Final design report

Statement of completion

The extensible and adaptable gameboard was required for the assignment so I decided to develop Gomoku in the framework. I have mentioned below the requirements and description for implemented features.

Game options: program asks the user to select the game they want to play.

Player Information: here program will ask for player name and their character which will appear on board as player's disc. A player can insert anything as their character except 'C' and '- 'as these symbols are occupied by the program. The program will ask accordingly if the player decided to play with the computer or another player.

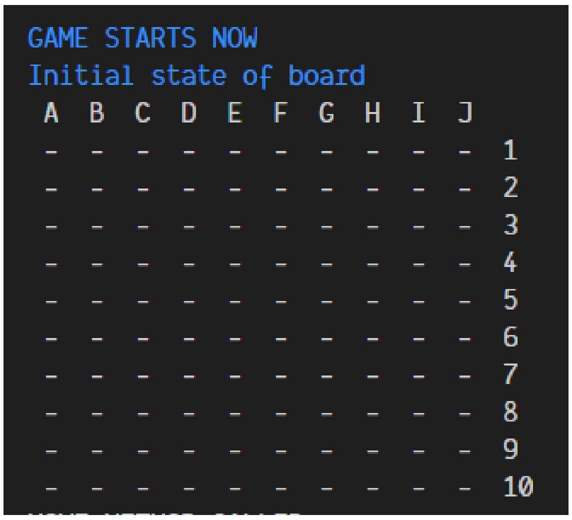
Play with human/computer: the program has both the functionality implemented within the framework.

Move information: once the game is started the program will ask to enter Row as alphabet then column as a number. If provided cell location does not exist user will get an error as Invalid move. Refer to figure 1 to understand how to identify rows and columns. Alphabets on top represent the column. Numbers on, left side indicates rows.

Save/load data: with each move user will have an option if they want to save the game. After saving, the program will prompt message and will stop the further execution. with running the program again the program will ask to restore game if a saved game exists.

Different modes(easy/hard): I have not developed this feature. Though the current algorithm for computer moves tries to win and defend very efficiently at the same time.

Undo/redo: I have not implemented the feature because I was running out of time. Though there is full functionality inside the game to extend this as there is a history of moves stored in the program.



```
GAME STARTS NOW
Initial state of board
 A  B  C  D  E  F  G  H  I  J
 -  -  -  -  -  -  -  -  -  -  1
 -  -  -  -  -  -  -  -  -  -  2
 -  -  -  -  -  -  -  -  -  -  3
 -  -  -  -  -  -  -  -  -  -  4
 -  -  -  -  -  -  -  -  -  -  5
 -  -  -  -  -  -  -  -  -  -  6
 -  -  -  -  -  -  -  -  -  -  7
 -  -  -  -  -  -  -  -  -  -  8
 -  -  -  -  -  -  -  -  -  -  9
 -  -  -  -  -  -  -  -  -  -  10
```

Figure 1Gameboard

Overview of design

While the implementation of the program I was not required to make huge changes from my initial presented class diagram. In the proposed diagram I had introduced 8 classes and 1 interface. The current implementation contains 7 classes and 0 interfaces. Where proposed classes were Player class, Gameboard class, Cell class, GameHistory class, Othello class, Gomoku class, Rivers class and Final Game class responsible for the execution of the main method. The current game is capable of extending game board size, increase /decrease players and increment in the sequence of winning. E.g. just with 1 change in code players will require 6 consecutive moves to win.

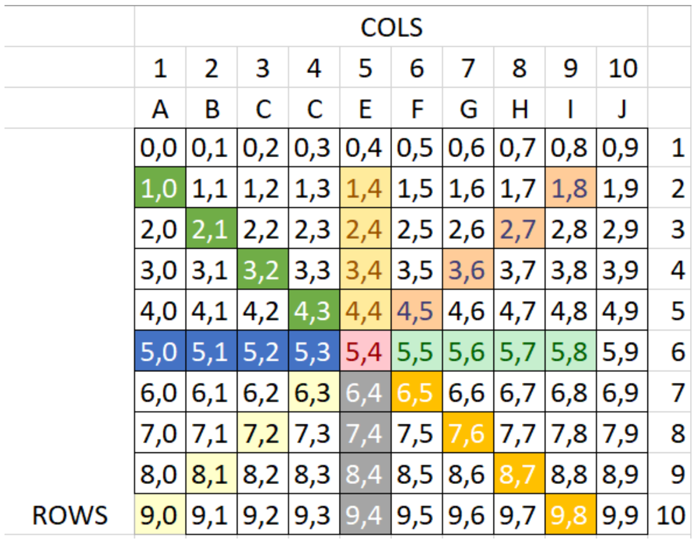
Changes I have made.

