Overview of Final Design

Framework Structure Explanation

In the framework, both concrete classes and abstract classes exist, where the abstract classes will be **inherited and overridden** by different board games where necessary.

The framework is designed to maximize the "responsibility separation", meaning that each class is designed to only perform task that is relevant to itself. Also, to make the project scalable, the classes are categorized into **Logic** and **UserInterface**.

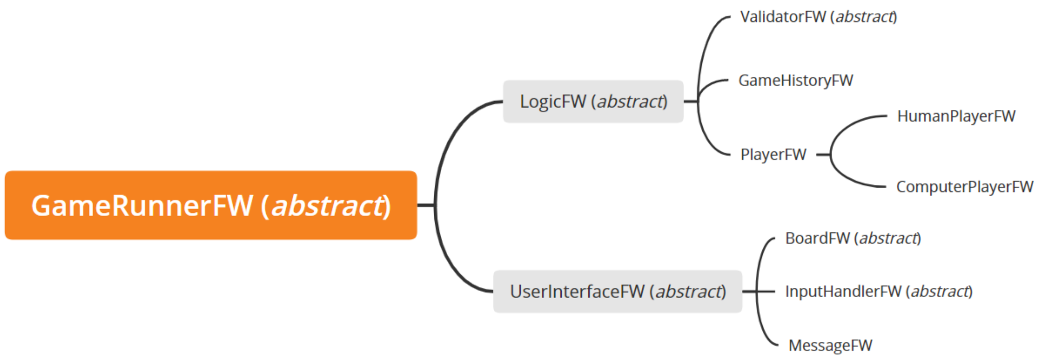
For classes that perform tasks related to the user interface, such as **BoardFW** to display the game board, or **InputHandlerFW** to prompt user input, exist as class objects in **UserInterfaceFW**.

For classes that perform tasks related to the logic part of the game, such as **GameHistoryFW** to keep track of the game history, or **ValidatorFW** to validate the moves, exist class objects in **LogicFW**.

GameRunnerFW serves as the medium to collaborate all classes under **LogicFW** and **UserInterfaceFW**.

