**Software Development 1**

**Report**

**Contents Page**

1. My Design …………………………………………………………………………….3

2. How I Represented and Displayed the Grid……………………………………….5

3. How I Checked for 4 Disks in a Row ………………………………………………7

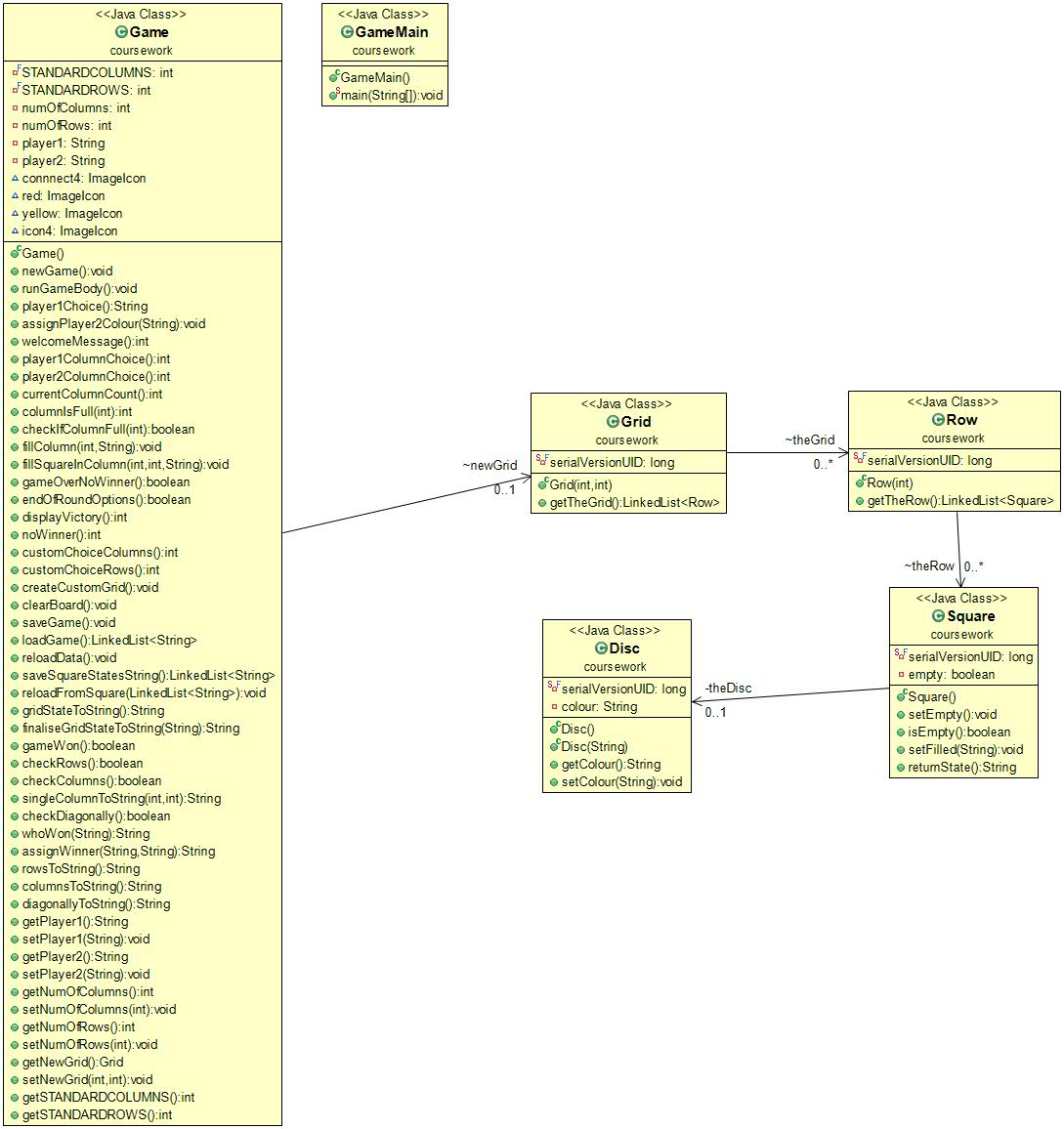
4. How I Prevented the Players Adding Discs to a Full Column …………….…….8

5. Advanced Features ………………………………………………………………....9

5.1 Save and Load ……………………………………………………………..9

5.2 Diagonal Victory …………………………………………………………....9

5.3 Custom Grid(additional) …………………………………………………...10

**1. My Design:**

**Figure1: *Class Diagram***

I chose to complete this coursework using completely object-oriented programming, rather than a 2D array, as I wanted to choose the more challenging route, and create a more robust program.

Referring to the above class diagram, we can see that the Grid class is made up of a LinkedList of the Row class, which in turn is made up of a LinkedList of the Square class, which can be filled with an instance of the Disk class.

I have a separate Game class which contains an instance of the Grid class, which is manipulated throughout the program in line with the game logic.

