CPSC 2150 Project Report

Joel Miller

Requirements Analysis

Functional Requirements:

1. Enumerated list of functional requirements
2. **PLAYER**

1) As a game player, I expect to provide the number of rows, columns, and tokens in a row needed to win the game at the start of each new game. The maximum number of rows and columns for all GameBoards should be 100, and the maximum number needed to win will be 25 for all GameBoards. The minimum number of rows, columns, and tokens in a row needed to win are all 3.
2) As a game player, I should not be able to provide a "number of tokens needed to win" larger than the number of rows or the number of columns.
3) As a game player, I expect the formatting of the displayed board to allow double-digit numbers.
4) As a game player, if I decide to play again, I should be able to change the size and number of tokens needed to win settings from the previous game.
5) As a game player, I expect the program to validate the number of rows, columns, and tokens needed to win that I provide.
6) As a game player, I expect to be able to specify the number of players for each game at the start of each new game. I must enter at least 2 players and at most 10 players.
7) As a game player, after selecting the number of players, I should be able to pick what character will represent me in the game. I can assume that I will always enter just a character. Each player must have their own unique character, and the game should not allow a player to pick a character that is already taken.
8) As a game player, at the start of each new game, I should be able to choose whether I want a fast implementation or a memory-efficient implementation.
9) As a game player, I should be able to change the number of players and their characters if I choose to start a new game.
10) As a game player, I should be able to change the choice of fast or memory-efficient implementation if I start a new game.
11) As a game player, I expect the game to have two different implementations of the IGameBoard interface: a fast implementation that uses a 2D array, and a memory-efficient implementation that uses a Map.
12) As a game player, I expect the GameScreen class to be able to switch between these two implementations easily.
13) As a game player, in the memory-efficient implementation, I expect the game board to be represented using a Map. The key field of the Map should be a Character that represents the player, and the value associated with that player should be a List of BoardPositions that the player occupies on the board.
14) As a game player, I expect that if a player has not placed a token yet, then there will be no key or List in the Map to represent that player. The empty board should not use up any space in memory.
15) As a game player, I expect that when a player adds their first token, the key for that player and a List with that BoardPosition in it should be added. As a player adds more tokens, the BoardPosition should be added onto the List for that player.

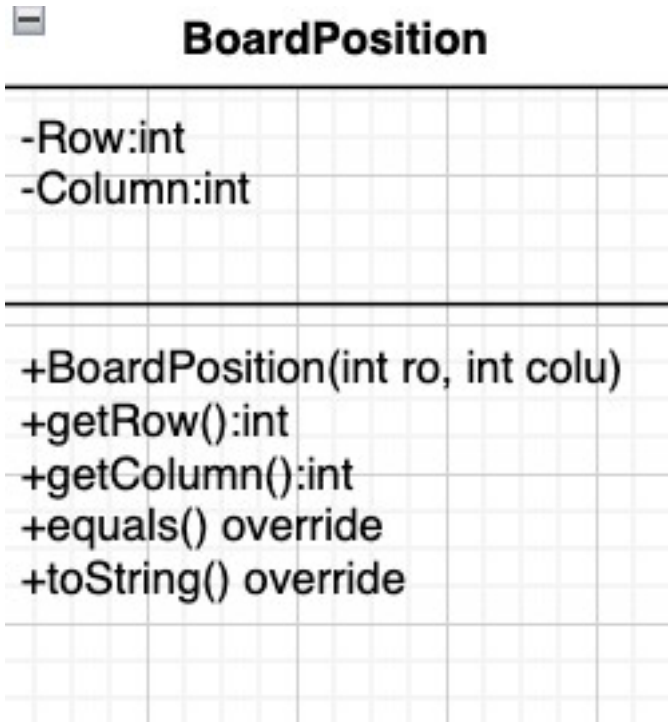
Non-Functional Requirements

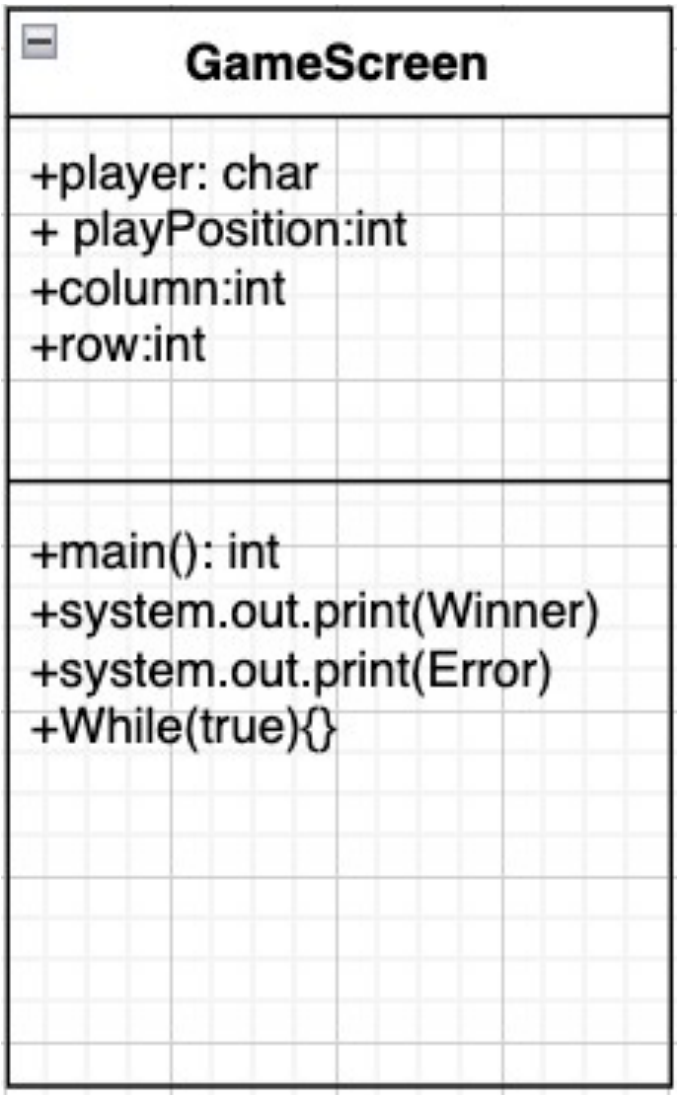
- Enumerated list of Non-functional requirements