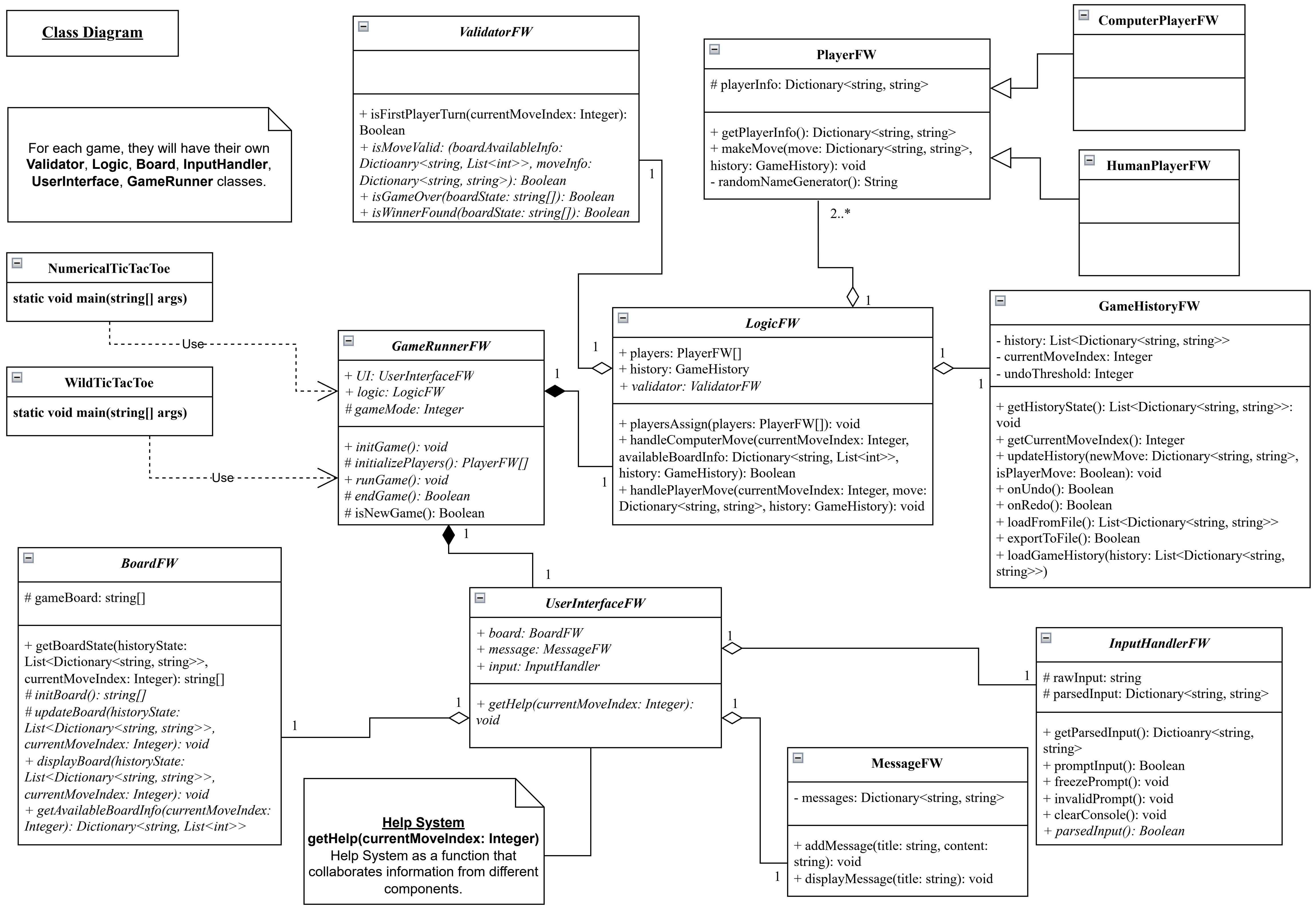
With this, when relevant functions are called, they can be called through via the class objects with clear separation. For example, **logic.validator.validateMove()** or **UI.board.displayBoard()**.

