Assignment06-Q1:

Game	(abstract) **Player**	Board
-board: Board -player1: Player -player2: Player	-symbol: char	-grid: char[][]
+Game(player1: Player, player2: Player) +Play(): void	+Player(symbol: char) +getSymbol(): char +(abstract)makeMove(board: Board): int[]	+Board() +display(): void

The Game class contains the board and player objects, the game logic, and the interactions between the board and player objects. The Player class is initialized with the Player's symbol and contains the abstract method makeMove so that the logic can be defined differently for different subclasses. The Board class contains the private grid which is set up in the constructor and a method to display the grid.

Encapsulation. The internal state of each class is hidden, with functionality only exposed through defined methods. For example, the grid in Board is private, but can be displayed by the public display method. The game class contains the overall game flow, and the player and board objects only interact through the methods contained in this class.

Assignment06-Q2: Encapsulation: multiple classes with clear responsibilities which contribute to modularity. The Game class manages all interactions between the objects, which fully contain their respective operations. Open-Closed Principle: using the abstract Player class with abstract makeMove method, making it easy to implement new Player types without modifying existing code.

One of the challenges I faced in handling user input was deciding how if I wanted the input range of values to be 0-2 or 1-3. I chose 1-3 to be more user-friendly, which only required slight modification in the code to convert to the valid grid indices. Since I already made the Player class abstract, it was easy to create a new HumanPlayer class which extends the Player class, and getting the move input inside of the override makeMove method.

Assignment06-Q3: I already implemented polymorphism in the abstract Player class in the previous steps, with the game logic in the Game class using the abstract method makeMove for dynamic dispatch.

Dynamic dispatch ensures that the correct implementation of makeMove() is called based on the player type at runtime. This allows the game to seamlessly switch between different player types without any additional logic, improving the flexibility of the game.

Assignment06-Q4: The checkWin() method in Board checks horizontal, vertical, and diagonal win conditions (3 identical elements in line) using loops and fixed cases (e.g., [0][0], [1][1], [2][2] for the diagonal).

To implement replay functionality with duplicating code, I put the existing game logic of the play() method inside of another while loop, with a prompt at the end of the loop to continue playing or break the loop.

Assignment06-Q5: I restructured Game.java to split the play() method into a playGame() method, handling the game logic, and a play() method, handling the replay functionality. This way, the methods are split so they each have one responsibility. I also updated the HumanPlayer class to have exception handling for the user input, so the program won't crash for an invalid input.

Encapsulation: Each class has is contained within itself, with interactions (Board and Players) managed within the Game class. Interface Principle: Players share a common interface, enabling flexible logic. Information Hiding: Important data like Board's grid cannot be directly accessed and is only utilized through public methods within the class. Open-Closed Principle: New player types can be added by extending the Player class and overriding the makeMove() method, not modifying the code. Most references to the grid size are not hard-set so its easy to modify in the future to add new functionality. Some important variables are protected rather than private to support future extension. Single-Responsibility Principle: Each class has a clear responsibilty: Game manages game flow, Board handles the grid logic, etc..

Assignment06-Q7: The OOP principles which helped me reuse the previously implemented classes in developing the upgraded game the most were the interface principle and encapsulation principle. Because each class had a well-defined interface, it was easy to extend them to add new functionality. I created two new classes, UpgradedGame (extends Game) and UpgradedBoard (extends Board), and only override and added a few methods in these classes. This was much more efficient than having to recreate each method, and certain variables were "protected", making them accessible to subclasses. The encapsulation principle helped because since each class encapsulates their own state, its easy to extend their functionality with new classes without worrying about how it will affect the other classes. For example, the way that the UpgradedGame game logic interacts with the UpgradedBoard and Players is the same as the super Game class, even though the inner workings of the upgraded classes are different.

The main difficulty I experienced in developing the upgraded game was implementing the win condition checking methods. In the normal Board class, the win conditions are hardcoded to check for the winning conditions of a 3x3 board exclusively. For the UpgradedBoard class I implemented new methods to be called from the override checkWin method, containing more complex logic to check for winning conditions (based on the variable for winLength) for any size of board.