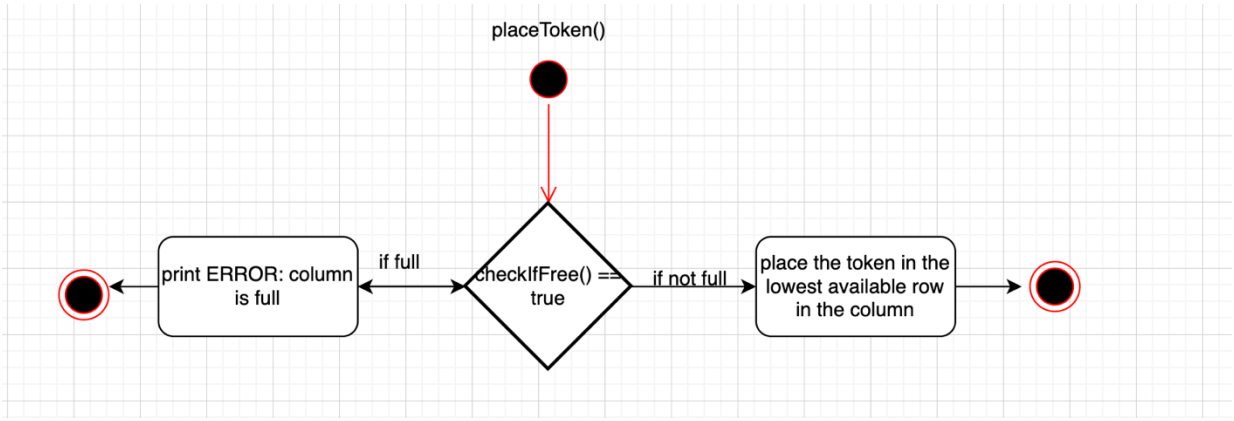
1. The game code must be written in Java.
2. The game must be compatible with a command line interface.
3. The game must have three classes: GameScreen.java, BoardPosition.java, and GameBoard.java.
    - **GameScreen.java:**
        - Must contain the main method.
        - This class will interface with the player{It will alternate the game between players, say whose turn it is, get the column they would like, and place their marker}.
    - **BoardPosition.java:**
        - Will keep track of the row position and column position.
        - Only 1 constructor which takes in an int for row and an int for column.
        - Must also have some getter functions getcolumn and getrow.
    - **GameBoard.java:**
        - All attributes must be private.
        - Functions must be made public.
        - Each position will have a blank character.
        - The constructors for the game boards should take parameters in the order of rows, columns, and the number of tokens needed to win.
        - The maximum number of rows and columns for all GameBoards is 100, and the maximum number needed to win will be 25 for all GameBoards.
        - The minimum number of rows, columns, and tokens in a row needed to win are all 3.
        - GameBoard.java must still use a 2D array to represent its game board, but it's ok if most of the array is ignored for smaller boards.
        - The user should not be able to provide a "number of tokens needed to win" larger than the number of rows or the number of columns.
        - The program must validate the number of rows, columns, and tokens needed to win that is provided by the user.
        - The size and number of tokens needed to win settings can be changed if the users decide to play again.
        - At the start of each new game, the users will be able to specify the number of players for each game. They must enter at least 2 players and, at most, 10 players.

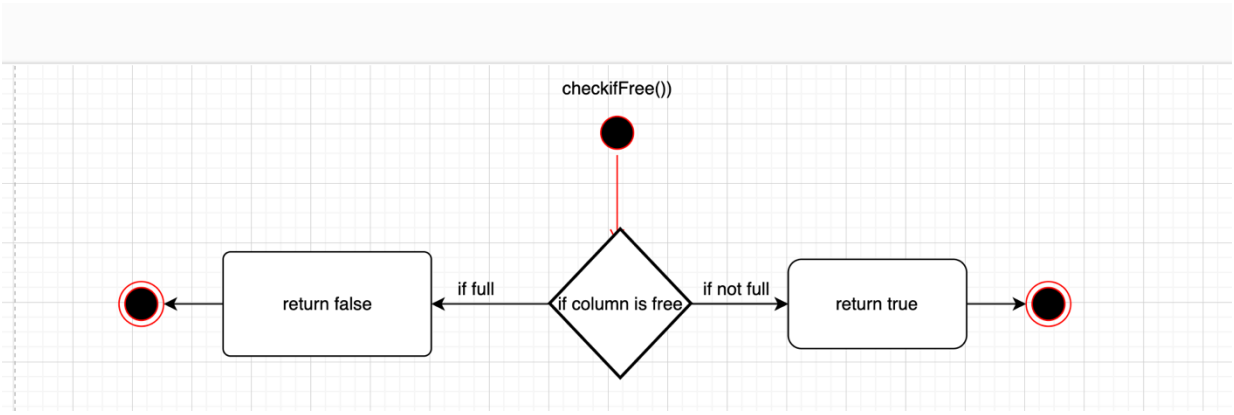
- After selecting the number of players, each player will be able to pick what character will represent them in the game. You can assume they will always enter just a character. Always convert the character to upper case. Each player must have their own unique character, and the game should not allow a player to pick a character that is already taken.
- At the start of each new game, the players can choose whether they want a fast implementation or a memory-efficient implementation. The game must check to make sure they entered either f, F, m, or M as a choice at this option.
- The number of players and their characters can change if they choose to start a new game.
- The choice of fast or memory-efficient implementation can change if they start a new game.
- Our game must now have two different implementations of our IGameBoard interface.
    - We already have our fast implementation, which uses a 2D array. However, we will now check to ensure the implementation is efficient. So, you should not check the entire row if you are looking for the horizontal win; you should check to start at the last position played.
    - Our memory-efficient implementation is detailed below.
- Our GameScreen class should be able to switch between these two implementations easily.