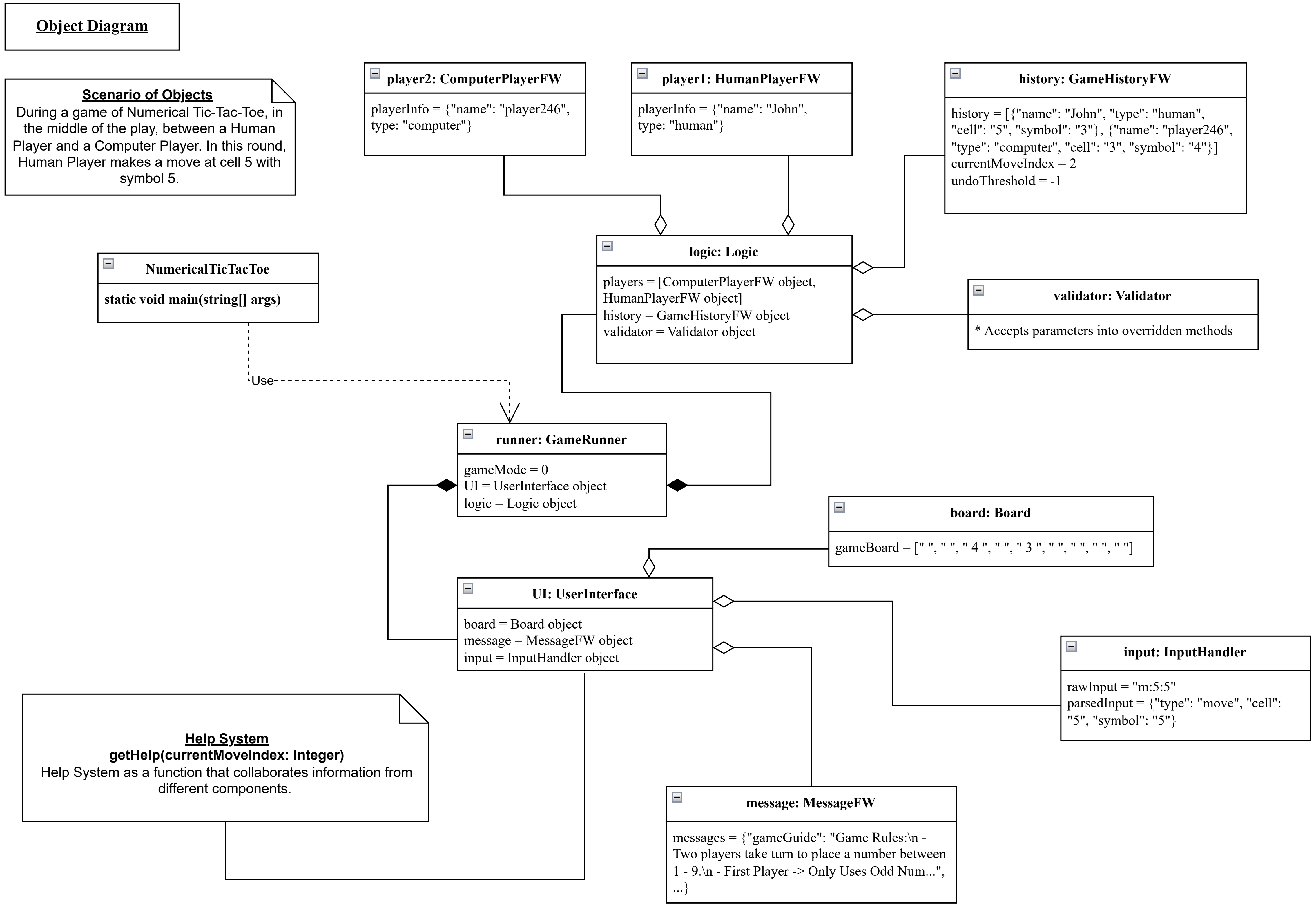
The figure below shows the brief structure of the entire framework.



