Student ID: n11232862

Student Name: Phan Thao Nhi Nguyen

Email: n11232862@qut.edu.au

This assignment is my ***individual assignment***

IFN563 – Assignment 2

1. Declare the requirements

This assignment is designed for reused in board game with two dimensional, with two players. Because this is implemented by individual, I only implement for Tic Tac Toe game. But this project can be easy to extend for other board game.

a. Features is implemented
Tic Tac Toe Game
_ A reused game for board games with two players (Reversi, Tic Tac Toe…).
_ 2 players with mode: Computer move is random move, Human player is validated move
+ Human vs Human
+ Computer vs Human
_ This game can be loaded from any position after load from a log file. And user can choose to save their game or not.
_All user's move can be undoable and redoable (the full history of moves is tracked). Undo and redo are available after new moves are made
_ This game provides a Primitive Online Help: game information, valid moves, how to use advance tasks (Menu, Undo, Redo, Quit, Save game).

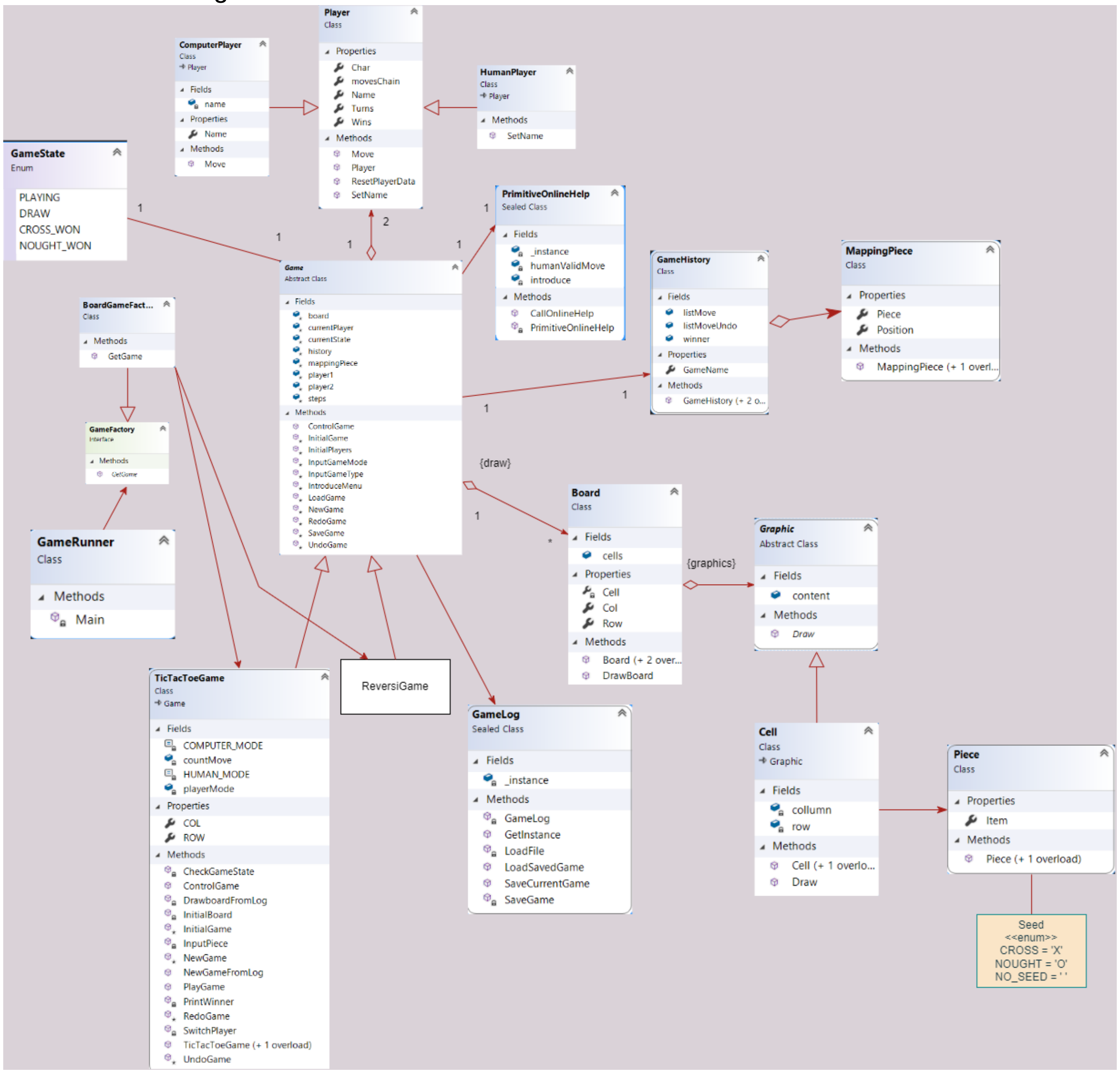
b. Features haven't implemented
_ Cannot choose Computer play mode: hard, easy.
_ Haven't implemented for Reversi game logic, UI (individual assignment)
_ User cannot choose directory, filename for saving game