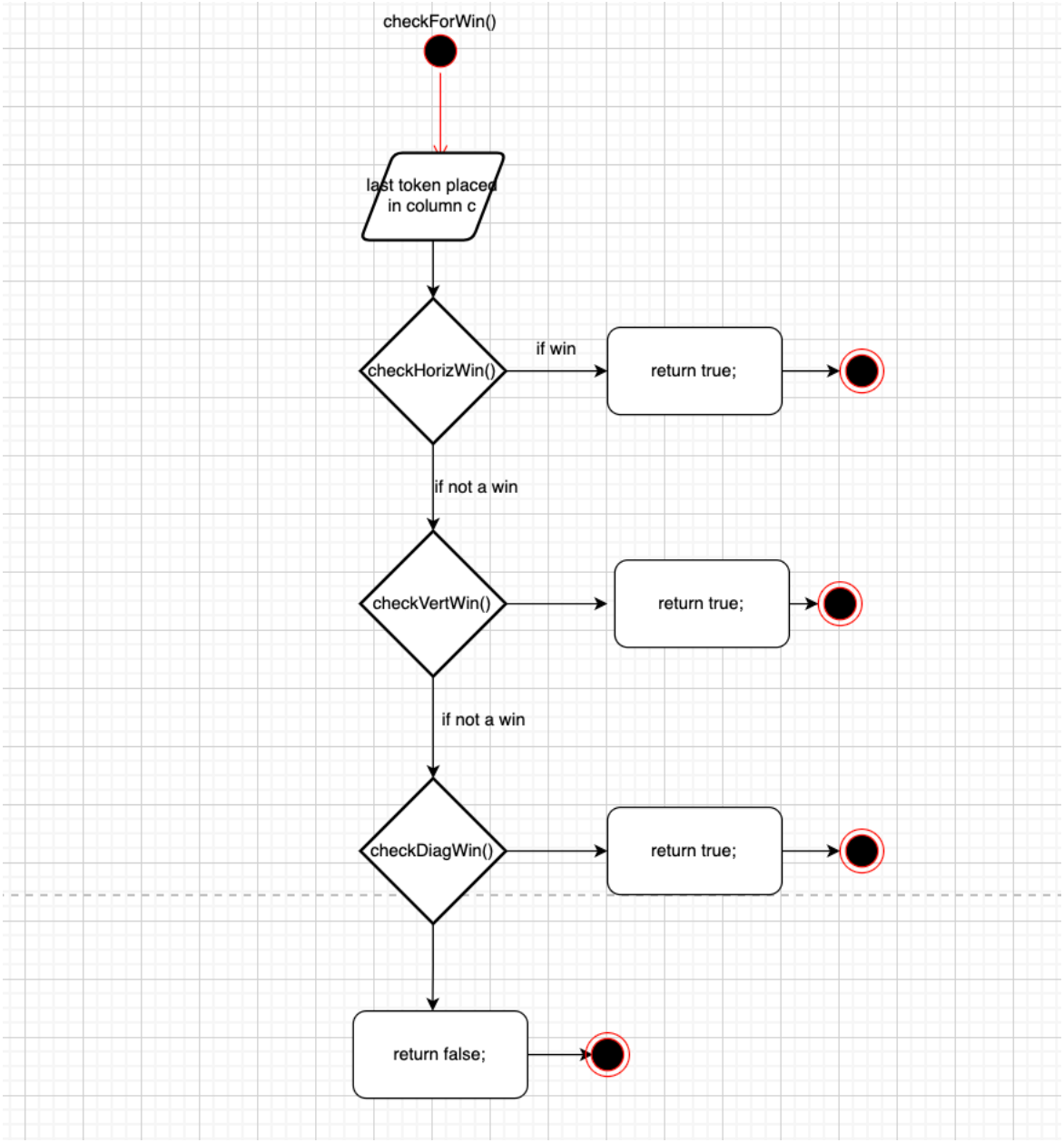
4. **GameBoardMem** class should be created, which will extend AbsGameBoard and implement the IGameBoard interface for memory-efficient implementation.
    - This new implementation will be slower but more memory efficient.
    - To represent the game board, a Map will be used. The key field of the Map will be a Character that represents the player so that each player will get