Changes comparing to Preliminary Design

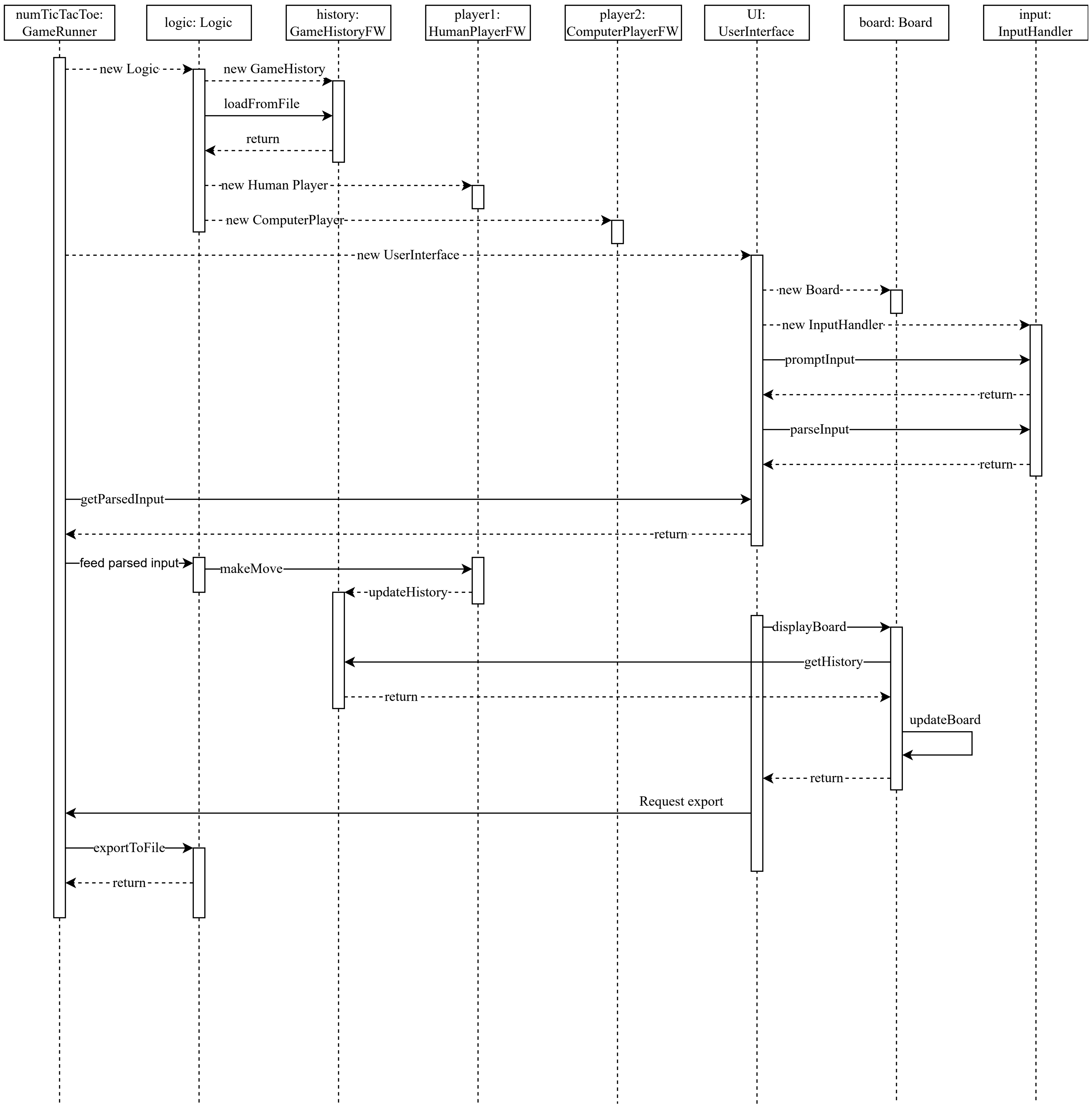
1. **LogicFW** class added – In the previous design, classes related to the logic part of the game were not put together. **LogicFW** class is added for the ease of maintenance.
2. **InputHandlerFW** class – In the previous design, the user input part was separated into 1. RawUserInput 2. InputParser. However, this is redundant and makes the design more complicated. Therefore, features from both classes are combined to one.
3. **GameHistoryFW** class – In the previous design, GameHistory only keeps track of the game history, and the export and the load functions are performed by GameFileHandler. However, GameFileHandler has not attributes, and require the game history to work, therefore, combining them together simplifies the design.
4. **Help System** becomes **getHelp()** that lives under **UserInterfaceFW** – Originally, help system itself was a class. However, after implementing it to the game, help system requires information from **different classes** in UI. Therefore, it will be easier to design the help system as a function under user interface.
5. To make it more reusable, the classes that were originally concrete or an interface, and are now abstract are: **ValidatorFW, BoardFW, UserInterfaceFW.**


