SET11102 – Software Development 1

Coursework 1

40331195

Contents

Class Diagram

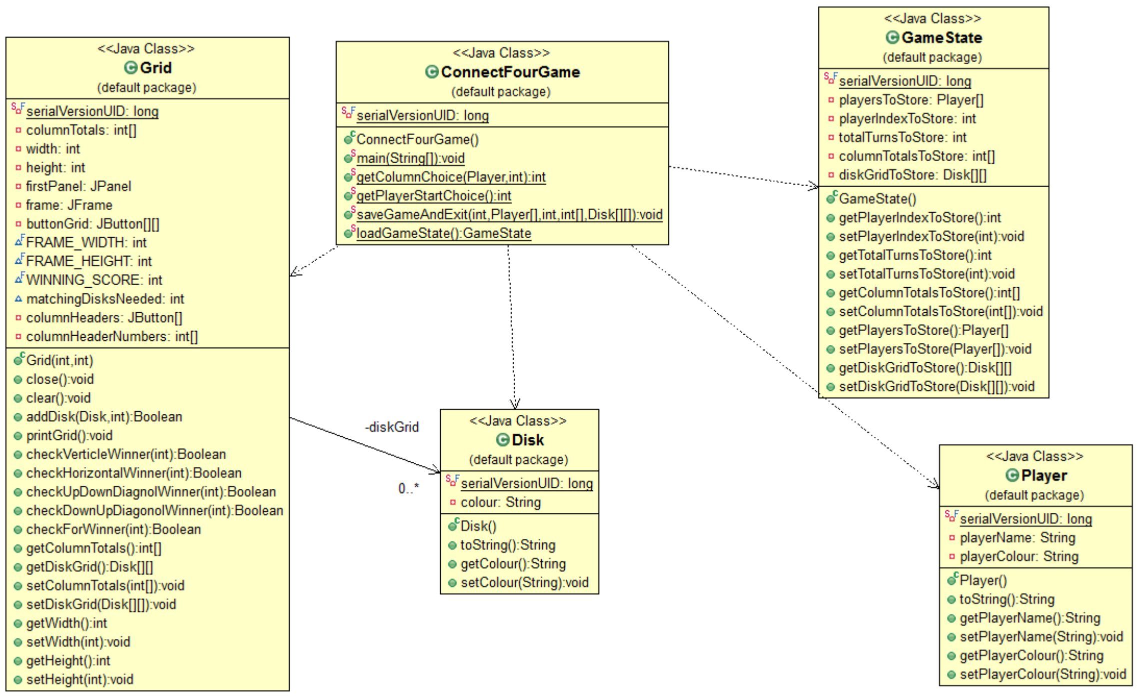

Figure 1. Class Diagram for Connect 4 Game.

Report

The game of connect four which has been created is represented in the Class diagram above (Figure 1). The game is made up of Disk objects which are inserted into an empty 2d Array Grid of datatype Disks. The game alternates turns between two Player objects which are created when the game commences.

On start up the game commences by asking the player to select between a New Game, Load Game and Quit. The new game will initialise by creating an empty 2d array of the grid.

The dimensions are set at 6 counter column height and 7 counter row length. The winning score to achieve is set at 4 in a row. It has been attempted to create the game in such a way that the changing of these 3 variables to other numbers would still allow the game still function. However further work would be required to fully achieve this.

The grid is displayed to the user as a 2d array of buttons contained within a JFrame. The method to display the grid iterates over the grid and represents each empty slot as a white button, each yellow player counter as a yellow button and each red player button as a red counter. These buttons do not have an action listener and are only used to visual represent the grid to the user. A row of buttons which are not linked to the game grid has been placed above each column. They contain the column number so that the player can see which column is which more easily.

The grid class handles most of the functionality that occurs when the game is played, such as the methods for displaying the grid, checking for victory and adding the disk to a column. These are all called on the instance of the grid class is created which is given the name 'theboard' .

The chosen column to place the turn's disk is selected by the user from a JOptionPane panel. They may select from columns 1 to 7, there is also an exit game button should they wish to quit mid game. This will then prompt whether they'd like to save the game before taking them back to the main menu.

Each turn a disk will be created with the colour property set to the colour the player has selected, which is stored as string type. It will then be placed into the column the player chooses. The column height variable is initialised as 6, so each time a player attempts to place a disk, the game checks whether there has already been 6 disks placed in this column. This is done by using an array named columnTotals, which keeps track of the total disks placed in each column. If the columnTotals array at the index of the column, contains 5 or less disks, it may enter the next available slot and the method will return true to confirm its placement. However if the columnTotal value is at 6 then the disk will not be placed and the method will return false. This will then instigate a loop prompting the user to choose a new column, until they select one that is not full. This will then return true and the loop will end.

An attempt has been made to ensure as few integers as possible have been left inside the game code, however a few 1s have been kept as they represent the difference between the array positions starting at 0 and the counts starting at 1.

To check whether a player has won the match by connecting 4 disks in a row, it was necessary to create 4 methods. The first and simplest method checked for a vertical win and returned the Boolean True if there was a win. This first of all involved finding how many disks were already in the column, if there were less than 4 then false was returned. If there were more than 4 disks, then the position of the last disk placed was found and a for loop iterated over the 3 disks below to see if they matched the last one placed. If any were found they were added to a counter variable and if they did not match the loop was broken. At the end if the counter was at 4 or more, then the game had been won so true was returned. The counter was started at 1 instead of 0 as the calculations did not include the disk which had just been placed, so just a further 3 matches were required.

The next check to be performed checked for horizontal victory. This required checking the chips both to the right and left of the disk placed. This was done by using 2 for loops checking up to the 3 disks to the right and then to the left, with a shared counter. They would first check whether the slot about to be examined existed on the board, if it was out of the grid's limits, the loop would break to not crash the program. Once this had been established, they would check whether there was a disk in the slot, if there was not, then the loop would also break. Lastly they would check the grid position for a disk that did not match the player's colour, if one did not match then then loop would also break. If the loop had not broken by this point then the disk would belong to the player and the counter would be incremented. After both loops had ran, if the counter had been incremented from 1 to 4 then the game had been won.

The last checks implemented involved checking for diagonal victories. This involved two separate checks, one for diagonals going upwards from left to right and another for diagonals going downwards from right to left. Similarly to the horizontal checks, each one of these checks involved 2 loops with a shared counter. Each loop iterates its way diagonally from the disk that was placed, breaking the loop if it is either about to check the edge of the board, check an empty slot, or it finds a disk that does not match the players colour. If the loop is not broken, then a matching disk has been found and one of the counters is incremented. This was also slightly more complicated than the horizontal check as it was necessary to guard against the loops checking non-existent grid references above and below the board as well as to the left and the right. Again if either of the counters for the two loops reached 4 then the game had been won.

In order to store save game data, a 'Gamestate' class was created. This class contained its own version of the properties necessary to recreate the game exactly where it had left off. The variables necessary to store were the two player objects, the 2d grid object, the column totals array, the number of turns that had been taken and lastly, the player index to ensure the correct player would take their turn on loading the game. In the same game method, an instance of the Gamestate class was created and these variables were passed into their corresponding properties within this object. The Gamestate object, was then written to a binary file which would store it.

When Load game is then selected, a new instance of the Gamestate object was created and the binary file was loaded into it, so it would contain the properties of the saved game. This Gamestate object was then passed back to the main game loop where its properties were passed back into their corresponding properties in the game. This resulted in being to continue a game from where it had been saved.