I have removed the **GameHistory** class as it was one of the limitations I proposed in the previous class diagram. The reason behind removing was in my design Gameboard class is responsible for making moves and maintaining the history and specific game like Gomoku or riversi extends the gameboard class. So it was not possible to save the Gameboard object and restore the gameboard and child class from the saved data. It is achievable but then I have to make significant changes in my proposed flow. I preferred to stick with my proposed my flow and I developed the game save and load methods in the GameBoard class.

I removed the **ComputerMoves** interface as it was not necessary for the current program. And actual computer move algorithm is developed inside GameBoard class where all the data lies for the program

There are some minor changes in the methods and parameters of many classes. Because I think developing diagrams and developing code are different tasks. There were some changes that I had to made to meet the requirement.

Algorithm for Computer moves and score calculation

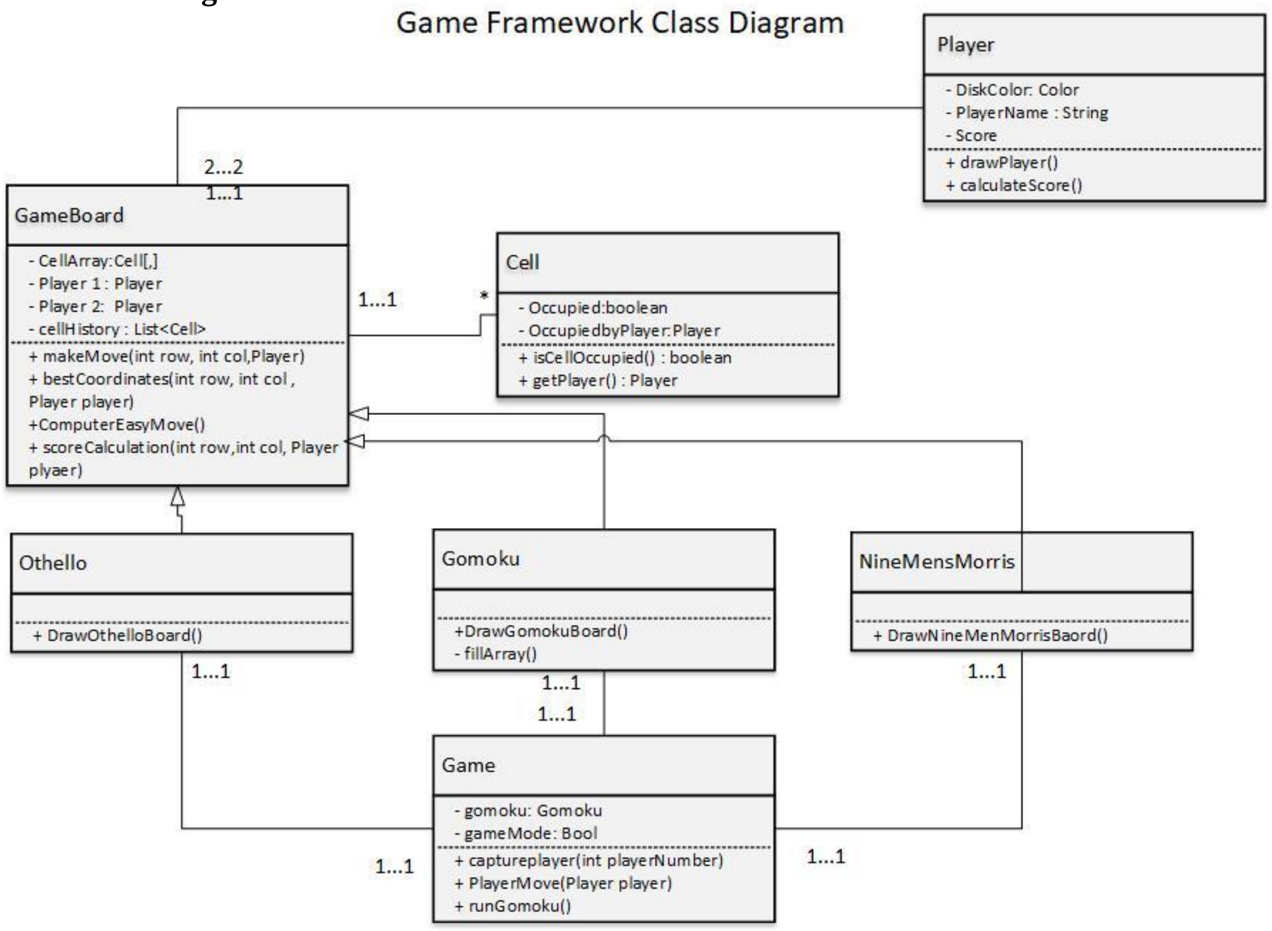


	COLS										
	1	2	3	4	5	6	7	8	9	10	
	A	B	C	C	E	F	G	H	I	J	
	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1
	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	2
	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	3
	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	4
	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	5
	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	6
	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	7
	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	8
	8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	9
ROWS	9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	10

