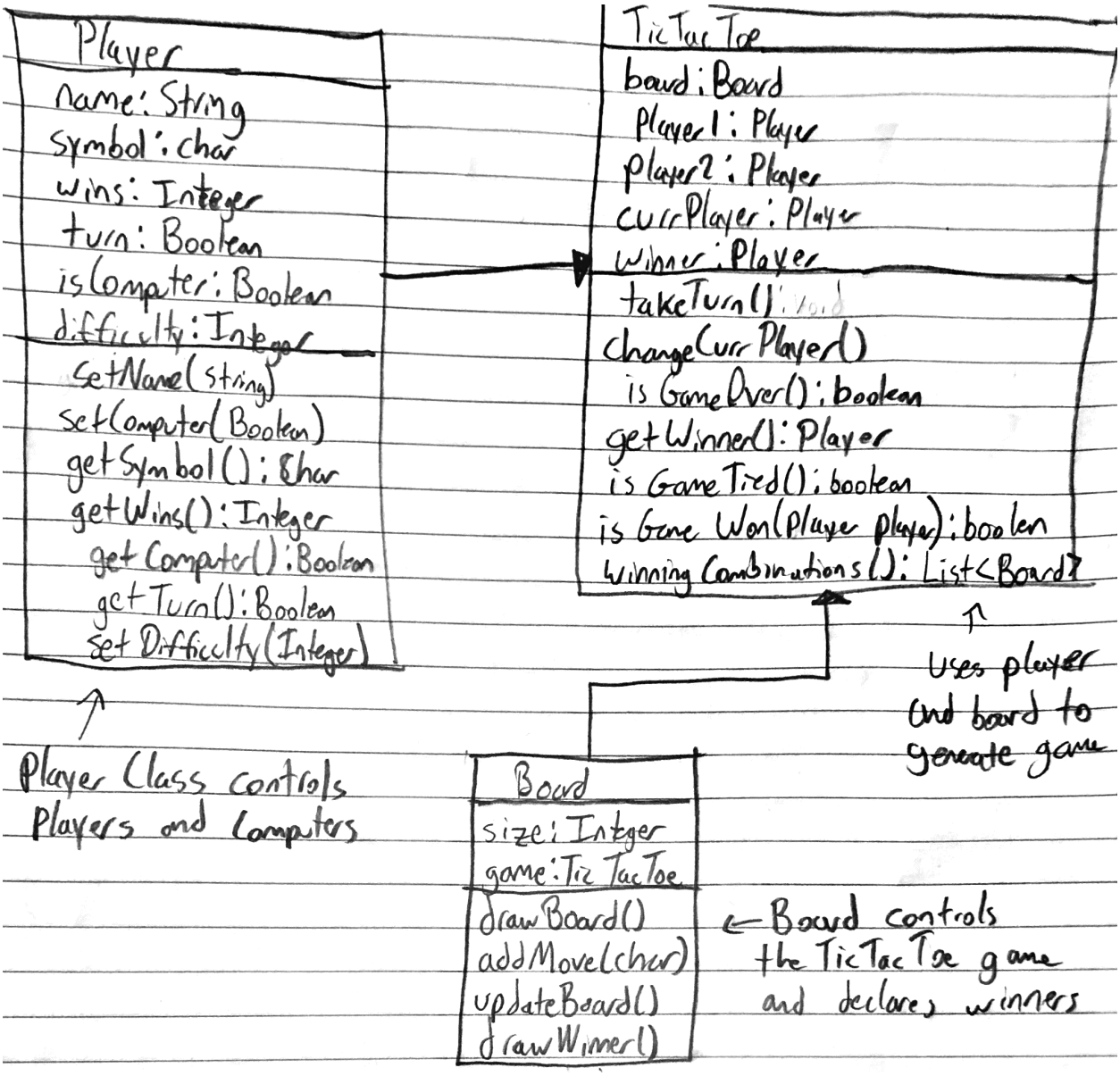
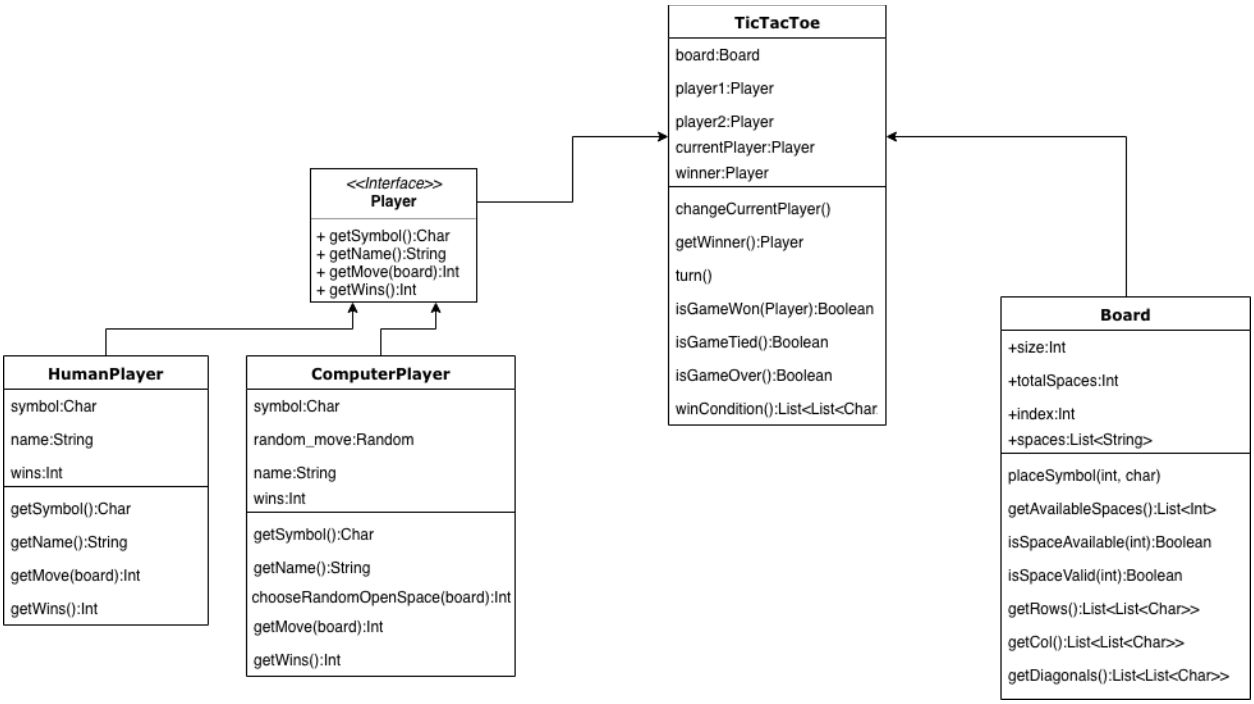


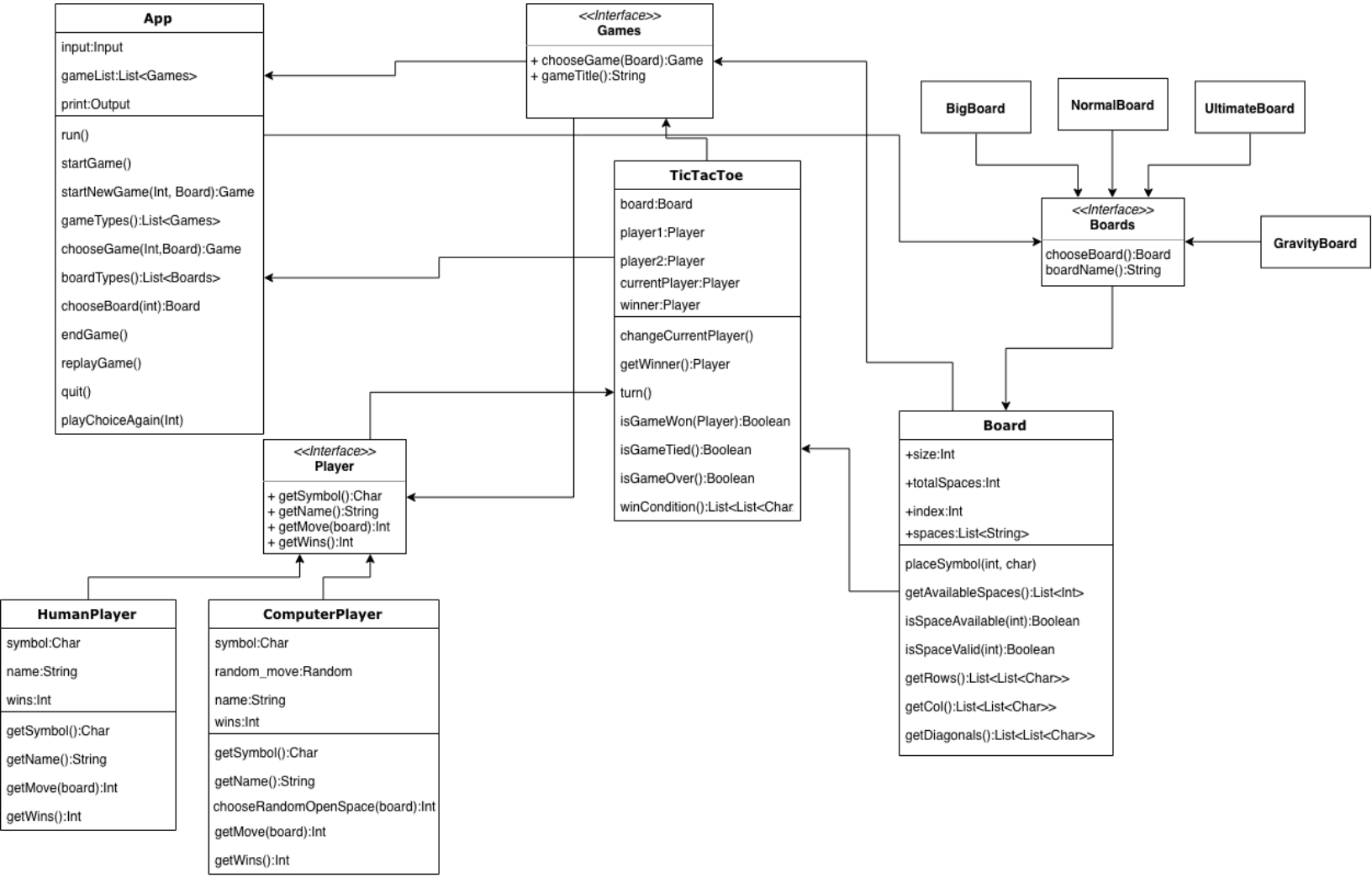
This is my initial design of the TicTacToe game. It uses 3 classes, Player, TicTacToe, and Board. It follows the general design principle of TicTacToe using two different players, each defined by their own Player properties, and takes in a simple Board. **As a clarifier, functions that don't use the ":" indicate the return type is void.** Starting with the Player class, each player sets their name, which allows the game to differentiate between the two players. In the case that the player rather plays against a computer, the Player class has an isComputer Boolean, and takes in a difficulty integer for the computer. Finally, a way to keep track of each player's

symbol is to store it in each player object, and to track the win count, each player has their own integer set for it. Onto the game, it takes in a board, 2 players, current player, and resulting winner. Current player helps the game keep track of whose turn it is to play. The class uses takeTurn() function and changeCurrPlayer() to alternate turns between players, as well as keeping an internal check of the winning combinations of Boards that are achievable given the board size. Since TicTacToe can result in ties, the game has a Boolean function that updates whether a player has won, if the game is tied, or if the game is over. Updating the winner updates the wins of the respective player. Finally, the board takes in the size and generates a size x size board. The board also takes in the game as TicTacToe, so if the app ever wants to add future game modes, then the board can be reused by changing the game the board takes in. The board only has barebones functions that add moves and draws/updates the board after each turn. The Board class will most likely be using the data structure of a 2D array to keep track of the characters in each position, and the indexing of each space will make it simpler to check for wins and ties. A place to improve the design is adding separate classes for each of the current classes and using interfaces to allow future additions of features.