Class Diagram
For each game, they will have their own Validator, Logic, Board, InputHandler, UserInterface, GameRunner classes.
ValidatorFW
+ isFirstPlayerTurn(currentMoveIndex: Integer): Boolean
+ isMoveValid: (boardAvailableInfo: Dictioanry<string, List<int>>, moveInfo: Dictionary<string, string>): Boolean
+ isGameOver(boardState: string[]): Boolean
+ isWinnerFound(boardState: string[]): Boolean
PlayerFW
# playerInfo: Dictionary<string, string>
+ getPlayerInfo(): Dictionary<string, string>
+ makeMove(move: Dictionary<string, string>, history: GameHistory): void
- randomNameGenerator(): String
ComputerPlayerFW
HumanPlayerFW
2..*
1
NumericalTicTacToe
static void main(string[] args)
WildTicTacToe
static void main(string[] args)
Use
Use
GameRunnerFW
+ UI: UserInterfaceFW
+ logic: LogicFW
# gameMode: Integer
+ initGame(): void
# initializePlayers(): PlayerFW[]
+ runGame(): void
# endGame(): Boolean
# isNewGame(): Boolean
1
1
1
1
LogicFW
+ players: PlayerFW[]
+ history: GameHistory
+ validator: ValidatorFW
+ playersAssign(players: PlayerFW[]): void
+ handleComputerMove(currentMoveIndex: Integer, availableBoardInfo: Dictionary<string, List<int>>, history: GameHistory): Boolean
+ handlePlayerMove(currentMoveIndex: Integer, move: Dictionary<string, string>, history: GameHistory): void
1
1
GameHistoryFW
- history: List<Dictionary<string, string>>
- currentMoveIndex: Integer
- undoThreshold: Integer
+ getHistoryState(): List<Dictionary<string, string>>: void
+ getCurrentMoveIndex(): Integer
+ updateHistory(newMove: Dictionary<string, string>, isPlayerMove: Boolean): void
+ onUndo(): Boolean
+ onRedo(): Boolean
+ loadFromFile(): List<Dictionary<string, string>>
+ exportToFile(): Boolean
+ loadGameHistory(history: List<Dictionary<string, string>>)
1
1
BoardFW
# gameBoard: string[]
+ getBoardState(historyState: List<Dictionary<string, string>>, currentMoveIndex: Integer): string[]
# initBoard(): string[]
# updateBoard(historyState: List<Dictionary<string, string>>, currentMoveIndex: Integer): void
+ displayBoard(historyState: List<Dictionary<string, string>>, currentMoveIndex: Integer): void
+ getAvailableBoardInfo(currentMoveIndex: Integer): Dictionary<string, List<int>>
1
1
UserInterfaceFW
+ board: BoardFW
+ message: MessageFW
+ input: InputHandler
+ getHelp(currentMoveIndex: Integer): void
1
1
1
1
InputHandlerFW
# rawInput: string
# parsedInput: Dictionary<string, string>
+ getParsedInput(): Dictioanry<string, string>
+ promptInput(): Boolean
+ freezePrompt(): void
+ invalidPrompt(): void
+ clearConsole(): void
+ parsedInput(): Boolean
MessageFW
- messages: Dictionary<string, string>
+ addMessage(title: string, content: string): void
+ displayMessage(title: string): void
1
Help System
getHelp(currentMoveIndex: Integer)
Help System as a function that collaborates information from different components.


Object Diagram
Scenario of Objects
During a game of Numerical Tic-Tac-Toe, in the middle of the play, between a Human Player and a Computer Player. In this round, Human Player makes a move at cell 5 with symbol 5.
player2: ComputerPlayerFW
playerInfo = {"name": "player246", type: "computer"}
player1: HumanPlayerFW
playerInfo = {"name": "John", type: "human"}
history: GameHistoryFW
history = [{"name": "John", "type": "human", "cell": "5", "symbol": "3"}, {"name": "player246", "type": "computer", "cell": "3", "symbol": "4"}]
currentMoveIndex = 2
undoThreshold = -1
NumericalTicTacToe
static void main(string[] args)
logic: Logic
players = [ComputerPlayerFW object, HumanPlayerFW object]
history = GameHistoryFW object
validator = Validator object
validator: Validator
* Accepts parameters into overridden methods
Use
runner: GameRunner
gameMode = 0
UI = UserInterface object
logic = Logic object
board: Board
gameBoard = [" ", " ", " 4 ", " ", " 3 ", " ", " ", " ", " "]
UI: UserInterface
board = Board object
message = MessageFW object
input = InputHandler object
input: InputHandler
rawInput = "m:5:5"
parsedInput = {"type": "move", "cell": "5", "symbol": "5"}
Help System
getHelp(currentMoveIndex: Integer)
Help System as a function that collaborates information from different components.
message: MessageFW
messages = {"gameGuide": "Game Rules:\n - Two players take turn to place a number between 1 - 9.\n - First Player -> Only Uses Odd Num...", ...}

Sequence Diagram

Scenario of Diagram
After loading a game from a saved file (Computer VS Human), the Human Player makes a valid move, then save (export) the game to file.



