1. Statement of completion

Game requirements:

No	Detailed information	Completed	Comments
1	C# console application on .NET 6 providing a text-based command-line interface	Yes	- The program has been tested with the commands: dotnet clean, build, and run - .NET version 6.0.201
2	The framework is extensible and reusable for a wide variety of board games	Yes	Template Method Design Pattern used as a template board game. Abstract class: GameController Concrete class: WTTTGameController
3	2 modes of play, including: • Human versus human • Human versus computer	Yes	Abstract Factory Design Pattern used. • Abstract class: Player • 2 concrete classes: HumanPlayer & ComputerPlayer
4	Implementing one game between Numerical Tic-Tac-Toe and Wild Tic-Tac-Toe (**chosen**)	Yes	• Symbol 'X' for player1 & 'O' for player2. • Winning rules for Wild Tic-Tac-Toe implemented.
5	For human players, the system must check the validity of moves	Yes	MakeMove() in the concrete class HumanPlayer has modified
6	For computer players, the system can randomly select a valid move	Yes	MakeMove() in the concrete class ComputerPlayer has modified
7	The game can be played from start to finish and can be saved and restored from a saved file	Yes	GameState can be saved to a file and be loaded from a file at any time.
8	All moves from both players can be undoable and re-doable	No	Some errors were found but still not able to fix properly.
9	Online help system	Yes	Help class implemented.

2. Overview of final design

Summary of some change

No	Preliminary design	Final design	Reasons for this change
1	GameStart class	Program class	• C# application starts with the Program class • The Main function within the Program class will execute, initiate the game, and create players
2	WildTicTacRule class	WTTTGameController class	• WTTTGameController is a concrete class from the abstract class GameController when applying Abstract Factory Design Pattern. • GameController is the framework for general board games: ○ Start the game, creating a new Game State ○ End the game ○ Create 2 players ○ Make turns for players
3	BoardRender class		This class was removed because Game State has own display function
4	GameState class	GameState class	Updated important attributes • board: char[] • playerMove: int [] • turn: int • currentPlayerID: int
5	Player class	Player class	Updated important attributes • playerID: int • symbol: char
6	giveCommand() in Player class	• giveCommand() in HumanPlayer class • giveCommand() in ComputerPlayer()	• Modify giveCommand() for concrete class HumanPlayer, user will type a command • Modify giveCommand() for concrete class ComputerPlayer, a valid move randomly generated.

7	Board game with positions in row and column	Board game with positions from 1 to 9	A single dimensional Array is easier to operate than a multidimensional array.

3. Detailed Design Documents

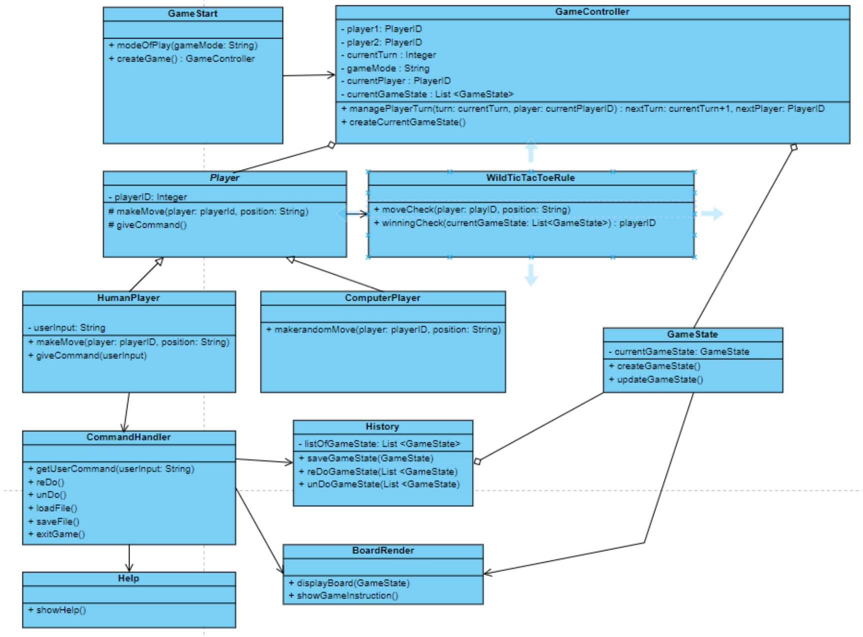

GameStart
+ modeOfPlay(gameMode: String)
+ createGame() : GameController
GameController
- player1: PlayerID
- player2: PlayerID
- currentTurn : Integer
- gameMode : String
- currentPlayer : PlayerID
- currentGameState : List <GameState>
+ managePlayerTurn(turn: currentTurn, player: currentPlayerID) : nextTurn: currentTurn+1, nextPlayer: PlayerID
+ createCurrentGameState()
Player
- playerID: Integer
# makeMove(player: playerId, position: String)
# giveCommand()
WildTicTacToeRule
+ moveCheck(player: playID, position: String)
+ winningCheck(currentGameState: List<GameState>) : playerID
HumanPlayer
- userInput: String
+ makeMove(player: playerID, position: String)
+ giveCommand(userInput)
ComputerPlayer
+ makerandomMove(player: playerID, position: String)
GameState
- currentGameState: GameState
+ createGameState()
+ updateGameState()
CommandHandler
+ getUserCommand(userInput: String)
+ reDo()
+ unDo()
+ loadFile()
+ saveFile()
+ exitGame()
History
- listOfGameState: List <GameState>
+ saveGameState(GameState)
+ reDoGameState(List <GameState)
+ unDoGameState(List <GameState)
BoardRender
+ displayBoard(GameState)
+ showGameInstruction()
Help
+ showHelp()