2. Overall design

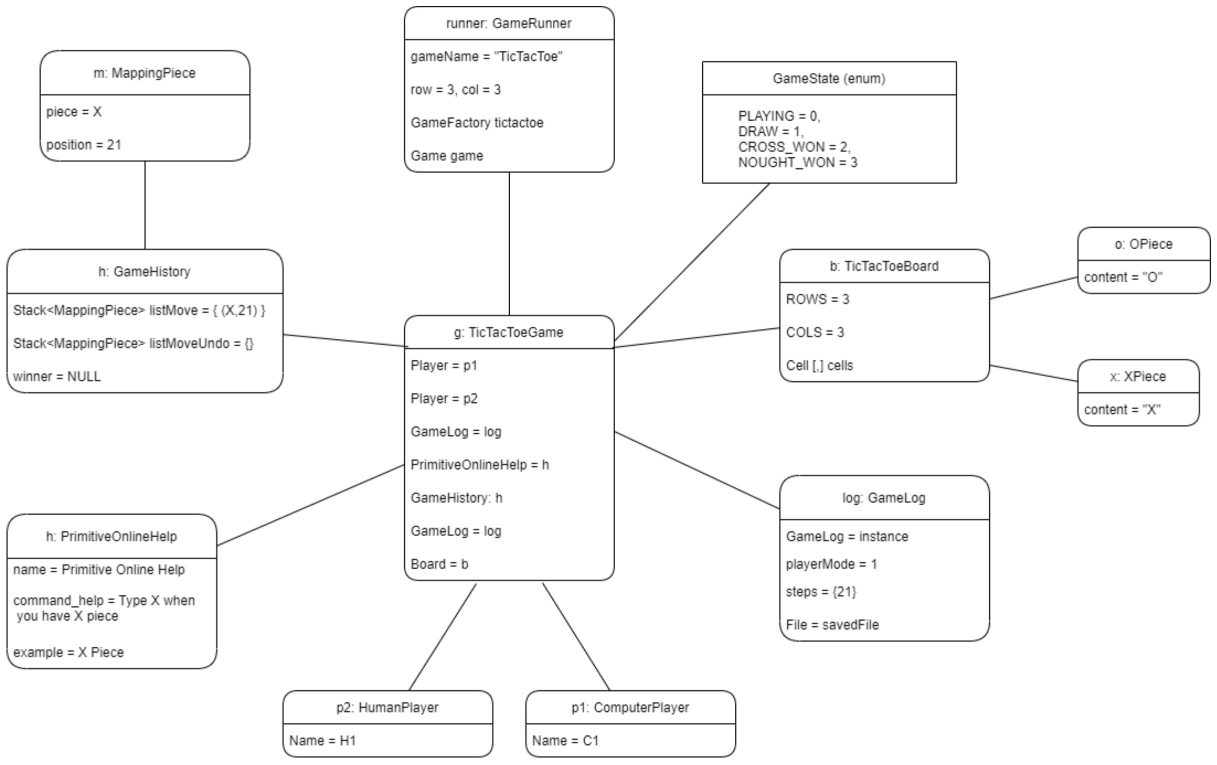
This program is designed with GameRunner which is called first to start board game. With the purpose extendable, this program is designed with abstraction idea about a Game that has skeleton for board game. TicTacToeGame or ReversiGame class should extend form this abstract game class. Secondly, in the future this program can be easily extended for any board game, so it needs a way to create different game so Abstract factory design pattern is implemented in this stage which is an improvement from preliminary design. Additionally, Players can be Human or Computer which shares some actions and have their own specific actions (Computer player have random move). Moreover, currently this Board's game is square with row and column size, but in the future, it can be triangle or square. From that idea, composite design pattern is used with Graphic class between Board and specific shape and Developer can be easier to extend. This game has history for track at any stage, so this history is only one instance through game.
Comparing with previous preliminary design, this final design has 2 majority changes. Firstly, Abstract factory design pattern is applied to select a board game from program. By this design, Abstract factory interface declares a set of methods that GameRunner or client code produce different type of board game UI. The client code does not need to rely on concrete classes and UI components when working with these objects via abstract interfaces. This also allows client code to handle any more factories or UI components that need to be implemented in the future.