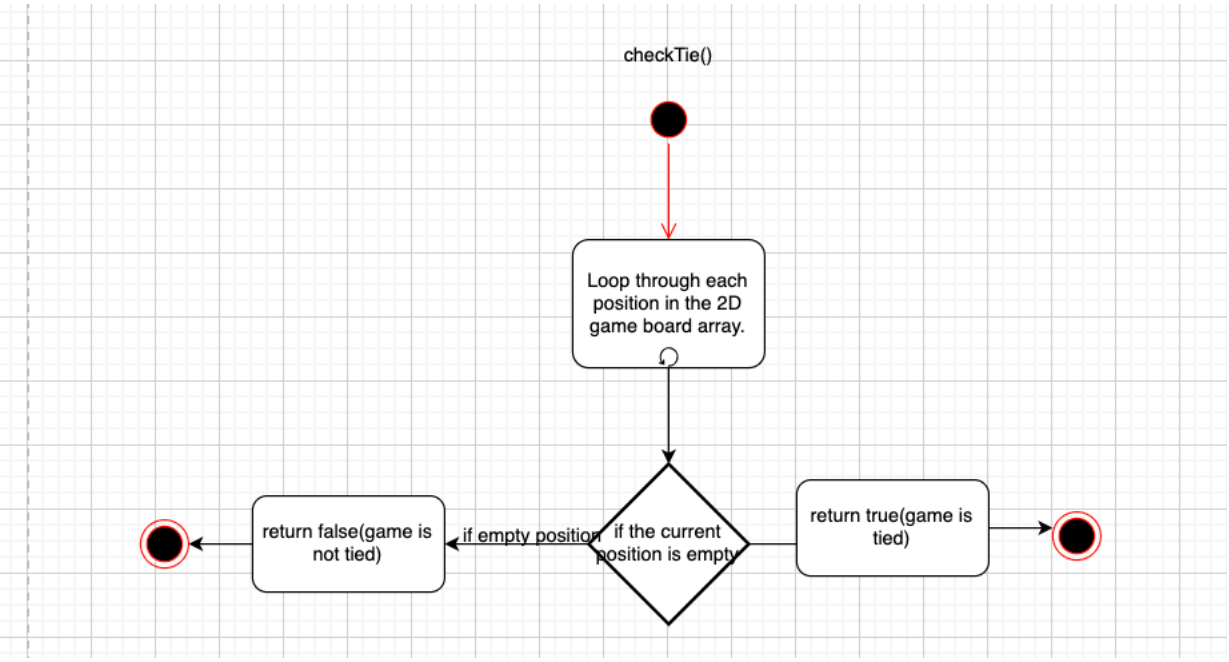

BoardPosition
-Row:int
-Column:int
+BoardPosition(int ro, int colu)
+getRow():int
+getColumn():int
+equals() override
+toString() override

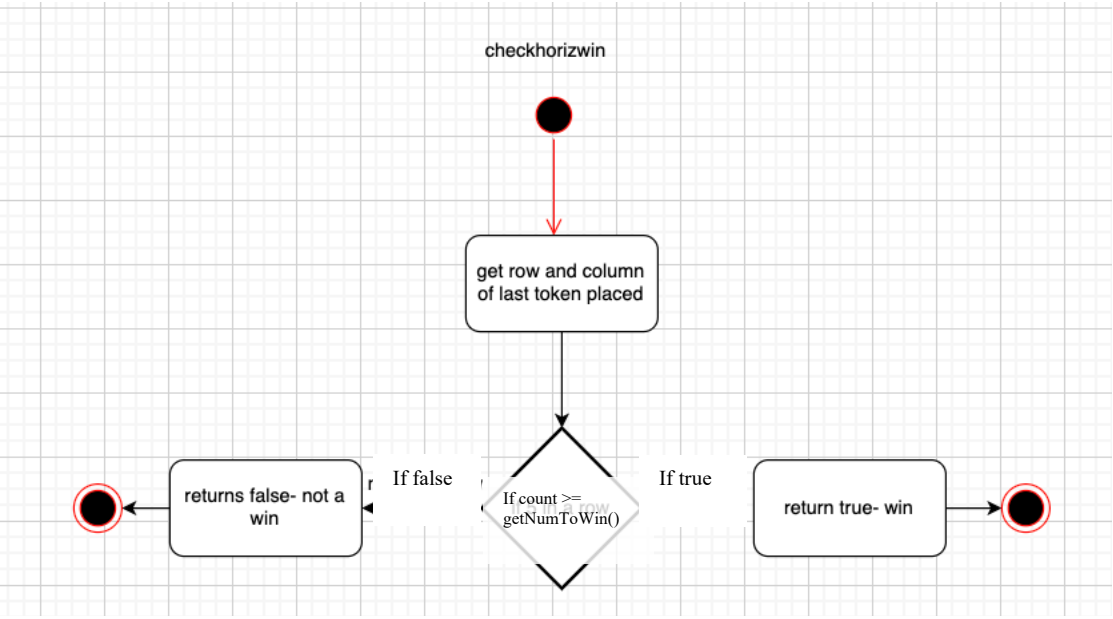

GameScreen
+player: char
+ playPosition:int
+column:int
+row:int
+main(): int
+system.out.print(Winner)
+system.out.print(Error)
+While(true){}

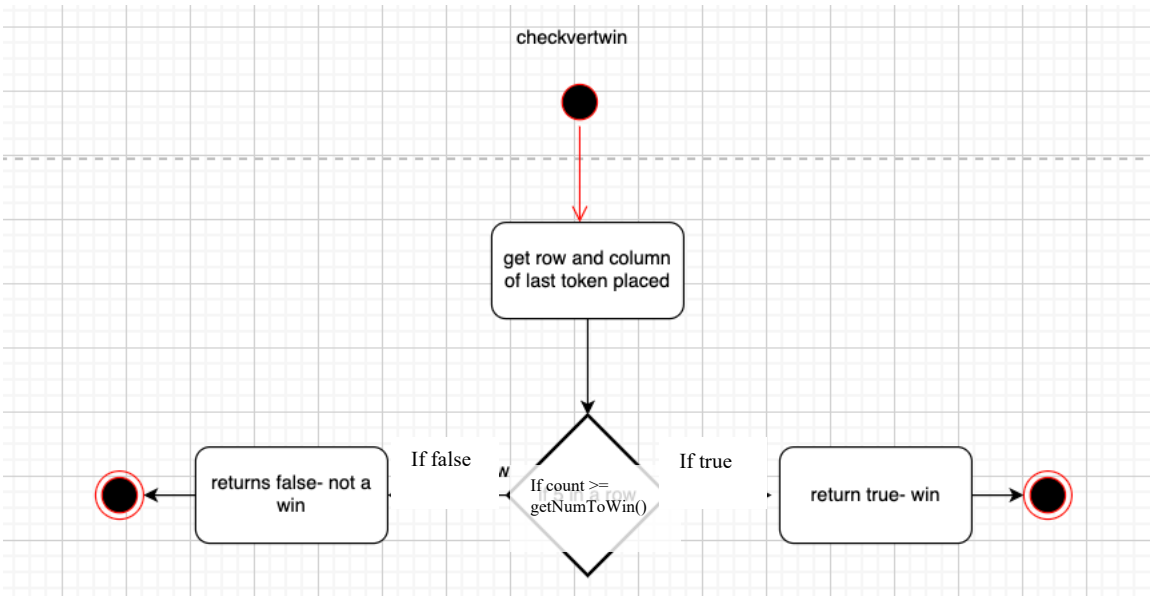

placeToken()
print ERROR: column is full
if full
checkIfFree() == true
if not full
place the token in the lowest available row in the column

