loop. The main (interesting) features of each class are explained below:

Board

Notable variables*:*
boardArray is the 2D array storing information about the board, containing information about each column in the array. I have chosen this variable type as it most intuitively represents a 2D rectangular board, and makes accessing and modifying entries relatively straightforward.
columnCounter is a 1D array representing how many moves have been played into each column. This is used in various methods involved in playing a given move and checking whether the current player has won.
Notable methods*:*
initialiseBoard is the method responsible for filling out **boardArray** with “.” characters, so that it can be displayed. It simply loops through every element, and sets it to “.”
playMove takes as a parameter the column that the user has chosen to play, and using the corresponding information from **columnCounter**, it places the move appropriately in **boardArray.**
displayBoard works by looping through each “line” of **boardArray**; It starts by selecting the index corresponding to the first line of the array, and printing each element in the line, repeating this operation for all lines.

Game

Notable variables:
currentBoard is the instance of **Board**, used to facilitate gameplay. It is used to make use of the **Board** methods and variables.
currentMove is the variable storing the last move made by the player. It is used in nearly all **Game** methods, for obvious reasons.
Notable methods:
checkWin is the method used to check if the win condition required has been fulfilled. It performs straightforward horizontal and vertical checks, storing all values present in the current row and column of the last move played. The diagonal checks are somewhat more complicated, explained under Fig 1.Finally, all the stored values are now part of the string variable **letters**. If it contains a sequence of 4 letters that are the same, the winner is declared and the game ends. If there is no more empty space on the board, the game is a draw and it ends.

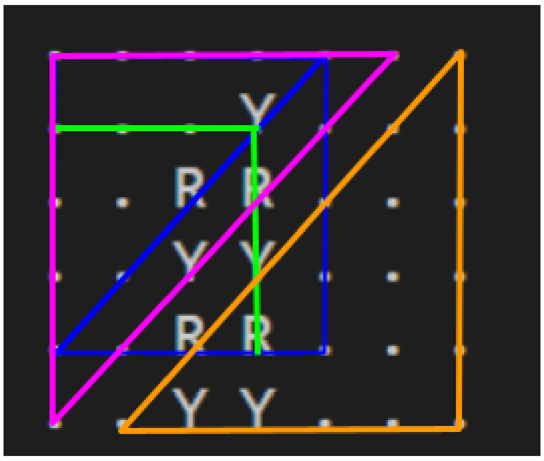

Fig 1

In this example, notice how for each diagonal, there is a corresponding square. The task of finding the start of the diagonal then becomes the task of finding the bottom row of the square corresponding to this diagonal. Notice that for all possible moves satisfying that the distance between the move and the leftmost side of the board is less than the distance between the move and the bottom side of the board, the diagonals start on the first column (purple triangle encloses all of these points). Notice further that for all moves in this subset of the board, the distance between the move and the leftmost side is equal to the distance between the move and the row on which the desired diagonal starts. Therefore, the starting row is given by subtracting the row of the move from the column of the move, as shown by the green lines. The diagonals in the orange triangle, as well as the ones going the other way are calculated in a similar fashion.

playGame, inputMove, and **validateMove** contain some input checks as well as calling the necessary methods for the gameplay loop. I noticed that during my testing, it was very inconvenient to accidentally input a letter, making the program throw an error and end, so I added additional functionality that solves this problem, placing invalid input errors under the “Illegal column” input message.

Design considerations

I have attempted to design the program in classic object oriented fashion, while trying to make the code as general as possible. For example, by modifying only the values for width and height with which the board is generated, it is possible to play the game on a different sized board. Further, it is easy to make it so the game becomes “Connect N” for any reasonable number N, with very minimal modifications to the code, or even increase the number of players, as the checking method supports this functionality. One design consideration which was mostly subjective is when the game should declare a draw - and on this I have found it most appropriate to only declare a draw when all of the spaces available on the board are filled. This is because the user has the option to exit the game at any time, and therefore if the user spots that the game is a draw they can exit; otherwise, unless the board is full, there is still plenty of gameplay to be had: the point of the program is not mathematical precision in determining when the game is a draw, but rather maximising the enjoyment of a potential user.

Testing:

Stacscheck:

```
Testing CS1002 W09 Practical
- Looking for submission in a directory called 'source': found at 'W09Practical/source'
* BUILD TEST - build-all : pass
* COMPARISON TEST - 1_public/prog-run-01_immediate_quit.out : pass
* COMPARISON TEST - 1_public/prog-run-02_single_move.out : pass
* COMPARISON TEST - 1_public/prog-run-03_vertical_play.out : pass
* COMPARISON TEST - 1_public/prog-run-04_illegal_column.out : pass
* COMPARISON TEST - 1_public/prog-run-05_full_column.out : pass
* COMPARISON TEST - 1_public/prog-run-06_yellow_horizontal_win.out : pass
* COMPARISON TEST - 1_public/prog-run-07_red_vertical_win.out : pass
8 out of 8 tests passed
```

Test	In	Out	Pass(Y/N)
Letter test	a	Illegal column	Y
Letter test	A.,,.b3123	Illegal column	Y
Column test	-20	Illegal column	Y
Win checks	N/A	Correct output	Y
Quit test	0; 00	Game over: user exit	Y
Decimals test	3.14159	Illegal column	Y

Conclusion:

My program fulfils all specification points with little room for user induced errors. It passes all tests supplied by me and stacscheck, and I believe I have provided tests for all reasonable use cases. Given more time, refactoring the code and improving the efficiency of the checkWin algorithms would be a reasonable direction to take the project in.