Diagram 1.1 Preliminary Class Diagram

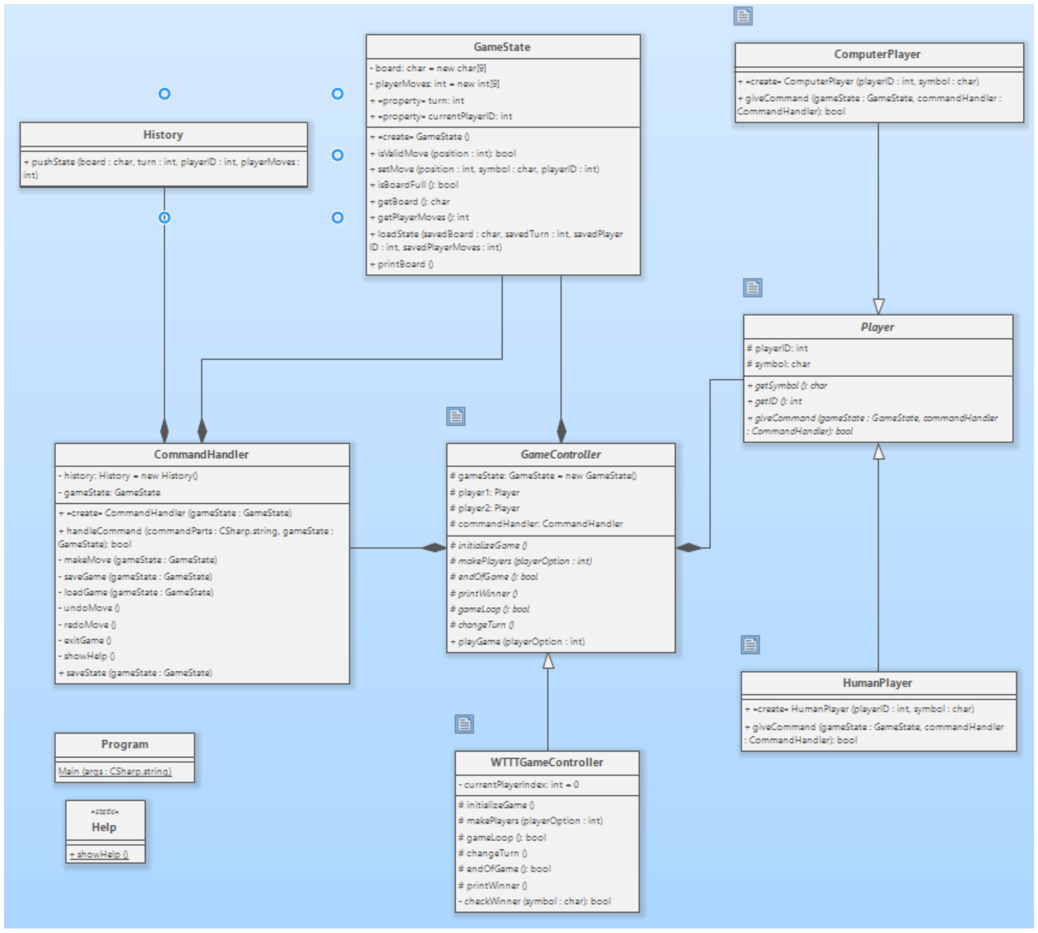