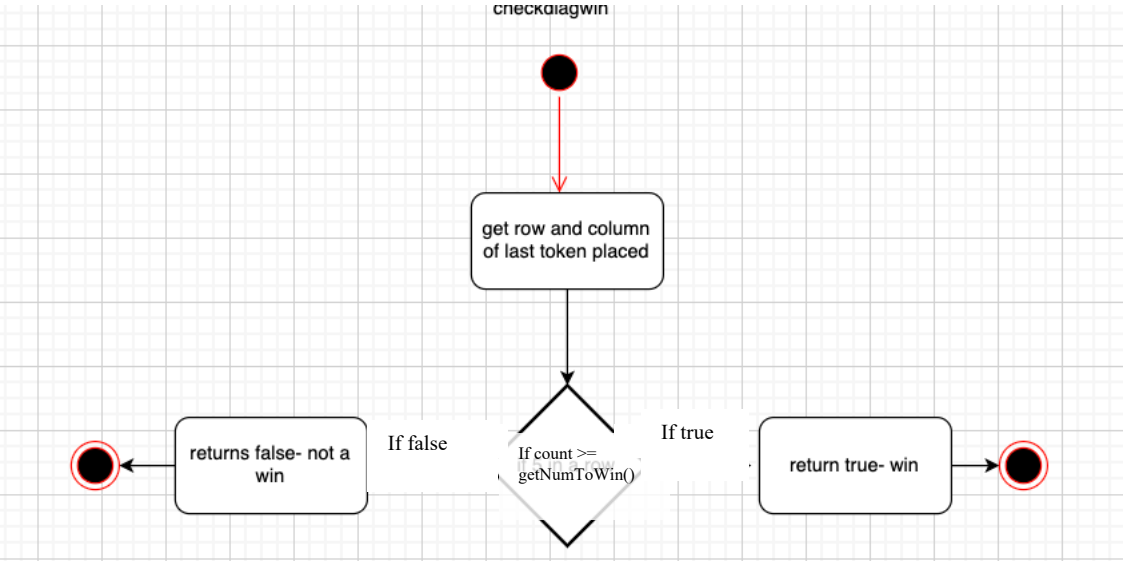

checkifFree())
return false
if full
if column is free
if not full
return true

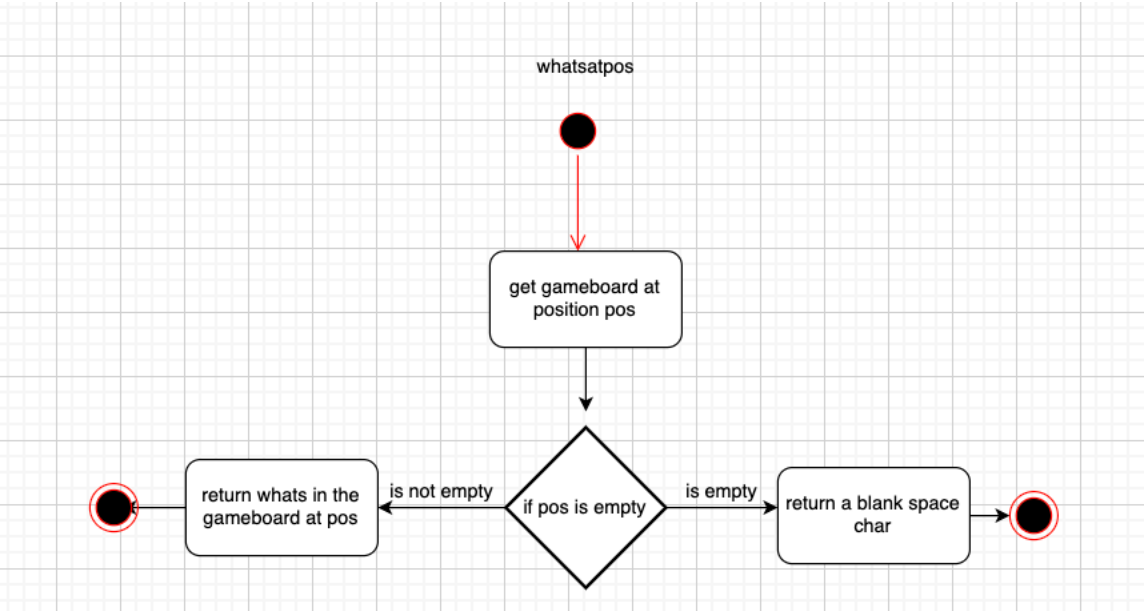

checkForWin()
last token placed in column c
checkHorizWin()
if win
return true;
if not a win
checkVertWin()
return true;
if not a win
checkDiagWin()
return true;
return false;

