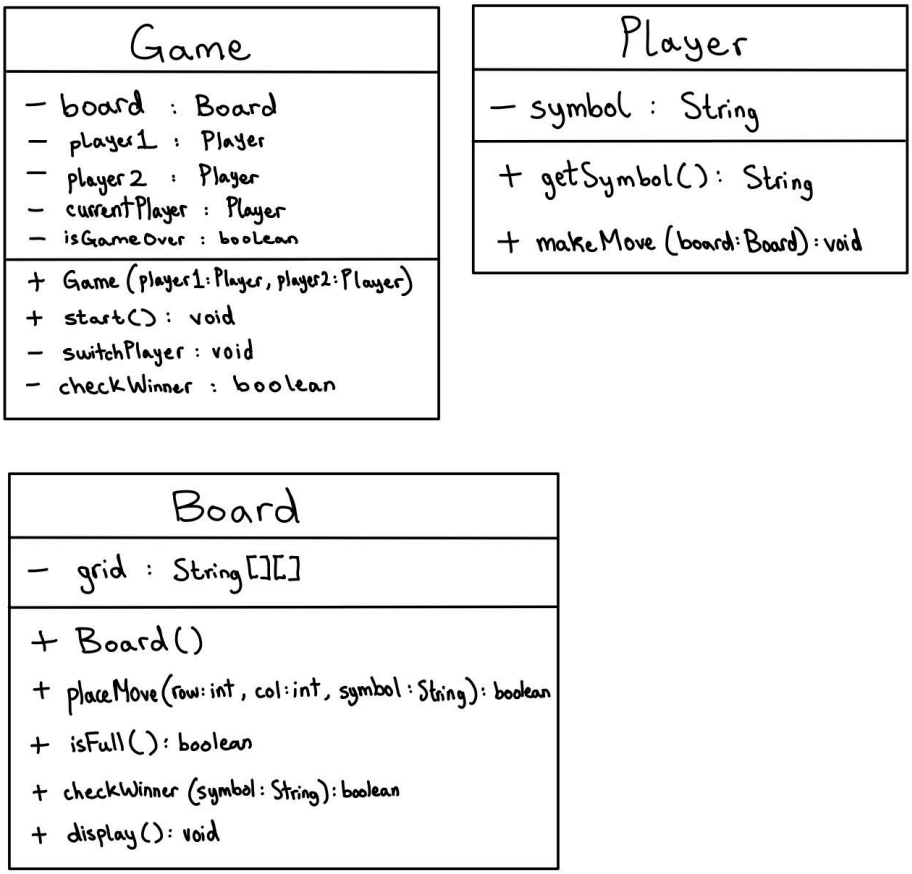
Assignment06-Q1:



Relationships:

- Game depends on Board (composition)
- Game interacts with Player objects (player1 and player2)
- Player is an abstract class and is intended to be extended by specific player types (i.e. HumanPlayer and ComputerPlayer)

Workflow:

- Main sets everything up (players, game, board)
- Game runs the game loop, switching between players
- Player makes moves on the Board
- Board validates moves and checks for a win or draw
- The game repeats or ends based on the player's choice

Assignment06-Q2:

My design reflects the encapsulation principle by using private attributes that can prevent direct access to a class's internal data. For example, Game has private attributes: board, player1, player2, which can't be accessed directly from outside the class. Only the public methods such as startGameLoop() are used to manipulate the internal state. These methods further place controlled access to the game logic, ensuring integrity for the data. In this case, the Player class

has an attribute ‘symbol’ declared as private, meaning that the only way external code could interact with it would be through the getSymbol() method. In the Board class, it encapsulates the grid and provides public methods like place(), display(), and resetBoard() to modify or display the board. Thus, encapsulation guarantees the protection of data for classes and easy maintenance by hiding internal details and showing only the required behavior.

Assignment06-Q3:
The game flow demonstrates solid design principles by assigning specific responsibilities to each class. The Game class controls the overall gameplay, in which it manages the player turns and the conditions to end the game. The Board class is responsible for managing the grid and determining if there is a winner. The Player class defines the behavior of the players within the game. This separation of tasks makes the code more modular, easier to maintain, and scalable for future additions. For instance, features like supporting different player types or changing board sizes can be added without modifying the core logic of the game. Additionally, this approach ensures that the important pieces of data in the game (such as player moves and board status) are encapsulated and accessed only in controlled ways, minimizing the risk of unexpected errors.