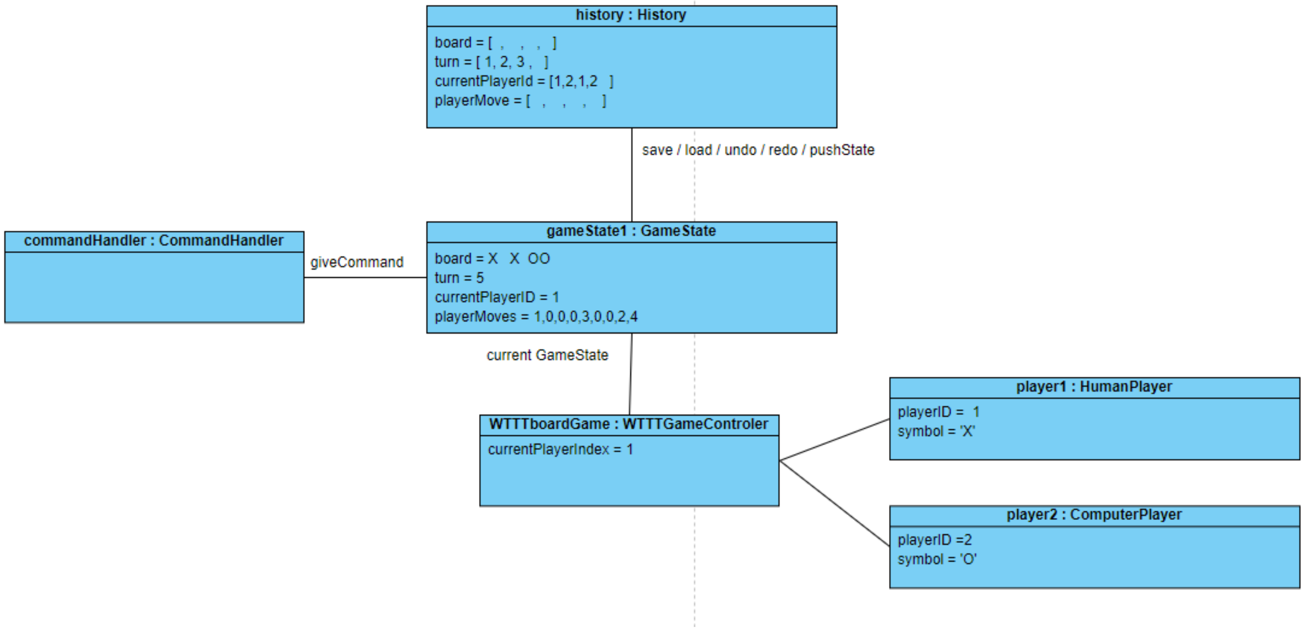
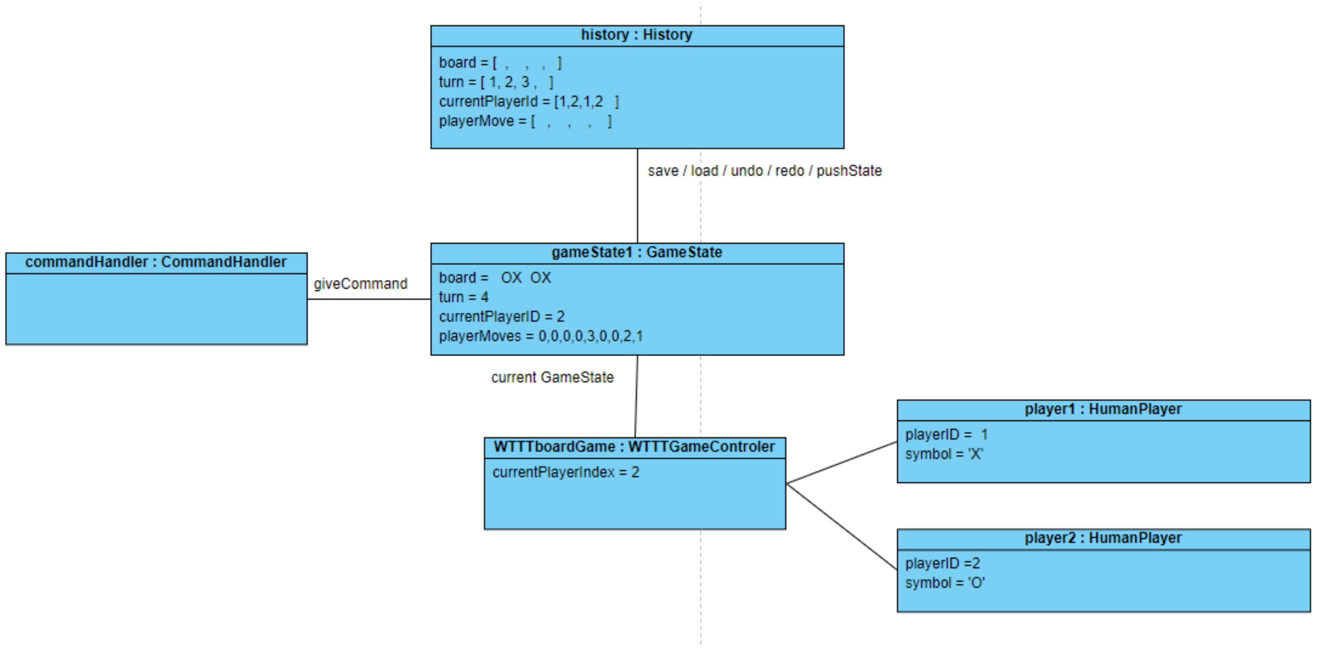