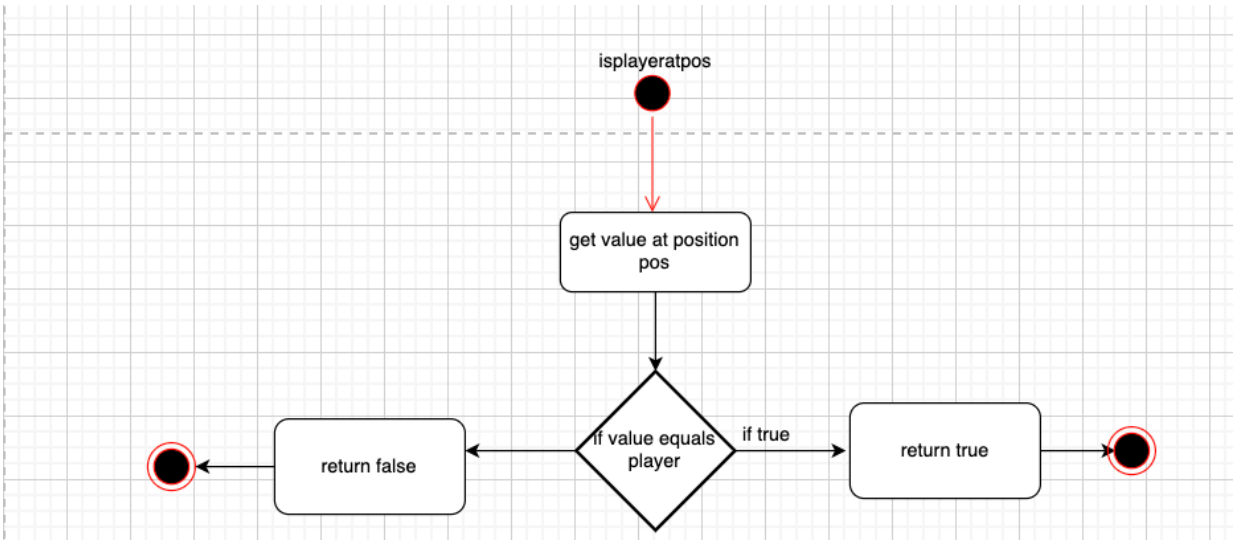

checkTie()
Loop through each position in the 2D game board array.
if the current position is empty
if empty position
return false(game is not tied)
return true(game is tied)

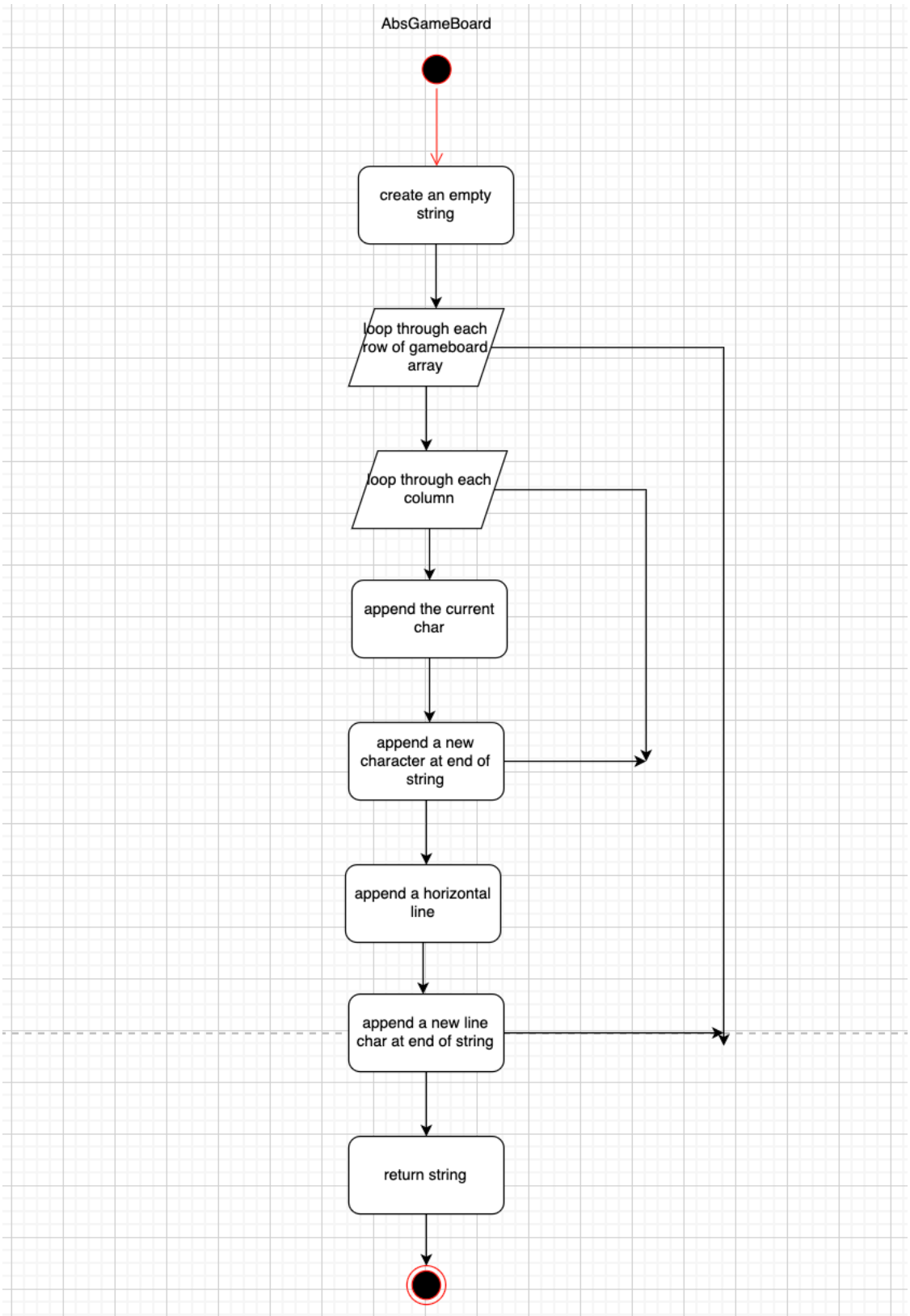

checkhorizwin
get row and column of last token placed
If count >= getNumToWin()
If false
If true
returns false- not a win
return true- win

