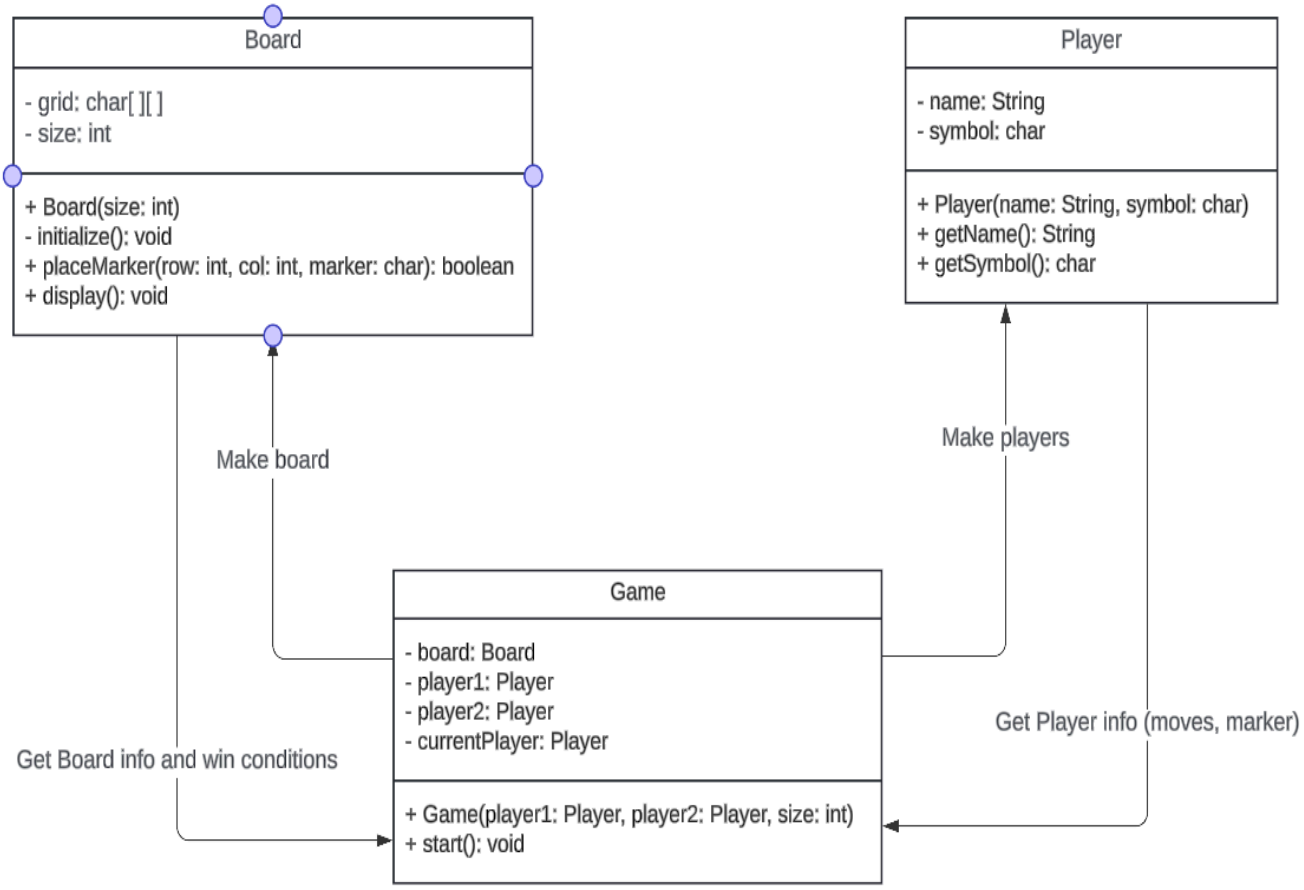
Tic-Tac-Toe Game Report

Draw the UML of each core class and explain your design.



How does your design reflect the encapsulation principle?

All the instance variables in Board, Player, and Game are private (or protected). This restricts direct access to these fields and ensures control of these variables is kept to their respective classes. Instead, I provided controlled access via methods for example: the placeMarker() method in the Board class controls where markers can be placed on the grid, getName() and getSymbol() methods in the Player access to player details for printing “winner messages” at the end of the game. I hid internal implementation details by making methods like initialize(), and private (since it's an internal operation specific for Board objects only). All this allows encapsulation and ensures data integrity.

How does the game flow reflect good design principles?