GameState
- board: char = new char[9]
- playerMoves: int = new int[9]
+ «property» turn: int
+ «property» currentPlayerID: int
+ «create» GameState ()
+ isValidMove (position : int): bool
+ setMove (position : int, symbol : char, playerID : int)
+ isBoardFull (): bool
+ getBoard (): char
+ getPlayerMoves (): int
+ loadState (savedBoard : char, savedTurn : int, savedPlayerID : int, savedPlayerMoves : int)
+ printBoard ()
History
+ pushState (board : char, turn : int, playerID : int, playerMoves : int)
ComputerPlayer
+ «create» ComputerPlayer (playerID : int, symbol : char)
+ giveCommand (gameState : GameState, commandHandler : CommandHandler): bool
Player
# playerID: int
# symbol: char
+ getSymbol (): char
+ getID (): int
+ giveCommand (gameState : GameState, commandHandler : CommandHandler): bool
CommandHandler
- history: History = new History()
- gameState: GameState
+ «create» CommandHandler (gameState : GameState)
+ handleCommand (commandParts : CSharp.string, gameState : GameState): bool
- makeMove (gameState : GameState)
- saveGame (gameState : GameState)
- loadGame (gameState : GameState)
- undoMove ()
- redoMove ()
- exitGame ()
- showHelp ()
+ saveState (gameState : GameState)
GameController
# gameState: GameState = new GameState()
# player1: Player
# player2: Player
# commandHandler: CommandHandler
# initializeGame ()
# makePlayers (playerOption : int)
# endOfGame (): bool
# printWinner ()
# gameLoop (): bool
# changeTurn ()
+ playGame (playerOption : int)
HumanPlayer
+ «create» HumanPlayer (playerID : int, symbol : char)
+ giveCommand (gameState : GameState, commandHandler : CommandHandler): bool
Program
Main (args : CSharp.string)
«static»
Help
+ showHelp ()
WTTTGameController
- currentPlayerIndex: int = 0
# initializeGame ()
# makePlayers (playerOption : int)
# gameLoop (): bool
# changeTurn ()
# endOfGame (): bool
# printWinner ()
- checkWinner (symbol : char): bool

Diagram 1.2 Final Class Diagram



1.3 Final Object Diagram – Human vs Computer



1.4 Final Object Diagram – Human vs Human

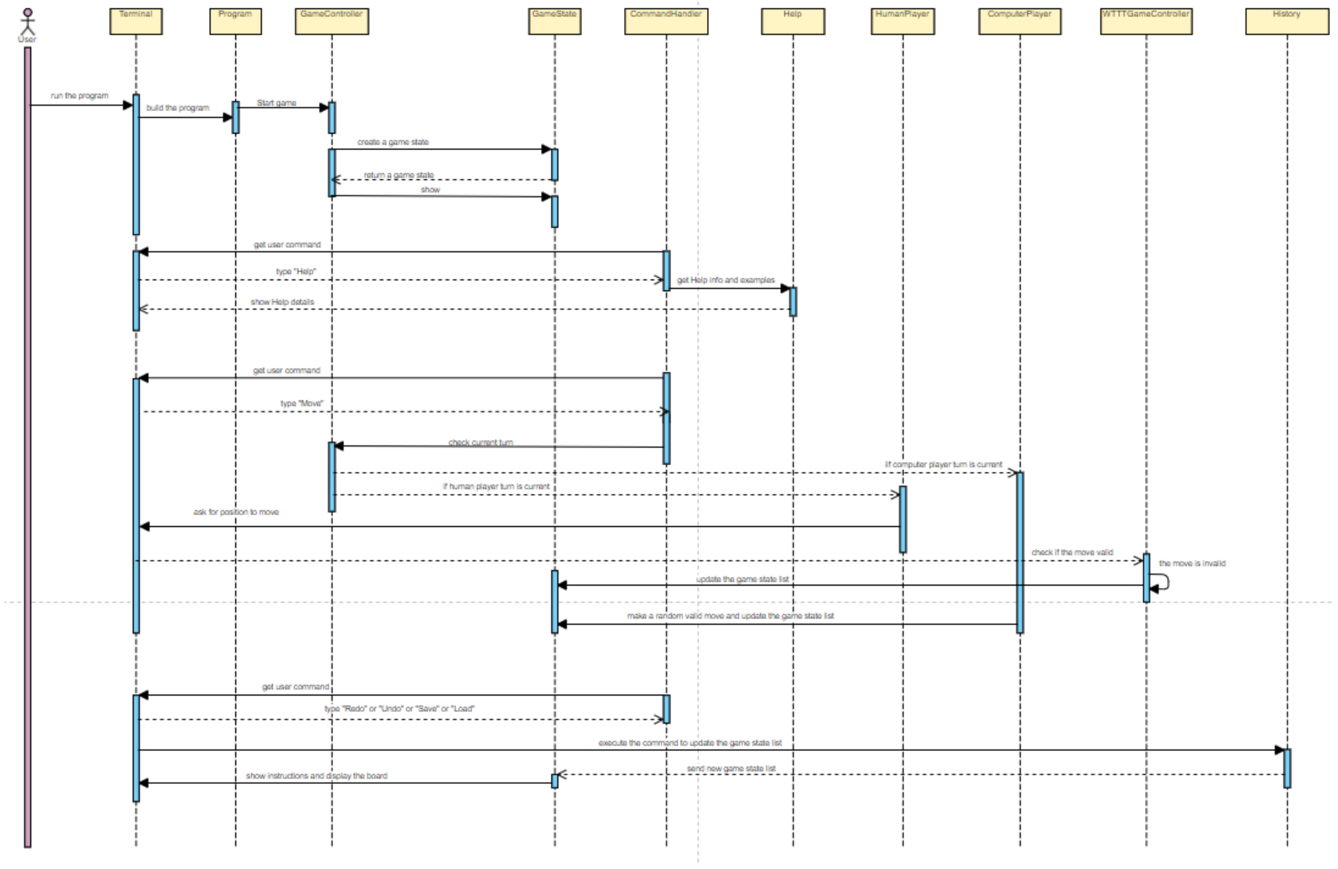


Diagram 1.5 Final Sequence Diagram

4. Design Principles and Patterns

There are two design patterns used in this application:

- Template Method design pattern
- Abstract Factory design pattern

4.1 Template Method design pattern

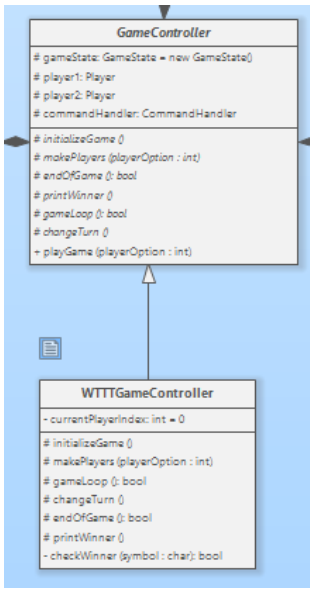
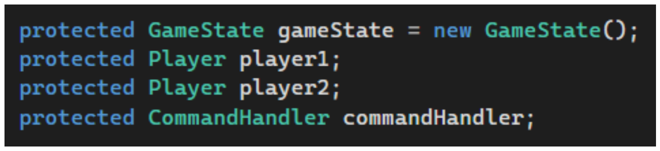


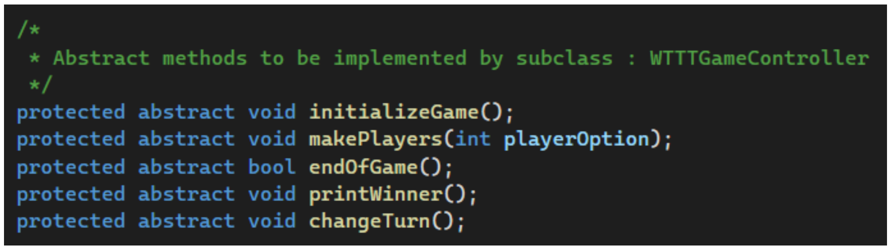
Diagram 1.6 Template Method Design Pattern

- GameController is the Abstract Class as a framework for general board games, that gives a game structure. All abstract functions must be declared empty with protected abstract:
    - Create a new GameState
    - Create a new CommandHandler
    - Create players
    - Make turns for all players
    - Check the winner
    - End game
- The Abstract Class must have the template method which is used by the Concrete Class. Here, playGame function is the template method of the Abstract Class GameController. And it will be used by the Concrete Class WTTTGameController.
- All concrete methods in the Concrete Class WTTTGameController must override to the Abstract methods in the Abstract Class GameController.
- All Abstract attributes within GameController Class:



```
protected GameState gameState = new GameState();
protected Player player1;
protected Player player2;
protected CommandHandler commandHandler;
```

- All Abstract methods within GameController Class:



```
/*
 * Abstract methods to be implemented by subclass : WTTTGameController
 */
protected abstract void initializeGame();
protected abstract void makePlayers(int playerOption);
protected abstract bool endOfGame();
protected abstract void printWinner();
protected abstract void changeTurn();
```

- The Abstract Class can be extended to create another board game like Numerical Tic-Tac-Toe. However, in this assignment, the Wild Tic-Tac-Toe board game chosen to build.

4.2 Abstract Factory design pattern

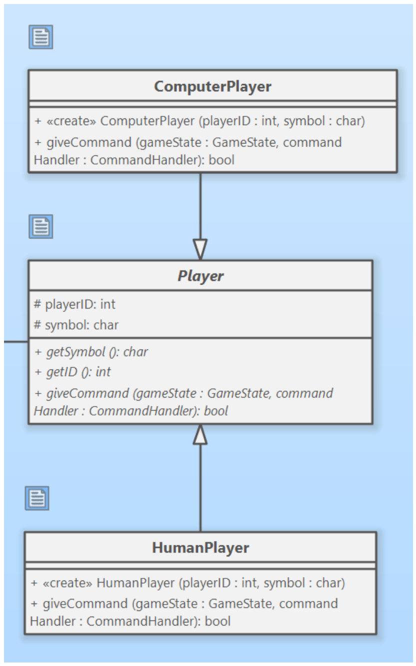
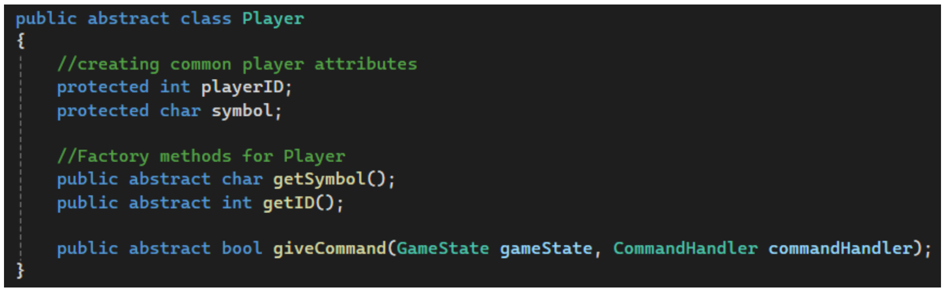