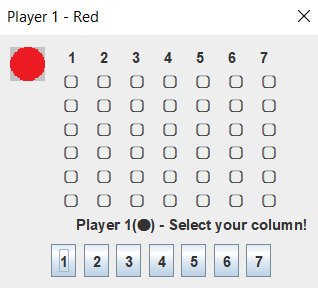
The individual classes that include and make up the Grid class have clean, minimal code and the vast bulk of the program methods are contained in the Game class.

This was to correspond with best practice programming methodologies, and ensure that the game logic was as maintainable and scalable as possible.

The Grid and game logic is also designed to not only allow for a standard game of connect 4 that meets the project brief, but also to allow for a “Custom Game” option to be selected by the player, which allows for any size of grid to be generated. Any size of custom grid can also be played with the exact same functionality as the standard game, as all code was generated with scalability in mind.

The motivation behind this was to best demonstrate the robustness and scalability of object-oriented programming, and also to demonstrate my understanding of the programming techniques we have been shown throughout the semester.

**2. How I Represented and Displayed the Grid.**



**Figure2: *Player Select*** **Figure3: *End of Round***

I represented the grid using special Unicode characters on a tailored JOptionPane window – see Figure2.

The chosen colours of red and yellow were displayed by specific Unicode characters that created a uniform and easily distinguishable interface.

From a code point of view, I created a method in my Square class that would return the state of an individual square object in string form – either “0”, “Y”, or “R” to represent empty, a yellow disk, or a red disk.

In my Game class I created a method that iterated (via nested enhanced for loops) over each square in the grid in sequence and called the ReturnState() method on each. A series of If statements then converted the returned character into a specified Unicode character and added them all into one string with appropriate spacing and new lines between them. Additionally, a line of integers, based on how many columns there are, is generated and listed at the top of the grid in order to help guide the players column choice.

I also created a method that would populate the options on the JOptionPane with as many buttons as there were columns, to offer the user a scalable and easy interface.

At the end of each round the user is presented with the option of playing another round or saving and exiting - see Figure3.

This was decided on because it offered a much neater and more intuitive interface than having additional, mis-aligned buttons next to the column choices.

Additionally, custom icons and player-specific text and titles are presented inside the JOptionPanes – these are programmed to be accurate no matter which colour choice players make.

**3. How I Checked For 4 Disks in a Row**

I created two methods in my game class to check for 4 disks in a row either horizontally or vertically.

The horizontal method iterated row by row over each square in the grid using nest for loops and called the ReturnState() method on each one. The returned String values where then combined into one string with a “.” Break to signify the end of a row. The method then simply checked if “RRRR” or “YYYY” appears in the string anywhere and returns a corresponding Boolean value.

The vertical method was similar to the above except it was coded to check the state of each square in the grid over a series of columns. This was achieved by using an enhanced for loop inside of a normal for loop and utilising the counter to access the correct row and square indexes.

Later, each method was separated into two methods as part of this functionality was required for other methods in the programme and it further tidied up the code overall.

**4.** **How I Prevented the Players Adding Discs to a Full Column**

I created a function that would iterate over the first row in the grid which is always the top row of the grid, regardless of size. I would use this function to check if the top square of a chosen column was empty, if it was, the game logic would then fill the chosen column with another disk.

If it was not empty, the game logic would then alert the player that the column was full and, using a while loop, would prompt them for an alternative column choice (which was also checked to see if that was full), until they entered a column that has free squares.

**5. Advanced Features**

I completed all of the advanced features.

**5.1 Save and Load:**

I achieved the save and load functionality through serialisation. When the save game function is chosen, the grid state is saved in string form, along with the player1 colour choice, and the current column and row count (in case of custom game).

This was all saved in order in one LinkedList<String> and, once loaded, the player1 colour choice, column count, and row count, are removed from the LinkedList using RemoveFirst() in sequence and utilised to update a new Game instance using dedicated methods. The main method that should be mentioned is a method that iterates over each square in the newGrid and changes the square’s state based on what String character represented that Square’s state during the save.

Originally, I attempted to save the states of the Grid or squares directly as objects but I continued to run into errors until I simplified the save process using one LinkedList of Strings.

**5.2 Diagonal Victory:**

This was achieved using two tailored nested for loops, inside an overarching for loop inside the method to check for this win condition.

The first nested loop consists of two nested normal for loops that use an incrementing and a decrementing counter to check diagonals to the right and, like the above horizontal and vertical win conditions, saved the state of the squares in string form with “.” break points. This was designed to scale based on the current number of columns and rows in the initialised grid, to ensure scalability.

The second nest loop is similar but utilises the counters differently in order to check diagonals to the left.

The overarching for loop triggers each of the above nested loops for every row in the grid, to ensure full scalability no matter what size of grid was initialised.

After the full diagonal square states are all saved in one string, a check similar to above was run to see if “RRRR” or “YYYY” appeared.

**5.3 Custom Grid(additional):**

This is described above but was designed as an optional advanced feature in order to showcase the overall program design and coding best practice.

The custom grid option is also validated to not allow a user to enter a value below 1 for the column or row count. The buttons presented to the player are also fully scalable and will always match the column count chosen. It should also be mentioned again that no matter the size of the custom grid, all of the program functionality still applied in the same way as a standard game.