Figure 2 Logical representation of Gameboard

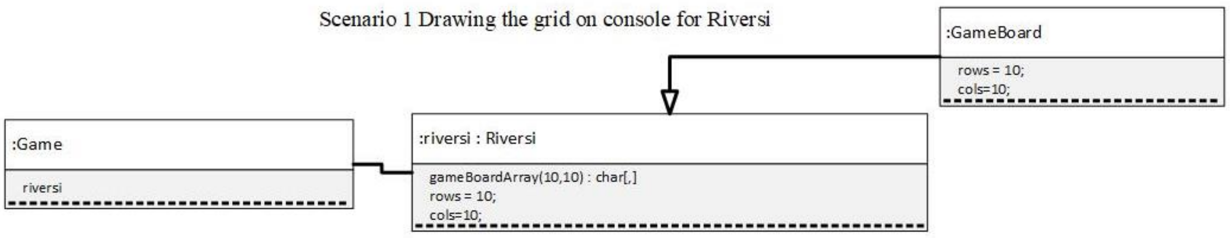
So in the current program, this is how the computer identifies the best possible move and this is the pattern used for next best move calculation and score calculation. E.g. when the player makes move on Column E and row 6(red cell) all the highlighted cells will be calculated to check if is there any sequence of 5. The same pattern is used for computer moves to check all the possibilities from the last human-made move and next possible computer move. If the computer detects a higher probability of human winning computer will occupy the next human winning position. If not computer will try to create its sequence of 5

Current class diagram

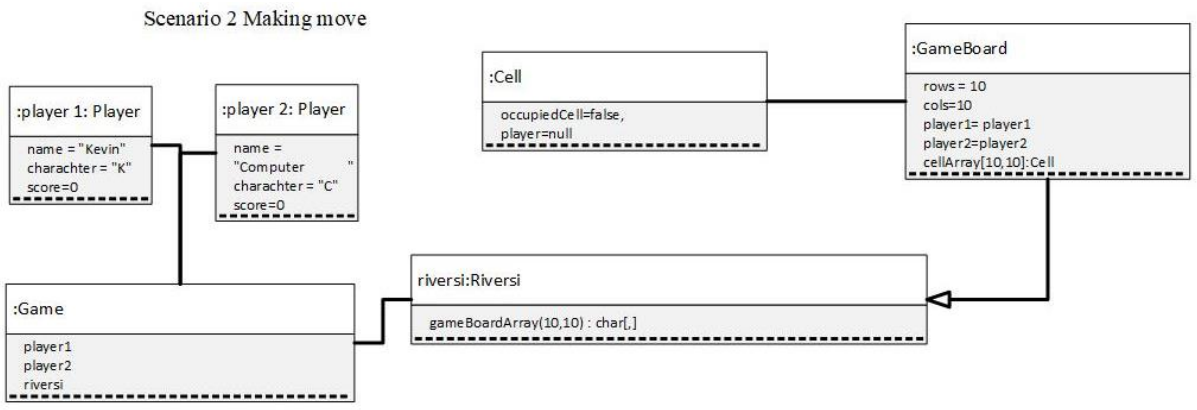


The class diagram only mentioned important methods developed inside classes. Getter setter properties are not mentioned inside the class diagram.

Object Diagram


Scenario 1 Drawing the grid on console for Riversi
:GameBoard
rows = 10;
cols=10;
:Game
riversi
:riversi : Riversi
gameBoardArray(10,10) : char[,]
rows = 10;
cols=10;

There are no changes in the flow of data so this diagram will display data in each object required to draw the grid of 10 X 10