Design Principles and Patterns

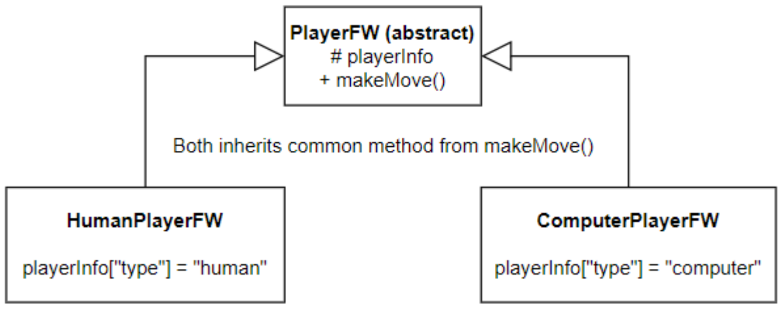
1. **Template Method**

   The participating classes are: **PlayerFW, HumanPlayerFW, ComputerPlayerFW**.

   When creating an instance of ComputerPlayerFW and HumanPlayerFW, they will inherit from the abstract class parent, PlayerFW. Each inherited class will implement their own attribute, “playerInfo”, that differentiates the type of players. These two inherited classes will inherit methods from the parent class, such as makeMove(), which allows the player to place a new move on the board.

   In my design, the computer player and human player will be created while initializing the game. The computer player will have a type of “computer” while the human player will have a type of “human” which differentiates them. Moreover, the ComputerPlayerFW will have a random name generated by its class.

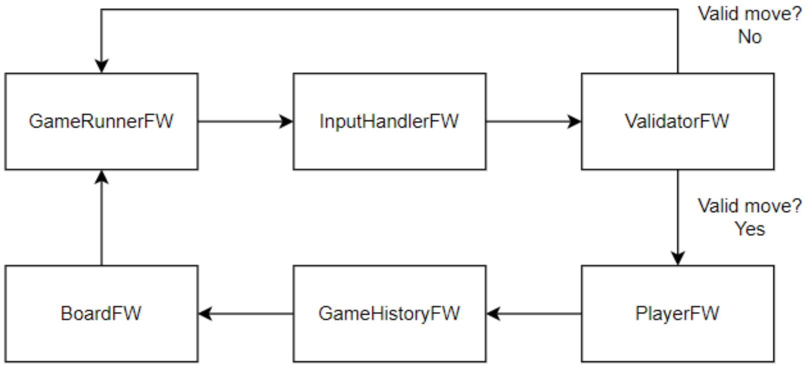
   When a human player makes a move, it is no different from the move made by a computer player. This is because the move made by a human player is first validated by the ValidatorFW class, then, the information is sent to the function. This would mean that information sent to the makeMove() function for both classes are the same. Therefore, they inherit this function from their abstract parent class, PlayerFW.


PlayerFW (abstract)
# playerInfo
+ makeMove()
Both inherits common method from makeMove()
HumanPlayerFW
playerInfo["type"] = "human"
ComputerPlayerFW
playerInfo["type"] = "computer"

2. **Chain of Responsibility Method**

   The participating classes are: **GameRunnerFW, InputHandlerFW, ValidatorFW, PlayerFW, GameHistoryFW, BoardFW**.

   When the game is running (via GameRunnerFW object), the InputHandler object will prompt the user in the console to enter a move command. After the user has entered the move command, it will be passed to ValidatorFW object to validate if the move command is valid. If the move command is valid, it will then be passed to the PlayerFW object, or the method that HumanPlayerFW inherited from the PlayerFW object to make the move. After doing this, the GameHisotryFW object is then triggered, thus updating the history of the game. Finally, when the BoardFW object displays the game board, it first updates the board by retrieving the information from the GameHistoryFW object, then the board is displayed back to the console.
   This is to ensure the entire flow of the gameMove is nicely handled, by passing through the information class by class, thus, making is more maintainable.