Single Responsibility Principle: each class has a defined role: Game manages the game loop and interactions between players and the board. Board handles the grid and marker placement Player handles player information and gets moves.

Encapsulation: Classes encapsulate their data, exposing only necessary functionality through methods.

Reusability: The game flow is flexible/modular to support different game modes (e.g., human vs. human, computer vs. human) by providing different player objects.

Open-Close Principle: The design makes adding new features, such as win conditions, grid sizes, and more players easy without modifying the existing code structure (or very few changes).

What challenges did you face in handling user input?

Problems I had with ensuring that user input was valid: checking if the row/column was within bounds or if the cell was already occupied.

I solved this by providing clear instructions and having proper error handling (try/catch/throw exceptions) ex: when a player attempts to place a marker in an occupied cell, the game informs them and allows them to retry.

1-Starting indexing for human players: This was done because humans typically think of 1 as the starting number for rows and columns. I made sure the game loop continued until a valid input was received from the user to avoid errors/programs crashing.

How did you implement polymorphism in your player classes?

The Player superclass contains shared attributes (name, marker) and a method (getName, getMarker) for all players. It also defines an abstract method getMove, which is overridden by subclasses.

The Subclasses override the getMove method to provide specific behaviours for human and computer players (get user input for HumanPlayer and random/smart input for ComputerPlayer).

The Game class interacts with Player objects without needing to know their specific type.

The correct getMove implementation is determined at runtime based on the object type, demonstrating dynamic dispatch.

How does dynamic dispatch improve the flexibility of your game?

Dynamic dispatch improves the flexibility of the game class. Using a common class - Player, the game class is able to interact with different player types in a different way. We can also add new types of players (e.g., a smarter AI) without modifying the Game class. This greatly simplifies the code by relying on polymorphism, making the design cleaner and easier to extend in the future.

How did you check for winning conditions?

The check-in method in the Board class iterates through all rows and columns and checks both diagonals. Using the checkRowsAndCols and checkDiagonals methods to verify if a player has a continuous line of their symbol.

How did you structure the game to allow for restarts without duplicating code?

The promptRestart method in the Game class is called at the end of the while loop (ie when the game is over). The prompt restart method asks the user if they want to play again. The game then creates a new game object and starts the new code with the same player name/conditions, etc, thus facilitating a seamless restart process without duplicating any code.

What improvements did you make during refactoring?
Made the instance variables in Game protected instead of private so they can be accessed in step 7 by its subclass (UpdatedGame). Adhering to the open-close principle.
Added comments for readability
Changed variable names and method names so it's easier to read and understand by other programmers adhering to the self-documenting principle.
Simplified the game flow in Game by delegating responsibilities to the Board and Player. This also adheres to the single responsibility principle.

How does your final codebase adhere to the OOP principles?

All fields are private, and the only necessary methods are public, showing encapsulation.

The modular design of the player and the extendable designs on the game and board show the Open-Close principle (especially useful in step 7)

The comments and variable names adhered to the self-documenting principle. They were clear and easy to understand for other programmers. For example, getMove in Player gets players to move, and start() in the game starts the game loop.

Had getMove as overridden methods in subclasses of Player (HumanPlayer, ComputerPlayer) so they are dynamically dispatched and called based on the type of Player during runtime. Adheres to encapsulation, interface, and single responsibility principle.

Also made sure Game and Board handled functions and tasks relating to their classes, adhering to the single responsibility principle.

What OOP principles help you reuse the previously implemented classes in developing the upgraded game? Why do the OOP principles help?

Encapsulation: By keeping the logic for the Game and Board classes self-contained, I could extend them without changing their internal implementations.
Interface: UpgradedBoard extends the functionality of Board to handle a dynamic grid size and custom winning conditions.
Open-Closed Principle: The existing classes are open for extension (via inheritance/extending) but closed for modification, ensuring compatibility with the upgraded game.

If you encounter difficulties in developing the upgraded game, what are the root causes of those difficulties? How do you tackle the difficulties?

Extending the winning logic to work dynamically with any N×N grids and M-win condition was complex. To solve this I implemented helper methods in the UpgradedBoard class to count consecutive markers dynamically for rows, columns, and diagonals (used 2 for loops - one that loops through **X** index in each row/col; and the other for loop that counted the number of consecutive markers in that row/col starting from **X** index).

The second problem I had was when restarting the game. The new Game object being created was a regular Game object and not an UpdatedGame object. I solved this problem by adding overridden methods in Updated Game (originally it was using the same methods in the Superclass). However, using the overridden methods helped solve that problem.