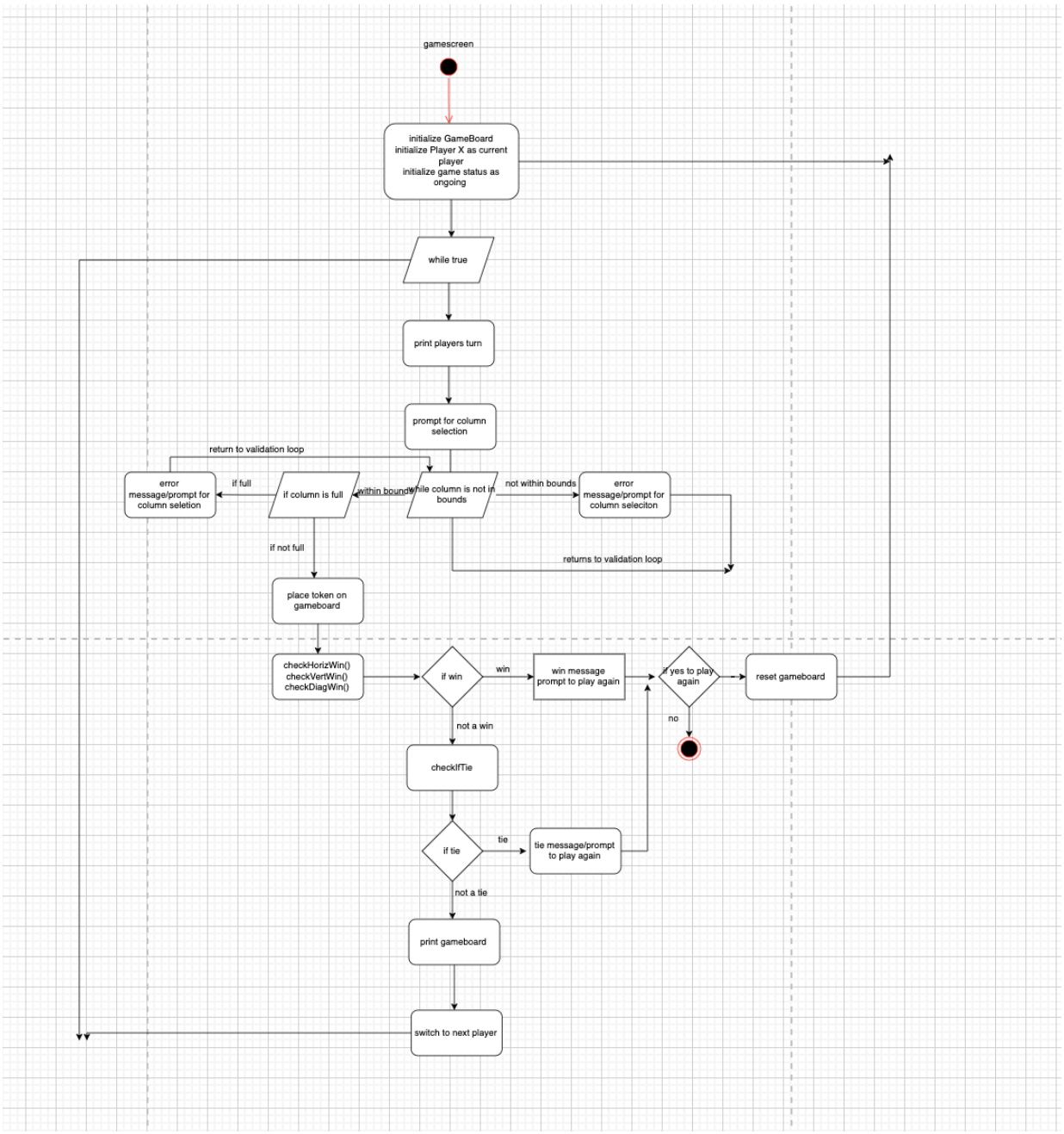

checkvertwin
get row and column of last token placed
If count >= getNumToWin()
If false
returns false- not a win
If true
return true- win