ComputerPlayer
+ «create» ComputerPlayer (playerID : int, symbol : char)
+ giveCommand (gameState : GameState, command Handler : CommandHandler): bool
Player
# playerID: int
# symbol: char
+ getSymbol (): char
+ getID (): int
+ giveCommand (gameState : GameState, command Handler : CommandHandler): bool
HumanPlayer
+ «create» HumanPlayer (playerID : int, symbol : char)
+ giveCommand (gameState : GameState, command Handler : CommandHandler): bool

Diagram 1.7 Abstract Factory design pattern

- Abstract Class Player is declared with common attributes and factory methods used by both Concrete Classes: HumanPlayer and ComputerPlayer:

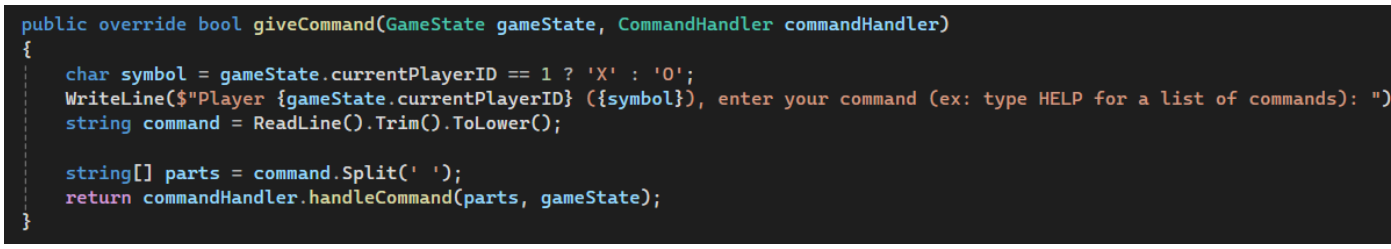


```
public abstract class Player
{
    //creating common player attributes
    protected int playerID;
    protected char symbol;

    //Factory methods for Player
    public abstract char getSymbol();
    public abstract int getID();

    public abstract bool giveCommand(GameState gameState, CommandHandler commandHandler);
}
```

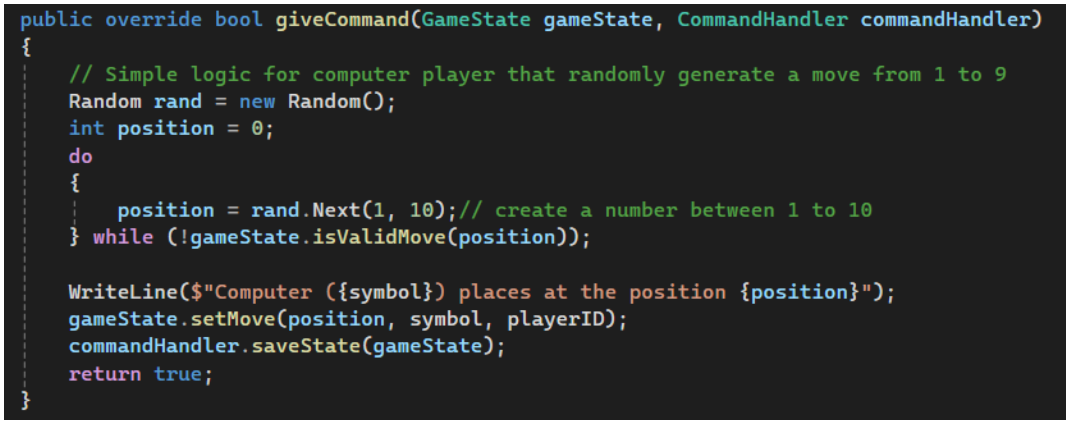
- Each Concrete Class has its function which is different from other Concrete Class, although the function is same.
- For the Concrete Class: HumanPlayer, the function giveCommand will get the input of command form user.



```
public override bool giveCommand(GameState gameState, CommandHandler commandHandler)
{
    char symbol = gameState.currentPlayerID == 1 ? 'X' : 'O';
    WriteLine($"Player {gameState.currentPlayerID} ({symbol}), enter your command (ex: type HELP for a list of commands): ")
    string command = ReadLine().Trim().ToLower();

    string[] parts = command.Split(' ');
    return commandHandler.handleCommand(parts, gameState);
}
```

- For the Concrete Class: ComputerPlayer, the function giveCommand implements a random valid move:



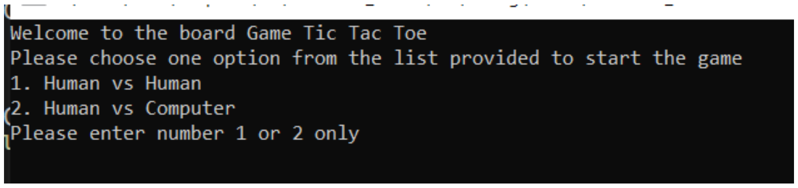
```
public override bool giveCommand(GameState gameState, CommandHandler commandHandler)
{
    // Simple logic for computer player that randomly generate a move from 1 to 9
    Random rand = new Random();
    int position = 0;
    do
    {
        position = rand.Next(1, 10);// create a number between 1 to 10
    } while (!gameState.isValidMove(position));

    WriteLine($"Computer ({symbol}) places at the position {position}");
    gameState.setMove(position, symbol, playerID);
    commandHandler.saveState(gameState);
    return true;
}
```

- By applying Abstract Factory design pattern, other type of player can be created without affecting the game structure.

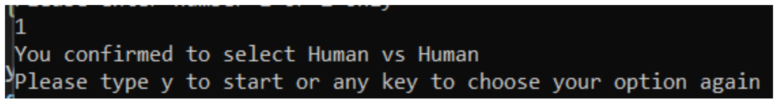
5. Execution instructions

5.1 The game starts with a welcome statement and a guide to enter one number 1 or 2 according to the mode of play preferred.



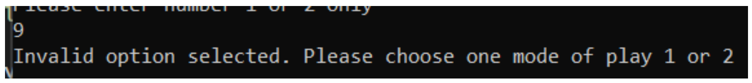
```
Welcome to the board Game Tic Tac Toe
Please choose one option from the list provided to start the game
1. Human vs Human
2. Human vs Computer
Please enter number 1 or 2 only
```

5.2 When a valid number is entered, an instructed statement will be shown to confirm a game started by typing 'y'



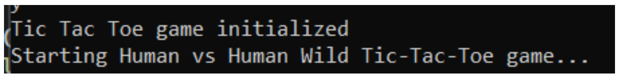
```
1
You confirmed to select Human vs Human
Please type y to start or any key to choose your option again
```

5.3 If user enters invalid option, user is required to enter again



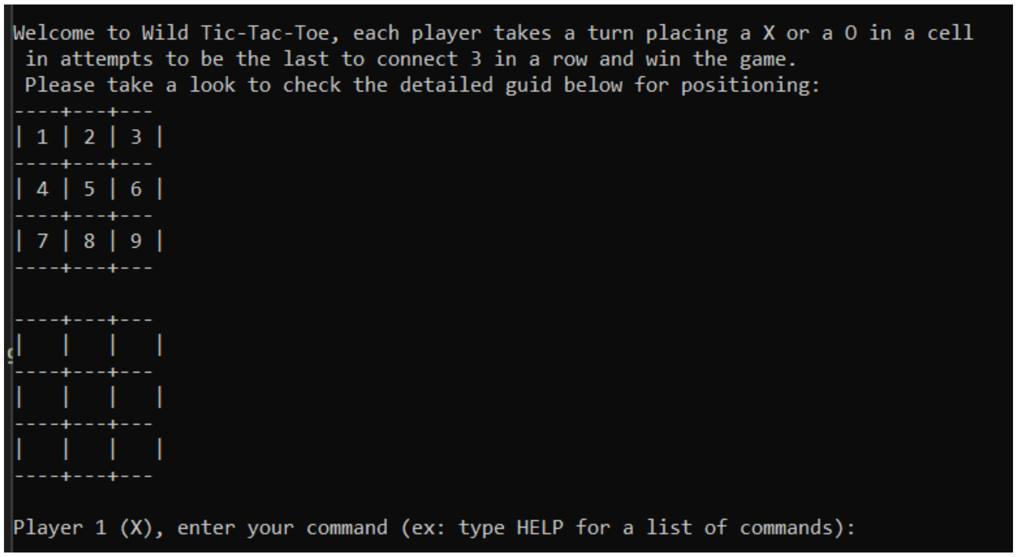
```
9
Invalid option selected. Please choose one mode of play 1 or 2
```

5.4 When the game starts, the screen will show a confirmation of play mode that the user chooses between Human vs Human or Human vs Computer.



```
Tic Tac Toe game initialized
Starting Human vs Human Wild Tic-Tac-Toe game...
```

5.5 Detailed instructions and a position map are shown to assist the user to know their symbol and how to place the position