3. Class diagram, Object diagram, Sequence Diagram

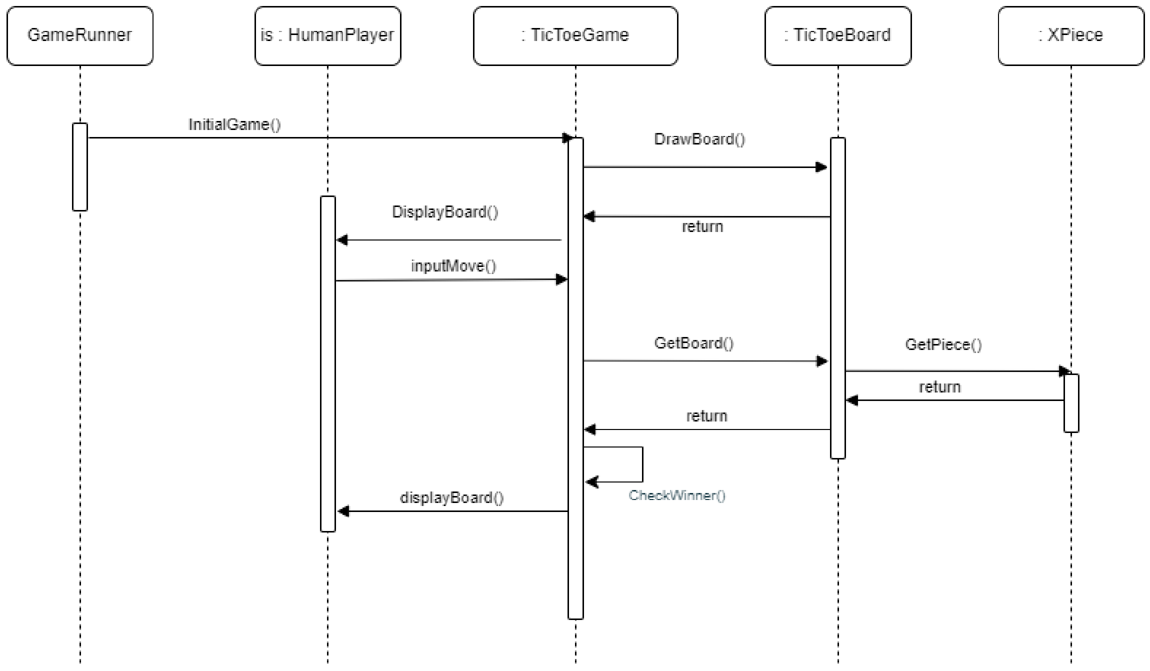
a. Class diagram



b. Object diagram



c. Sequence diagram



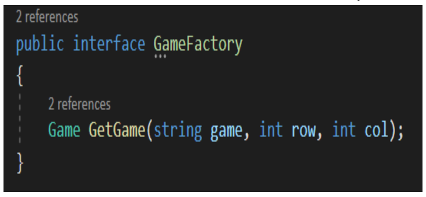
4. Design Pattern and Design Principles

a. Applying Design Principles
SOLID design pattern

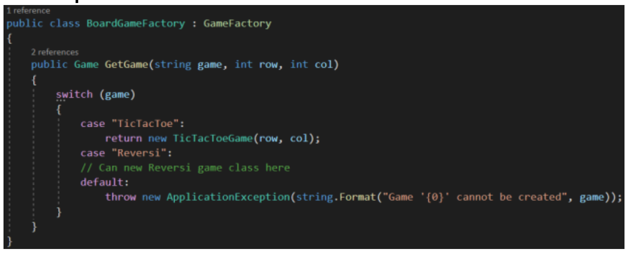
- ***Single Responsibility principle***: every class should have one and only reason for modifying
- ***Open-Closed Principle***: open for extension with class's behaviour can be extended and close for modification

This project is designed with interface and abstract class that can easily to extend with specific requirement (For example: Game…)
Show screen shot Game, TicTacToeGame

- ***Liskov Substitution***: if class A is a subtype of class B, we should be able to replace B with A without disrupting the behaviour of our program.
  For example: class TicTacToeGame is a subtype of class Game, class TicTacToeGame should be able to replace Game without disruption to our program. In the future, when implement ReversiGame class, developer should follow this principle.
- ***Interface Segregation***: larger interface should be split into smaller ones. Interfaces in this program is designed with consider about this principle
- ***Dependency Inversion***: high-level modules do not depend on low-level modules, both will depend on abstraction



```
2 references
public interface GameFactory
{
    2 references
    Game GetGame(string game, int row, int col);
}
```

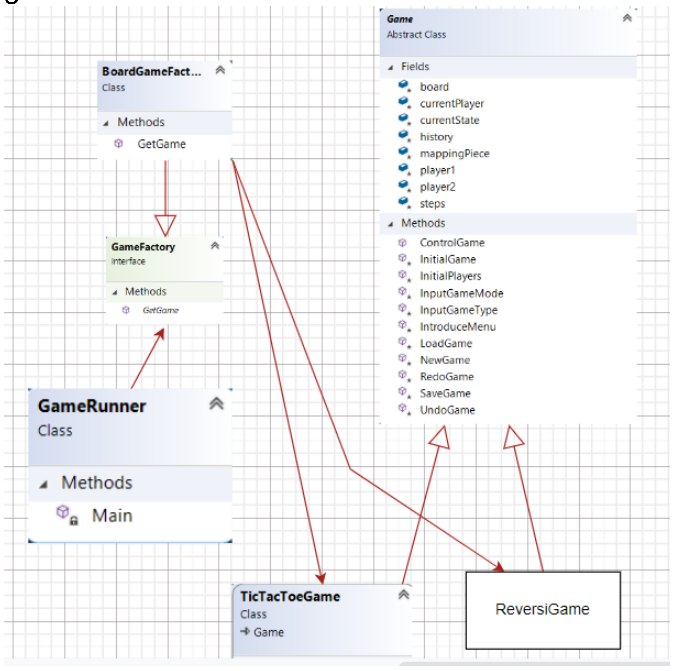


```
1 reference
public class BoardGameFactory : GameFactory
{
    2 references
    public Game GetGame(string game, int row, int col)
    {
        switch (game)
        {
            case "TicTacToe":
                return new TicTacToeGame(row, col);
            case "Reversi":
            // Can new Reversi game class here
            default:
                throw new ApplicationException(string.Format("Game '{0}' cannot be created", game));
        }
    }
}
```