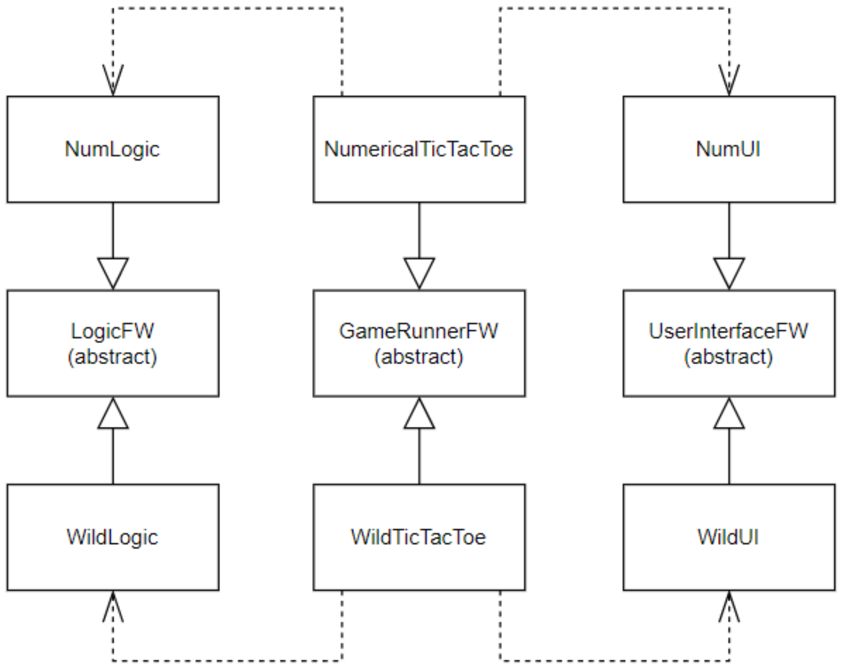
3. **Combination of Factory and Abstract Method**

   The participating classes are: **PlayerFW, ValidatorFW, BoardFW, InputHandlerFW, UserInterfaceFW.**

   The classes stated above are abstract base classes, which mean that the actual classes of these classes will be determined based on the specific game being implemented using the framework. This part can be deemed as Factory method as the appropriate objects of specific types of board game are created, without exposing the details of the implementation in each class.

   Also, as there are multiple abstract classes for each category with multiple related concrete classes, which each class contains methods that return their own instances of corresponding concrete classes. Therefore, it is also an abstract factory method.

   The use of this combination is to maximize the flexibility of the framework, as each board game has a different game rule, and by implementing this, it will make the framework extensible.



Execution of the program

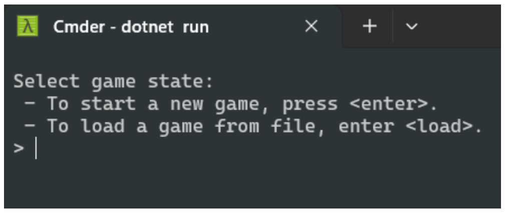
Running the game

1. To run this game in a terminal or a console, first **navigate to the folder** containing the code of the game.
2. Make sure that a **usual terminal or a console** is used, and **not debugging console**. This is because in the game, Console.Clear() is implemented in the code, where it will not work for a debugging console.
3. After navigating to the correct directory, first enter "**dotnet build**" to build the program.
4. Upon doing this, enter "**dotnet run**", which will execute the game.
5. The game will be ended upon a successful play, or to end the game at any time, press <ctrl + c>.

