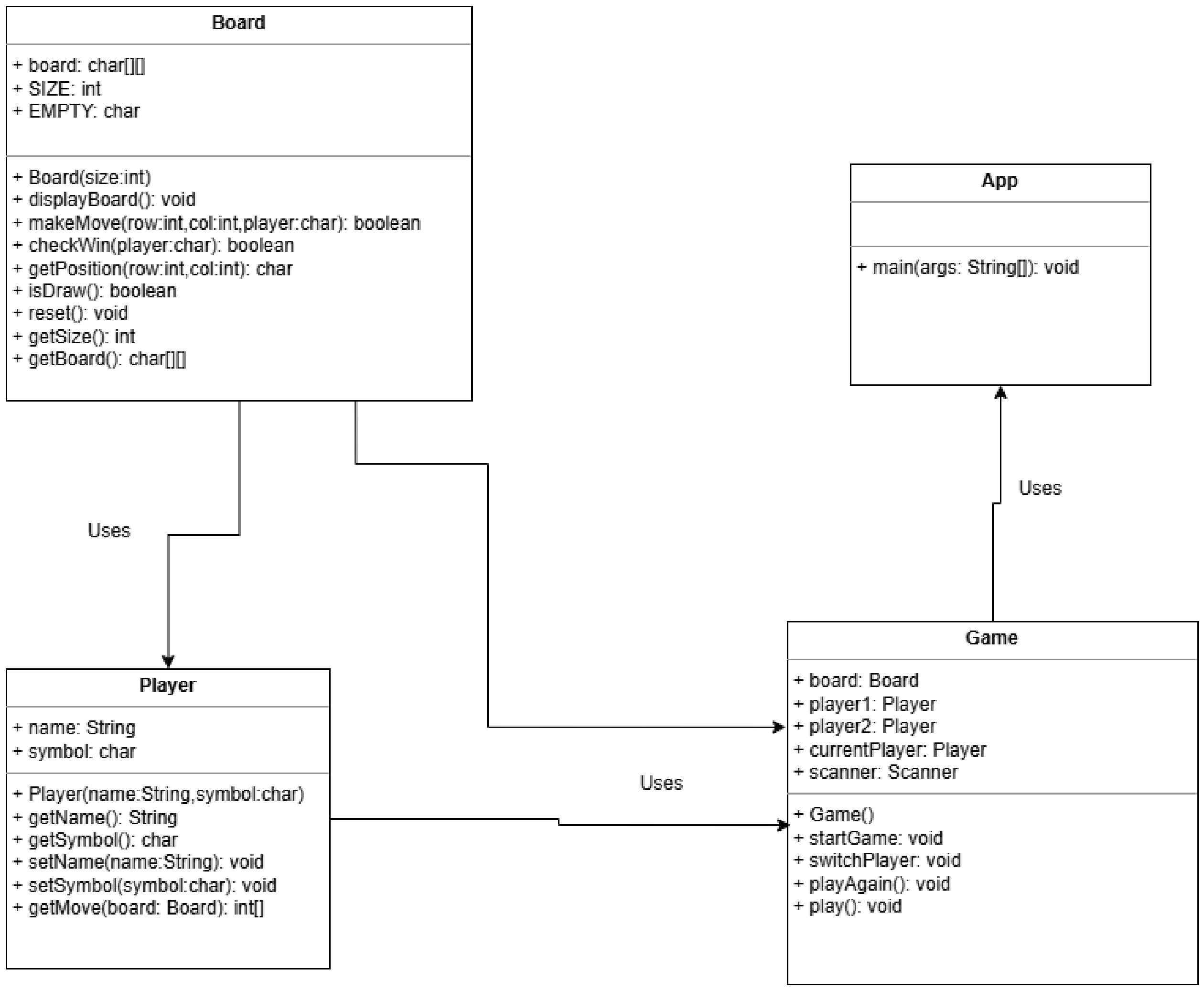
Assignment06-Q1:



Board Class:
Implements encapsulation by having private instance variables like Board and SIZE so they are not accessed from outside the class, while having public methods like makeMove, checkWin, getSize etc to interact with/access parts Board class without modifying it directly

Player Class:
Implements encapsulation by having protected/private instance variables such as player name, while having methods like getName and getSymbol so other classes can access information as needed without modifying it

Game Class:
Game has private instances of the previous classes that are used throughout the class, while having methods that interact with the variables within the class like switchPlayer. It also has it's own private helper methods like startGame which are used within other methods, to prevent them from being called outside the class.