b. Design Pattern

i. Abstract Design Pattern

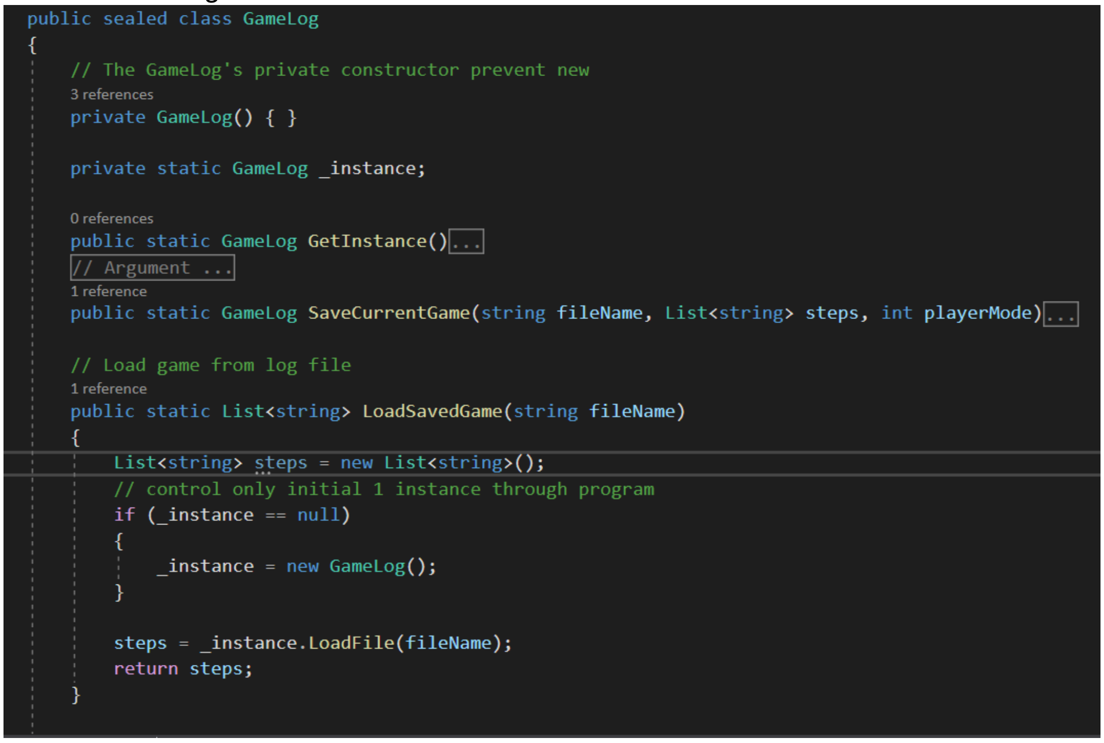
This program can be easily extended for any board game, so it needs a way to create different game (Reversi, TicTacToe).
This program has GameFactory interface that can create abstract games.
BoardGameFactory declares to create concrete game objects
Game is abstract class to declare interface for all board games.
TicTacToeGame implements the Game abstract class for all board game's behaviours.



Abstract factory DP screenshot

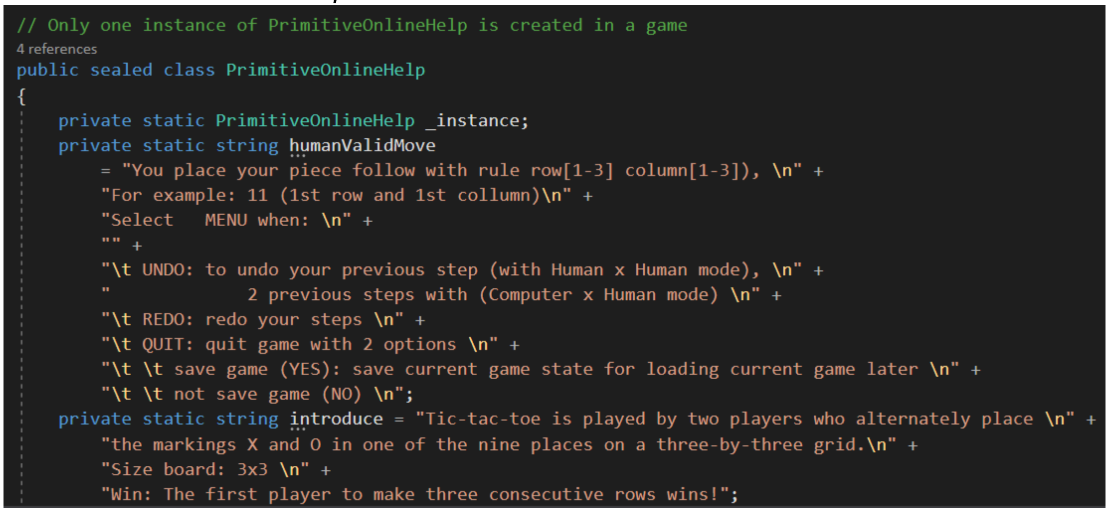
ii. Singleton Design Pattern
A class have only an instance in system: PrimitiveOnlineHelp, GameLog
Based on the requirements about PrimitiveOnlineHelp, this class has only one instance during game.
GameLog should have only instance during game. This GameLog will be created an instance when user choose save current game.