Assignment06-Q4:
One of the challenges I faced was ensuring that the program did not crash every time the user would enter an unexpected input, such as entering letters when prompted to enter numbers. When asking for the board size or game mode, the program needed to handle invalid inputs and provide the user with clear feedback. In order to resolve this issue, I used try and catch blocks for exception handling to catch and manage invalid inputs from the user, ensuring that the program would keep on re-prompting the user until a proper input is entered. Another challenge I encountered was creating simple and user-friendly user input. Specifically, I needed to make sure that the program correctly validated and processed the player’s moves in the, “row, column”, format. Thus, handling such errors and maintaining user convenience were important aspects in overcoming the challenges in handling user input and designing a smooth experience.

Assignment06-Q5:
I implemented polymorphism in the Player class by making it abstract and then creating two subclasses, HumanPlayer and ComputerPlayer. Each subclass overrides the makeMove() method to define its own implementation for making a move. For instance, the HumanPlayer subclass uses user input to make a move, whereas the ComputerPlayer subclass randomly selects a valid move. By doing this, the Game class can interact with the Player objects without knowing the actual type of player it is dealing with. Thus, the use of polymorphism makes it easy to add more types of players without modifying the existing game logic.

Assignment06-Q6:
Dynamic dispatch improves the flexibility of my game by enabling the program to determine at runtime which version of the makeMove() method to call based on the type of player. For example, the Game class doesn't need to know whether the player is HumanPlayer or a ComputerPlayer. Dynamic dispatch ensures the correct version of makeMove() is called based on the Player object and the appropriate behavior is executed. This makes the game highly adaptable as I can easily prompt between different player types to make their move without changing the game logic, simplifying future expansions with minimal changes to my code.

Assignment06-Q7:
To check for a winning condition, I implemented a checkForWin() method inside the Board class that checks every row, column, and both diagonals. This is done to see if all cells in a straight line have the same symbol, either X or O. If any of the conditions are met, then the method returns true, meaning there is a winner. Then, after every move, the Game class calls checkForWin() to confirm whether the current player has won the game.

Assignment06-Q8:
To structure the game for restarts without duplicating code, I created an endOfGame() method in the Game class, which resets the game state by resetting the Board and prompting the user to restart or exit the game. Therefore, by making the reset logic in a separate method, I avoided duplicating code when restarting the game. The game flow in the startGameLoop() method checks for the game's end conditions and calls endOfGame() to handle the end-of-game options, ensuring a clean restart without repeating setup logic.

Assignment06-Q9:
During refactoring, I aimed to maximize code readability, remove redundant code, and increase its adaptability. One of those major changes was centralizing the gameplay loop within the startGameLoop() method in the Game class, which allows for smooth restarting without repeating logic. As part of improving player management, polymorphism was introduced by creating an abstract Player class with specialized subclasses for different player types like HumanPlayer and ComputerPlayer. Furthermore, enhanced input validation was incorporated to better handle unexpected errors, thus enriching the flexibility of the application and its user experience. Also, I modularized key functions such as moving the win-checking logic into the Board class, which resultantly produced a far more organized codebase which is easy to maintain and call from the Game class.

Assignment06-Q10:
My final codebase adheres to a number of the key principles of OOP. First, encapsulation is achieved by making class attributes private and allowing only controlled access through methods such as the getSymbol() method in the Player class. Polymorphism is used within the Player class hierarchy, allowing the game to handle different player types uniformly, yet maintain distinct behaviors for each. Inheritance is applied to provide specialized players, like HumanPlayer and ComputerPlayer, which inherit from the Player class. Abstraction is used in Game and Board classes by encapsulating internal complexities and showing only the required public methods to interact with them. Thus, the use of the mentioned OOP principles make the code modular, maintainable, and extensible.

Assignment06-Q11:
In developing the upgraded game, the use of inheritance and polymorphism in the design helped significantly. Inheritance enabled HumanPlayer and ComputerPlayer to reuse shared behaviors from Player base class, like the handling of a player symbol, ensuring no duplication of logic. Polymorphism allowed the Game class to interact with all players generically, through the use of makeMove() abstract method which is implemented for human and computer players without the need to make changes to the game loop. Additionally, the Board class encapsulated the grid size along with operations including validation, resetting, and win-checking, such that the board size is configurable through its constructor without needing to modify other classes. This resulted in a modular design which ensured that an NxN adaptation only required the initialization of the board size, while other components remained untouched. By following these OOP principles, the code became easily maintainable and reusable for different grid sizes and future enhancements.

Assignment06-Q12:
One of the main problems I encountered while reusing the original Game and Board classes for a NxN grid layout size is that they were designed for a fixed size of 3x3, with dimensions and logic reflecting that specific size. Such specificity meant that methods such as win-checking, grid initialization, and move validation would not work dynamically for variable sizes of boards. The root cause lies in the total absence of scaling capability in the original implementation concerning size flexibility. I had to rework the Board class to include a size parameter for the grid initialization rather than having restricted values.