Assignment06-Q2:

It reflects good design principles by implementing object oriented principles to make the flow of the program run smoothly. The game flow is encapsulated within the Game class, which manages the state and behavior of the game. This ensures that the internal state of the game is hidden from other classes and can only be accessed or modified through well-defined methods. Each class has a single responsibility. For example, the Player class handles player-specific data and behavior, the Board class manages the game board, and the Game class controls the overall game flow. This separation of concerns makes the code easier to understand, maintain, and extend. Game also uses information hiding as it has private instances of the Player and Board classes, so that the main function only calls Game and nothing else.

An issue I came across with player input was dealing with the fact that there are so many user inputs, with lots of room for error. In a game like tic tac toe, everything has to be done by the people playing the game, so what they see has to be clear. I added print statements clarifying the inputs for every different case, and structured the flow to make it clear to the user what needs to be done. This also reflects good design principles since the end user knows exactly what to do when the program runs. The classes also contain checks for different errors in case an input is not correct. To simplify it further, a single scanner class is used throughout the entire program so there are no redundant memory streams, which decouples the code.

Assignment06-Q3:

Polymorphism is implemented in the program through the subclasses ComputerPlayer and HumanPlayer which extend Player. The Player class defines common methods for each of its subclasses, and abstract methods that must be implemented within each, that being getMove(), since the implementation is different depending on the subclass even though they have the same purpose. getMove for the HumanPlayer takes user input, but for ComputerPlayer it generates a random position to play on. Dynamic dispatch can be seen in the main game loop, as getMove, getName and getSymbol are called in it, and it knows whether to execute the one from HumanPlayer or ComputerPlayer based on who the current player is. This abides by the open-closed principle and allows for easy extensibility if new player classes need to be added, as no previous code has to be adjusted, and it keeps the code more readable by reducing redundancies.

Assignment06-Q4:

The win conditions in tic-tac-toe are that a straight line in either a row, column, or diagonal has been achieved by one player. Since this is running in a 2d array, we can use nested for loops to check each condition, and do it dynamically based on the size of the array, since the logic holds for any size. The checkWin() method starts by checking the rows, then columns, then each of the diagonals to see if the board has the same character in it, and if it does then return true. I created several methods like startGame(), play() and playAgain() so that the user can replay the game without making redundant code. At the start of each game loop (in play()) the startGame() function is called, and at the end the playAgain() function is called. playAgain() makes use of the resetBoard function to refresh the game without having to restart, and then if the user wants to play again then play() is called again. The game's flow is split into several reusable methods, which allows it to run continuously without repeating code.

Assignment06-Q5:

The first change I made was adding the makeBoard function to Game, as it was a better way of accessing the Board class and makes the code more readable. It also opens it to further implementation, as I looked ahead for step 7 and saw the new features that would be added. I went back and renamed several variable names to be more descriptive in line with the Self-Documenting principle, and added javadoc comments to every function. The encapsulation, information hiding and single responsibility principles were mentioned previously, and the new Player abstract class demonstrated the open-closed principle with the different types of sub classes extending it. The interface principle was not needed as much here, since open-closed fulfills a similar purpose, and there were no other classes besides Player that needed to be extended.

Assignment06-Q7:

The open-closed principle is used as UpgradedGame extends Game, and makes use of polymorphism and method overriding so we do not change any existing code. Since all the upgraded game needs to do is update the makeBoard function to take input, we only override that function and keep every other part of Game the same, so the main function can call upon UpgradedGame with no other changes. They help because it makes it easier to maintain the code and allows for easier implementations in the future like in this example.

One issue I came across was that in my original code, I hard-coded the board to display a 3x3 board, so that needed to be changed here to account for the new values up to 20x20. I used a string builder and dynamically displayed the board based on its size, allowing for cleaner usage.