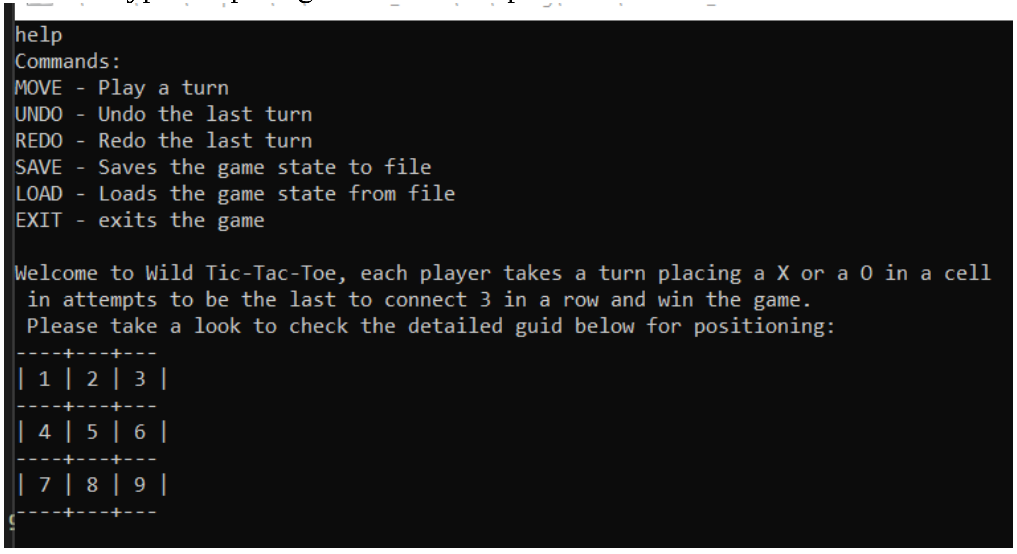


```
Welcome to Wild Tic-Tac-Toe, each player takes a turn placing a X or a O in a cell
 in attempts to be the last to connect 3 in a row and win the game.
 Please take a look to check the detailed guid below for positioning:
----+---+---
| 1 | 2 | 3 |
----+---+---
| 4 | 5 | 6 |
----+---+---
| 7 | 8 | 9 |
----+---+---

----+---+---
|   |   |   |
----+---+---
|   |   |   |
----+---+---
|   |   |   |
----+---+---

Player 1 (X), enter your command (ex: type HELP for a list of commands):
```

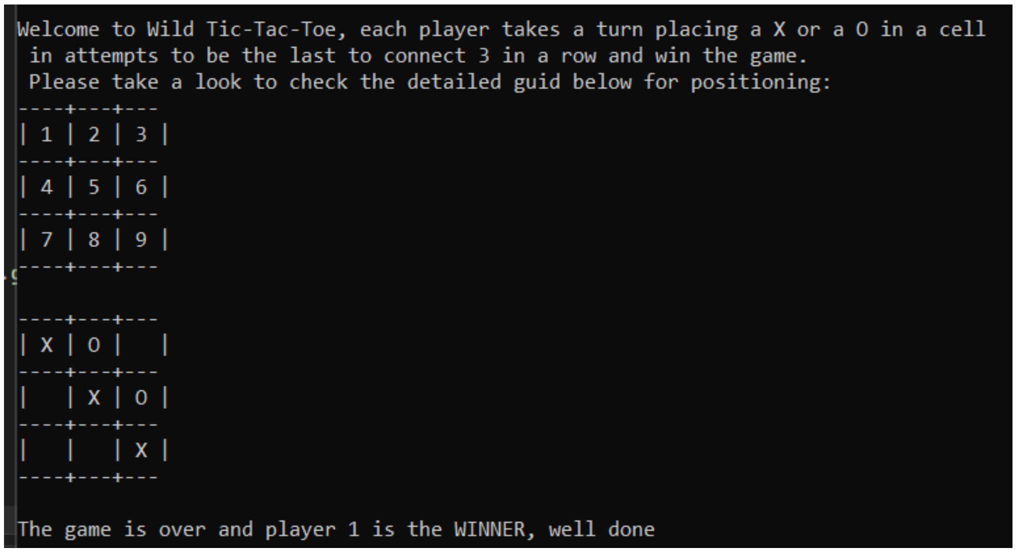
5.6 user can type ‘help’ to get a detailed explanation of all commands



```
help
Commands:
MOVE - Play a turn
UNDO - Undo the last turn
REDO - Redo the last turn
SAVE - Saves the game state to file
LOAD - Loads the game state from file
EXIT - exits the game

Welcome to Wild Tic-Tac-Toe, each player takes a turn placing a X or a O in a cell
 in attempts to be the last to connect 3 in a row and win the game.
 Please take a look to check the detailed guid below for positioning:
----+---+---
| 1 | 2 | 3 |
----+---+---
| 4 | 5 | 6 |
----+---+---
| 7 | 8 | 9 |
----+---+---
```

5.7 When the player has one winning pattern satisfied, he will win. The game will inform that.



```
Welcome to Wild Tic-Tac-Toe, each player takes a turn placing a X or a O in a cell
 in attempts to be the last to connect 3 in a row and win the game.
 Please take a look to check the detailed guid below for positioning:
----+---+---
| 1 | 2 | 3 |
----+---+---
| 4 | 5 | 6 |
----+---+---
| 7 | 8 | 9 |
----+---+---

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

The game is over and player 1 is the WINNER, well done
```

6. Summary of classes/interfaces

No	Libraries & Frameworks	Classes
1	.NET 6 framework	System.Console
2	.NET 6 framework	FileStream Class (System.IO)
3	.NET 6 framework	System.Linq
4	.NET 6 framework	Stack<T> Class (System.Collections.Generic)