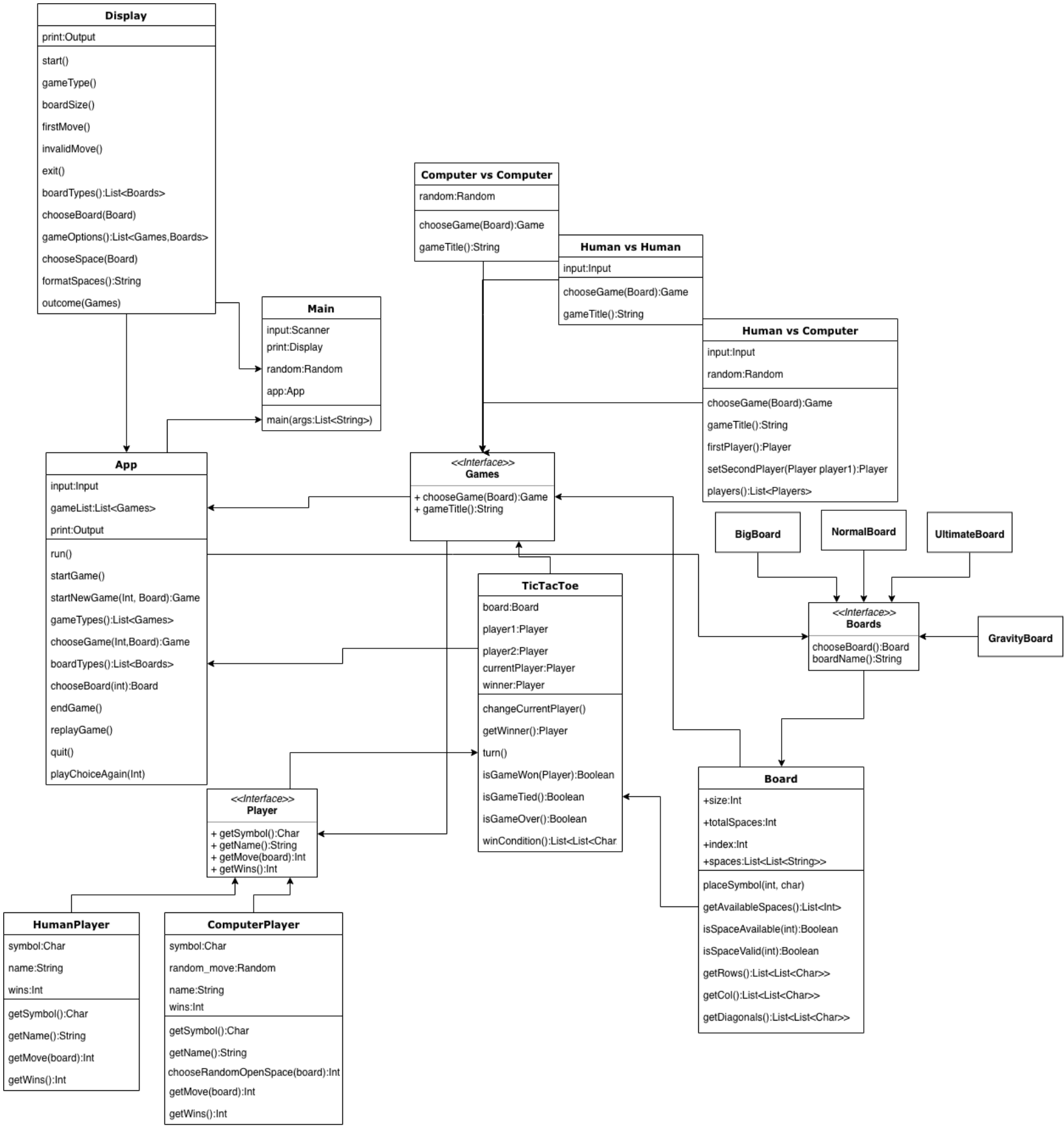


In the second iteration design, I've added a Player interface and clarified more functions for both the Board and TicTacToe classes, as well as adding a different class for both HumanPlayers and ComputerPlayers. The interface for the player is to make sure all players have the basic functions of using symbols, names, and moves. The only difference between the two is allowing the Human player to use their own move, whereas the ComputerPlayer will use the chooseRandomOpenSpace() using the Random class to choose an arbitrary open space on the board. This removes the issue of adding separate levels of difficulty to the ComputerPlayer. As for the Board class, I've expanded on the functions and variables that the board can contain. The board will now keep track of the total amount of spaces on the board and each space's index position. The idea behind this is to allow the board to be a 2D-array of characters, which makes the functions of getting rows, columns, and diagonals possible. Getting the rows, columns, and diagonals, allow the TicTacToe class to use the winCondition() function, which can check through a list of all possible winning combinations of 2D-arrays. The winCondition function seems like it could be improved upon, where rather than using a predetermined list of all possible

winning outcomes, it could be procedurally checked by using the board's rows, columns, and diagonals functions. This could help reduce runtime when checking if a board is correct for each move, by checking the entire list of possible outcomes and comparing them to the current board. Otherwise the Board function also implements checks to see if board moves are legal. isSpaceAvailable is used to check if the symbol placed is not overlapping the other player's already placed symbol, whereas the isSpaceValid is used to determine if the space chosen is within the boundaries of the board. On second thought, this could probably be condensed into a single function, rather than two separate ones.