GameLog screenshot

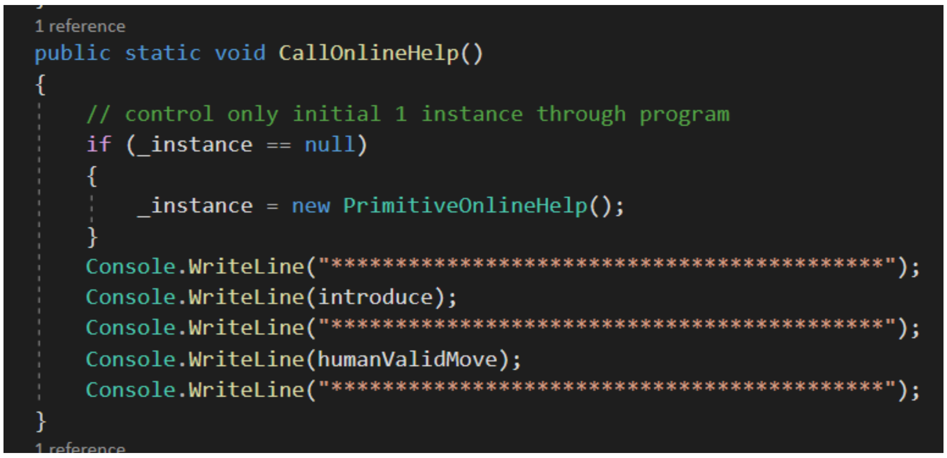


```
public sealed class GameLog
{
    // The GameLog's private constructor prevent new
    3 references
    private GameLog() { }

    private static GameLog _instance;

    0 references
    public static GameLog GetInstance()...
    // Argument ...
    1 reference
    public static GameLog SaveCurrentGame(string fileName, List<string> steps, int playerMode)...

    // Load game from log file
    1 reference
    public static List<string> LoadSavedGame(string fileName)
    {
        List<string> steps = new List<string>();
        // control only initial 1 instance through program
        if (_instance == null)
        {
            _instance = new GameLog();
        }

        steps = _instance.LoadFile(fileName);
        return steps;
    }
```