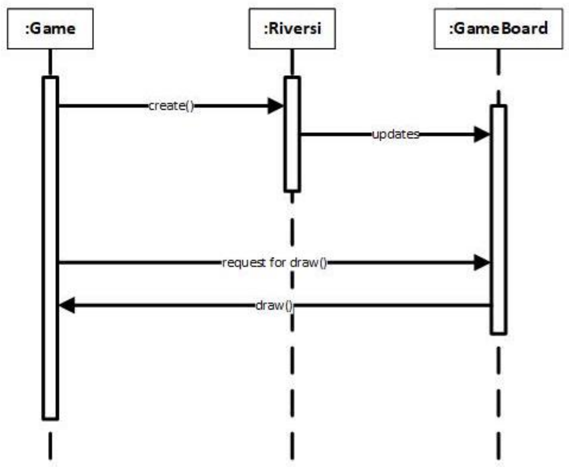


The mentioned diagram shows the data that each object will need to make a move

Sequence diagram

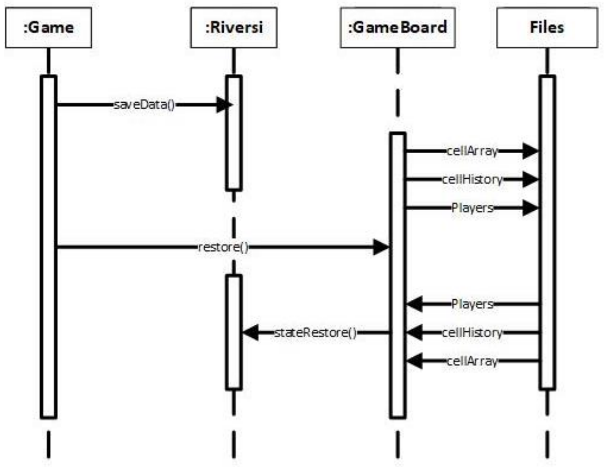
The diagram shows method calling when there is a request made to draw a board or update the board

Scenario 1 Drawing the grid on console for Riversi



The diagram shows method calling from objects when human requests to save the game. All the necessary attributes from Game board array are copied into files. When the restore method is called all these saved data is invoked and assigned to game board object class so all the properties get the values of their previous state.

Scenario 2 Saving/loading Game



Design principles and patterns used

The single responsibility principle (SRP)

According to SRP, the class can have only one reason to be changed. In this program, the whole board is made by a 2-dimensional array of 100 Cells. This whole array is made of the Cell class. Cell class plays a crucial role to identify if the cell is occupied or not, and if yes the cell also contains the information of the player who has occupied the cell. The cell class will change its state in under only under one condition when a player has made a valid move.

The player class also follows the SRP principle as it stores the information of players of the game. It is responsible for storing the player’s name and its character role of the player class is quite subtle as it is called only when a player is created and in other class, the same object is referred to a different class.

Don't Repeat Yourself (DRY) Principle

The Gomoku class is an example here as it extends the Gameboard class and inherits all the properties of parent class so when there is an update on gameboard instead of replicating or retrieving the objects of parent class Gomoku class can directly access these properties.

How to execute the program

The code is developed in visual studio code with .net library so to open it in visual studio please start Visual studio and select open project or solution. navigate to the location where you have downloaded the game code. Inside the game directory, you will see "**Game.csproj**" select and open it.

Hit run and you will see console asking for user inputs. **The program is case sensitive so please enter data accordingly**.

1. Press 1 to execute the Gomoku game
2. The program will ask you to select an option if you want to play with Human or computer select accordingly
3. If you select Computer the program will ask for human player's name. enter the name and hit enter.
4. The program will ask for a character for the player. Enter anything like A, B,% or $ this character will represent your position on game board.
5. Step 3 and 4 can be displayed 2 times if you have selected to play with another player.
6. Confirm your details with pressing Y.
7. Now you will see an empty board like mentioned in image 1.
8. Select 1 to make a move or 2 to save data
9. If you hit 1, the program will display the player's name and will ask to enter a column where you want to make a move. Columns are displayed on top of the board (A to J). Enter column and hit enter
10. Now the program will ask for a row to make a move, rows are represented on right side of the board in numeric digits 1 to 10. Enter row and hit enter. You will see your character on the specified location on the board. If you enter an invalid location the program will give you an error and will ask for the input again
11. If you are playing with the computer, the computer will make a move by itself. If you are playing with a human player the program will prompt same again for the second player's move.
12. If you hit option 2 to save data game will be saved and the program will be terminated. Now run again select Gomoku and program will ask you, there is saved data from the previous game do you want to continue? Select accordingly.
13. Once any of the players will make 5 consecutive moves program will show the winners name and terminate itself. The completed game will be not saved.

How to reuse classes

Gameboard class contains all the necessary data extended with proper methods can be helpful to develop any game logic. Although all the different games will have different playing logic and winning logics so these methods needed to be developed inside the respective game class.

Draw board logics are pretty dynamic you can generate any size of the board with it. It just has to be same rows as columns e.g. 5X5 or 7X7 or 9X9. The code is pretty strong and can handle different types of run time exceptions. To change the winning sequence you just have to change winning score in gameboard and you can change the game for let’s say 7 or 6 consecutive moves.

The mentioned algorithm for checking the score and making the move is pretty robust and can be easily modified/used for a different type of games like Othello where all logic lies in rows and columns.