In this design, I've added interfaces for each of the original 3 classes, as well as added an App class that represents the app the game is running on, and the functions available to the app. Starting with the App class, it takes in inputs, which in this case I have using built in input classes, much like the one calculator application we had to use. This just uses those classes to develop a simple look for the app, as well as act as the center class to which all classes eventually connect to. The app class also uses a gameList<Games> which keeps an internal list of all games, which is used to start games, choose game types, and choose board types. These functions use the interfaces for each of the classes as references to determine which board/game is being played. Adding the functionality of interfaces allows the developer to create more than one type of board, while always reusing the same board class. This applies to the game types, where new game types can be added. For example, I added the different boards like Big Board, Ultimate Board, and Gravity Board. The idea behind the gravity board is that the placed markers now fall to the bottom of the board, making the game more similar to connect four. Although these games use different rules and win conditions, they're all applicable to the board class. I only have the App class connecting to the board interface because I believe that the board dictates the game that is being played, therefore, the choice of the board will determine the win conditions of the TicTacToe games. Nothing changed for the Human Players and Computer Players classes, as I think they're much simpler because they only need to place markers, not implement game types. I've added a game interface so that new games can be added. The only functions that the new interfaces use is executing the type of board/game being used by using the titles of the board names.


Display
print:Output
start()
gameType()
boardSize()
firstMove()
invalidMove()
exit()
boardTypes():List<Boards>
chooseBoard(Board)
gameOptions():List<Games,Boards>
chooseSpace(Board)
formatSpaces():String
outcome(Games)
Computer vs Computer
random:Random
chooseGame(Board):Game
gameTitle():String
Human vs Human
input:Input
chooseGame(Board):Game
gameTitle():String
Human vs Computer
input:Input
random:Random
chooseGame(Board):Game
gameTitle():String
firstPlayer():Player
setSecondPlayer(Player player1):Player
players():List<Players>
Main
input:Scanner
print:Display
random:Random
app:App
main(args:List<String>)
App
input:Input
gameList:List<Games>
print:Output
run()
startGame()
startNewGame(Int, Board):Game
gameTypes():List<Games>
chooseGame(Int,Board):Game
boardTypes():List<Boards>
chooseBoard(int):Board
endGame()
replayGame()
quit()
playChoiceAgain(Int)
<<Interface>>
Games
+ chooseGame(Board):Game
+ gameTitle():String
BigBoard
NormalBoard
UltimateBoard
<<Interface>>
Boards
chooseBoard():Board
boardName():String
GravityBoard
TicTacToe
board:Board
player1:Player
player2:Player
currentPlayer:Player
winner:Player
changeCurrentPlayer()
getWinner():Player
turn()
isGameWon(Player):Boolean
isGameTied():Boolean
isGameOver():Boolean
winCondition():List<List<Char
Board
+size:Int
+totalSpaces:Int
+index:Int
+spaces:List<List<String>>
placeSymbol(int, char)
getAvailableSpaces():List<Int>
isSpaceAvailable(int):Boolean
isSpaceValid(int):Boolean
getRows():List<List<Char>>
getCol():List<List<Char>>
getDiagonals():List<List<Char>>
<<Interface>>
Player
+ getSymbol():Char
+ getName():String
+ getMove(board):Int
+ getWins():Int
HumanPlayer
symbol:Char
name:String
wins:Int
getSymbol():Char
getName():String
getMove(board):Int
getWins():Int
ComputerPlayer
symbol:Char
random_move:Random
name:String
wins:Int
getSymbol():Char
getName():String
chooseRandomOpenSpace(board):Int
getMove(board):Int
getWins():Int

In the final iteration of the program, I've added the preset game types of Human versus Computer combinations, as well as included a new Input class and defined the main function class. The goal of the print class was to print out the board after each update in the game. This resulted in many of the functions return void types, as they would use print statements to show the board. The exceptions were formatSpaces():String, returning a string of properly formatted spaces and symbols in the board, and gameOptions and boardTypes, as these would need to return the list of games and boards that are usable. The display has an arrow connecting it to the App class, as the print statements will directly interact with the App as the game changes. Same goes for the Main class, since input will use a scanner to read the inputs, and print will use the Display class to print out the boards. For games interface, I added the preset of combinations between Human and Computer variations of gameplay. The human vs human class only needs to read inputs because I assume the players will determine who goes first themselves, compared to the Human vs Computer class, that implements both input and random, random being the computer's input, and uses a function to determine the firstPlayer. The firstPlayer() function determines who goes first randomly, and the setSecondPlayer(Player), takes in the first player to assign the second person as the second player. For this reason, the class also implements playerList<Players> which allows it to see the Players interface and determine which classes are being used. Finally, the Computer vs Computer class is entirely random inputs to determine moves. Overall, I think this design uses efficient use of methods and interfaces to create a basis for a functional application, but also allows versatility in adding more functions later on.