PrimitiveOnlineHelp



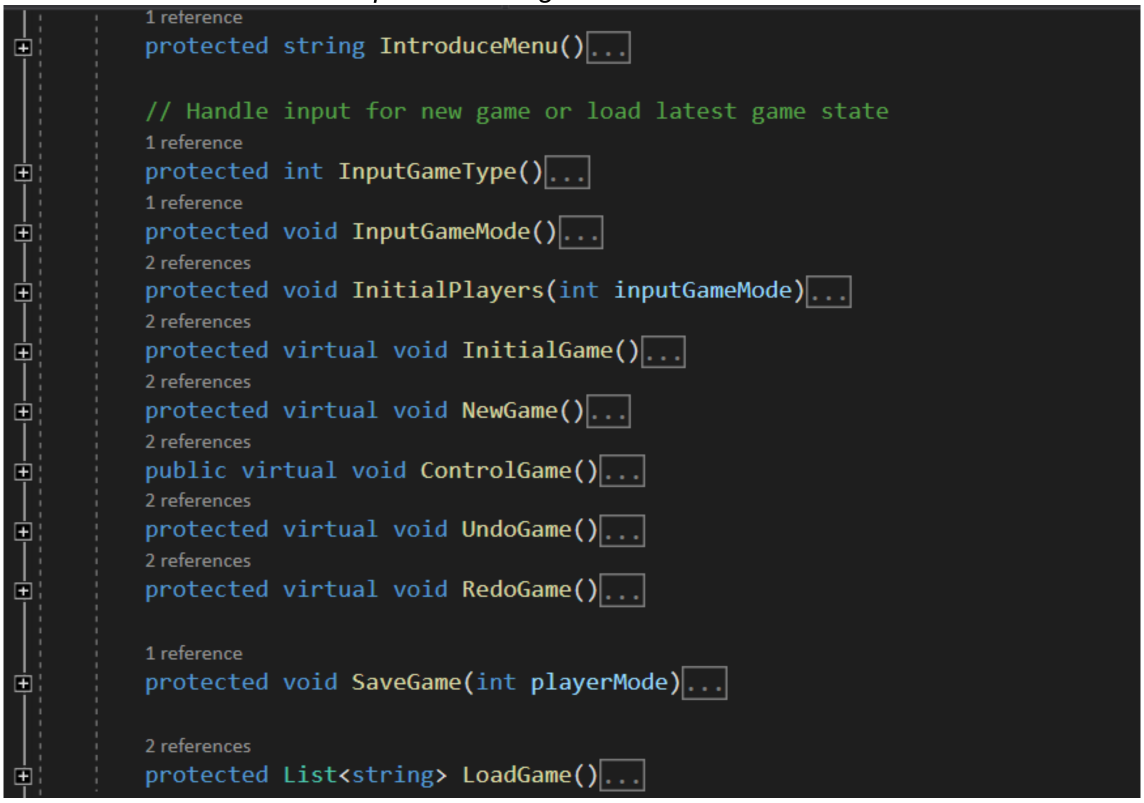
```
// Only one instance of PrimitiveOnlineHelp is created in a game
4 references
public sealed class PrimitiveOnlineHelp
{
    private static PrimitiveOnlineHelp _instance;
    private static string humanValidMove
        = "You place your piece follow with rule row[1-3] column[1-3]), \n" +
        "For example: 11 (1st row and 1st collumn)\n" +
        "Select   MENU when: \n" +
        "" +
        "\t UNDO: to undo your previous step (with Human x Human mode), \n" +
        "              2 previous steps with (Computer x Human mode) \n" +
        "\t REDO: redo your steps \n" +
        "\t QUIT: quit game with 2 options \n" +
        "\t \t save game (YES): save current game state for loading current game later \n" +
        "\t \t not save game (NO) \n";
    private static string introduce = "Tic-tac-toe is played by two players who alternately place \n" +
        "the markings X and O in one of the nine places on a three-by-three grid.\n" +
        "Size board: 3x3 \n" +
        "Win: The first player to make three consecutive rows wins!";
```



```
1 reference
public static void CallOnlineHelp()
{
    // control only initial 1 instance through program
    if (_instance == null)
    {
        _instance = new PrimitiveOnlineHelp();
    }
    Console.WriteLine("********************************************");
    Console.WriteLine(introduce);
    Console.WriteLine("********************************************");
    Console.WriteLine(humanValidMove);
    Console.WriteLine("********************************************");
}
1 reference
```

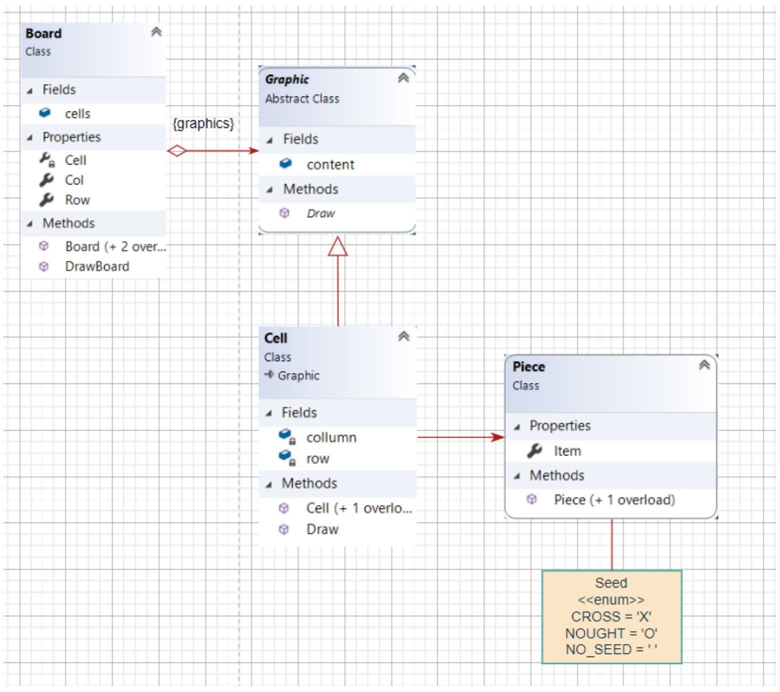
iii. Template Design Pattern
With this template design pattern helps this program defines the skeleton for board game.
Game is abstract class to declare skeleton for all board games.
TicTacToeGame class can extend and redefine for certain steps.