Game manual

Loading a previous game

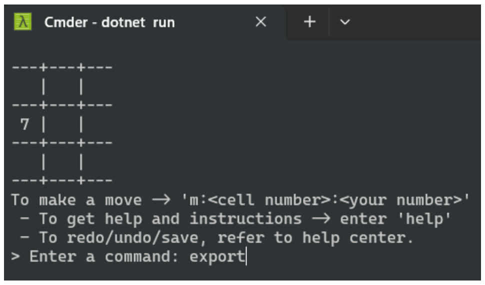
- In the starting of the game, you will be prompt whether to load a previous game.
  To load, simply enter <**load**> and follow the manual; To start a new game, simply press <**enter**>.



```
Select game state:
 - To start a new game, press <enter>.
 - To load a game from file, enter <load>.
> 
```

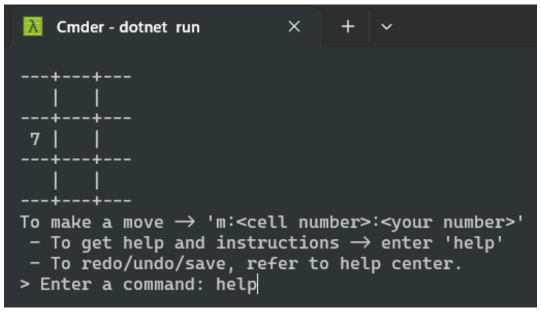
Commands while playing the game

- During the game, to save your current game, simply enter <**export**>.



```
---+---+---
   |   |
---+---+---
 7 |   |
---+---+---
   |   |
---+---+---
To make a move -> 'm:<cell number>:<your number>'
 - To get help and instructions -> enter 'help'
 - To redo/undo/save, refer to help center.
> Enter a command: export
```

- To get more details regarding on other commands and examples of commands such as **redo**, **undo**, **move**, enter <**help**>, and follow the prompt.



```
---+---+---
   |   |
---+---+---
 7 |   |
---+---+---
   |   |
---+---+---
To make a move -> 'm:<cell number>:<your number>'
 - To get help and instructions -> enter 'help'
 - To redo/undo/save, refer to help center.
> Enter a command: help
```

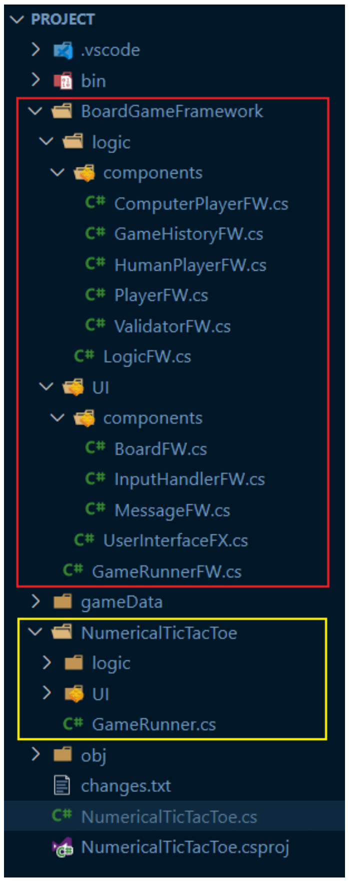
Reused library summary

- System.Int32
- System.Boolean
- System.String
- System.Array
- System.Collections.Generic.List
- System.Collections.Generic.Dictionary
- System.Threading.Thread
- System.Text.Json.JsonSerializer

Information on Project Structure

In the project folder, as shown in the image below:

- The section that is surrounded by the **red** box, is the general **reusable framework.**
- The section that is surrounded by the **yellow** box, is the **Numerical Tic Tac Toe** game that uses the framework.
- The saved file of the **game data** is located in "**gameData**" directory.



```
PROJECT
  > .vscode
  > bin
  v BoardGameFramework
    v logic
      v components
          ComputerPlayerFW.cs
          GameHistoryFW.cs
          HumanPlayerFW.cs
          PlayerFW.cs
          ValidatorFW.cs
        LogicFW.cs
    v UI
      v components
          BoardFW.cs
          InputHandlerFW.cs
          MessageFW.cs
        UserInterfaceFX.cs
      GameRunnerFW.cs
  > gameData
  v NumericalTicTacToe
    > logic
    > UI
      GameRunner.cs
  > obj
    changes.txt
    NumericalTicTacToe.cs
    NumericalTicTacToe.csproj
```