Game class with steps for board game

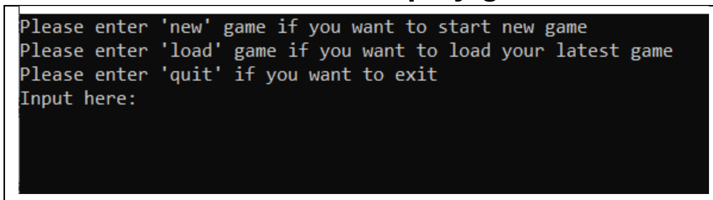


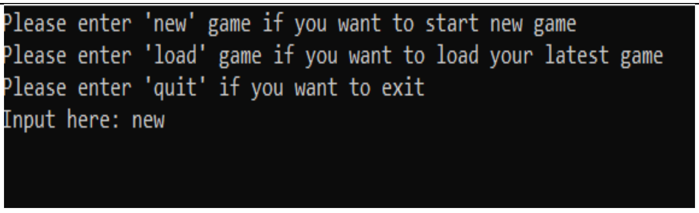
```
1 reference
protected string IntroduceMenu()[...]

// Handle input for new game or load latest game state
1 reference
protected int InputGameType()[...]
1 reference
protected void InputGameMode()[...]
2 references
protected void InitialPlayers(int inputGameMode)[...]
2 references
protected virtual void InitialGame()[...]
2 references
protected virtual void NewGame()[...]
2 references
public virtual void ControlGame()[...]
2 references
protected virtual void UndoGame()[...]
2 references
protected virtual void RedoGame()[...]

1 reference
protected void SaveGame(int playerMode)[...]

2 references
protected List<string> LoadGame()[...]
```

iv. Composite Design Pattern
Graphic Component is an object in a composition provides common interface and shared implementation for Board Composite. Cell class with Draw() funciton implements behaviour for primitive class

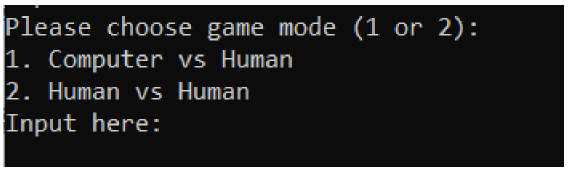


5. Introduction about how to play game

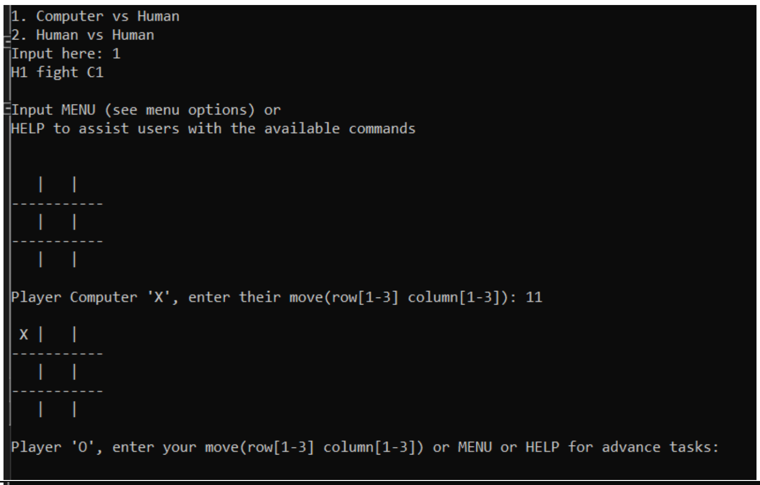
 `Please enter 'new' game if you want to start new game` `Please enter 'load' game if you want to load your latest game` `Please enter 'quit' if you want to exit` `Input here:`	Input before start game 1. **new**: start a game 2. **load**: load a saved game 3. **quit**: quit the program



```
Please enter 'new' game if you want to start new game
Please enter 'load' game if you want to load your latest game
Please enter 'quit' if you want to exit
Input here: new
```

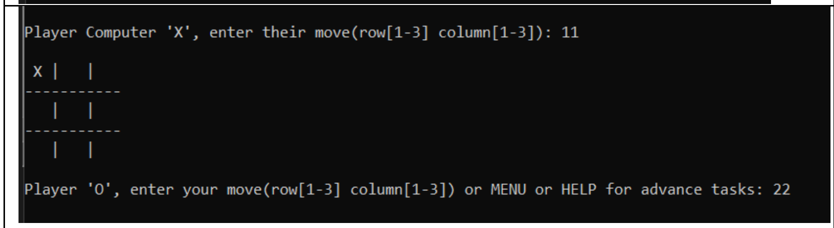


```
Please choose game mode (1 or 2):
1. Computer vs Human
2. Human vs Human
Input here:
```

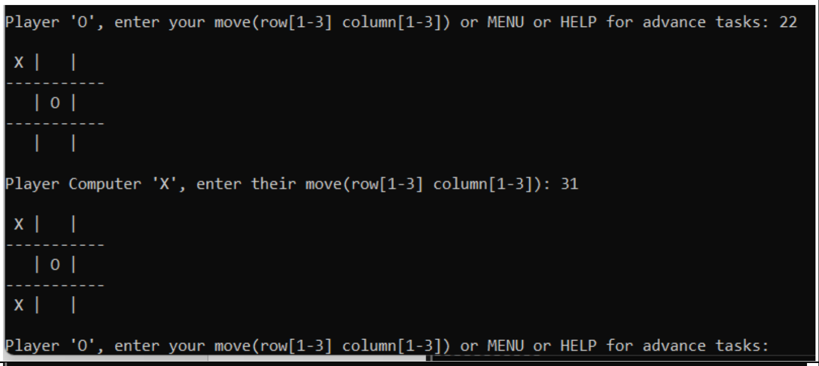


```
1. Computer vs Human
2. Human vs Human
Input here: 1
H1 fight C1

Input MENU (see menu options) or
HELP to assist users with the available commands


   |   |
-----------
   |   |
-----------
   |   |

Player Computer 'X', enter their move(row[1-3] column[1-3]): 11

 X |   |
-----------
   |   |
-----------
   |   |

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks:
```

Start a game with
(1) Computer and Human mode

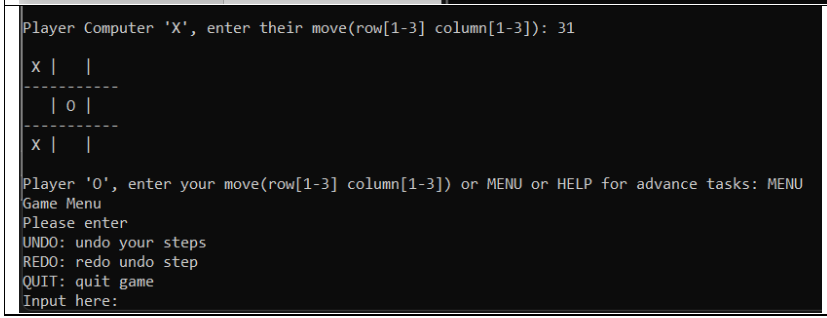


```
Player Computer 'X', enter their move(row[1-3] column[1-3]): 11

 X |   |
-----------
   |   |
-----------
   |   |

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: 22
```



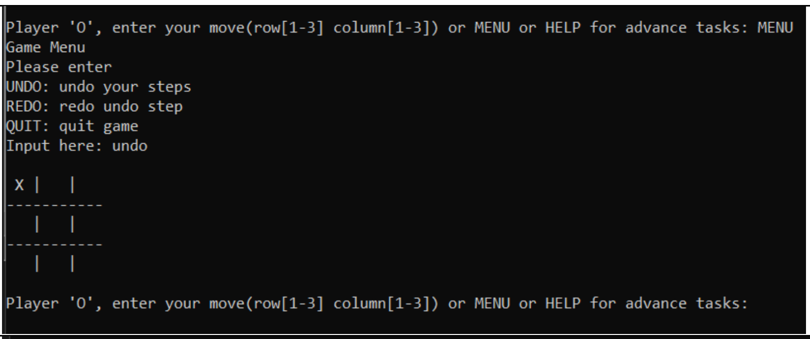
```
Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: 22

 X |   |
-----------
   | O |
-----------
   |   |

Player Computer 'X', enter their move(row[1-3] column[1-3]): 31

 X |   |
-----------
   | O |
-----------
 X |   |

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks:
```

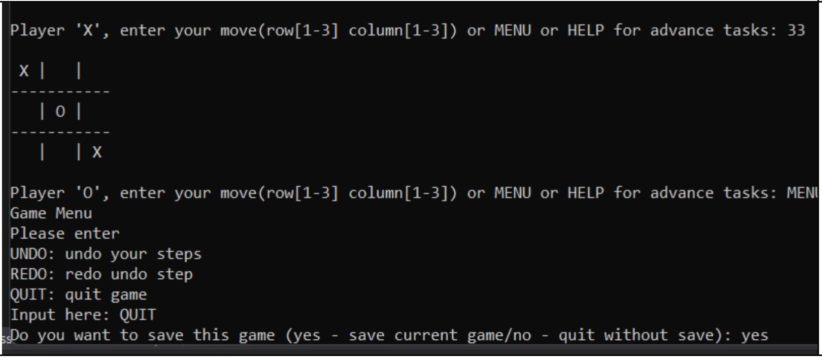
Computer with Piece X as default
Human with Piece O.
Your move must follow the rule:
_ 1st digit: row position (count from 1)
_ 2nd digit: column position (count from 1)



```
Player Computer 'X', enter their move(row[1-3] column[1-3]): 31

 X |   |
-----------
   | O |
-----------
 X |   |

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: MENU
Game Menu
Please enter
UNDO: undo your steps
REDO: redo undo step
QUIT: quit game
Input here:
```

Menu option:
_ Please input **MENU** to see list of menu items (including: **undo**, **redo**, **quit** (with **save** game or not)


```
Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: MENU
Game Menu
Please enter
UNDO: undo your steps
REDO: redo undo step
QUIT: quit game
Input here: undo

 X |   |
-----------
   |   |
-----------
   |   |

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks:
```
	For example: with **undo** item


```
Player 'X', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: 33

 X |   |
-----------
   | O |
-----------
   |   | X

Player 'O', enter your move(row[1-3] column[1-3]) or MENU or HELP for advance tasks: MENU
Game Menu
Please enter
UNDO: undo your steps
REDO: redo undo step
QUIT: quit game
Input here: QUIT
Do you want to save this game (yes - save current game/no - quit without save): yes
```
	To save current game.
	Please input **QUIT** and yes to save game
	(input **no**: game will exit without save game)

6. **Class/Interfaces to be reused from existing libraries and frameworks**
_ Collections: Game (List), GameHistory (Stack), GameLog (list), TicTacToeGame (List)
_ IO: GameLog (File),
_ System.Text: GameLog (StringBuilder)