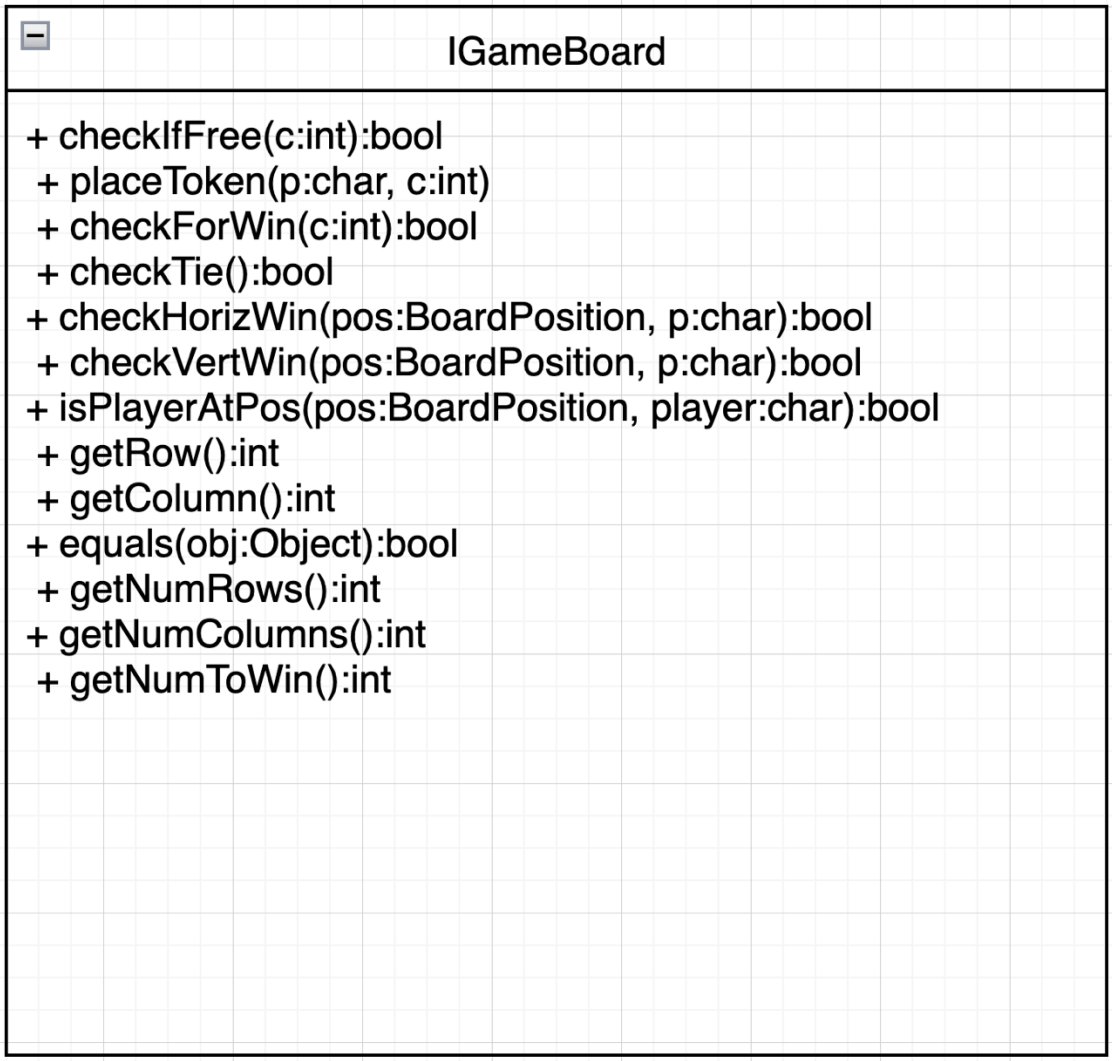

checkdiagwin
get row and column of last token placed
If count >= getNumToWin()
If false
returns false- not a win
If true
return true- win

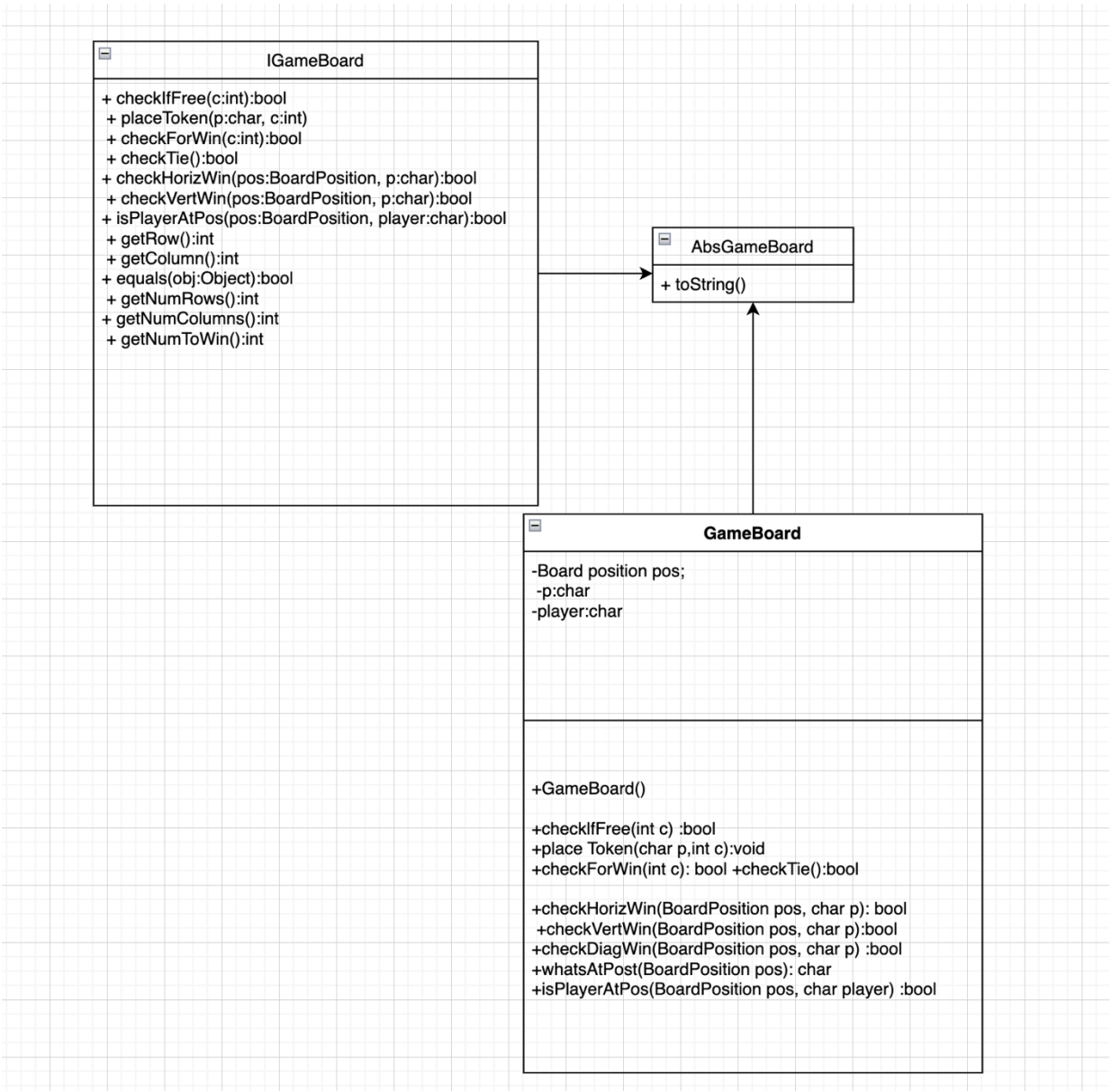

whatsatpos
get gameboard at position pos
if pos is empty
is not empty
return whats in the gameboard at pos
is empty
return a blank space char

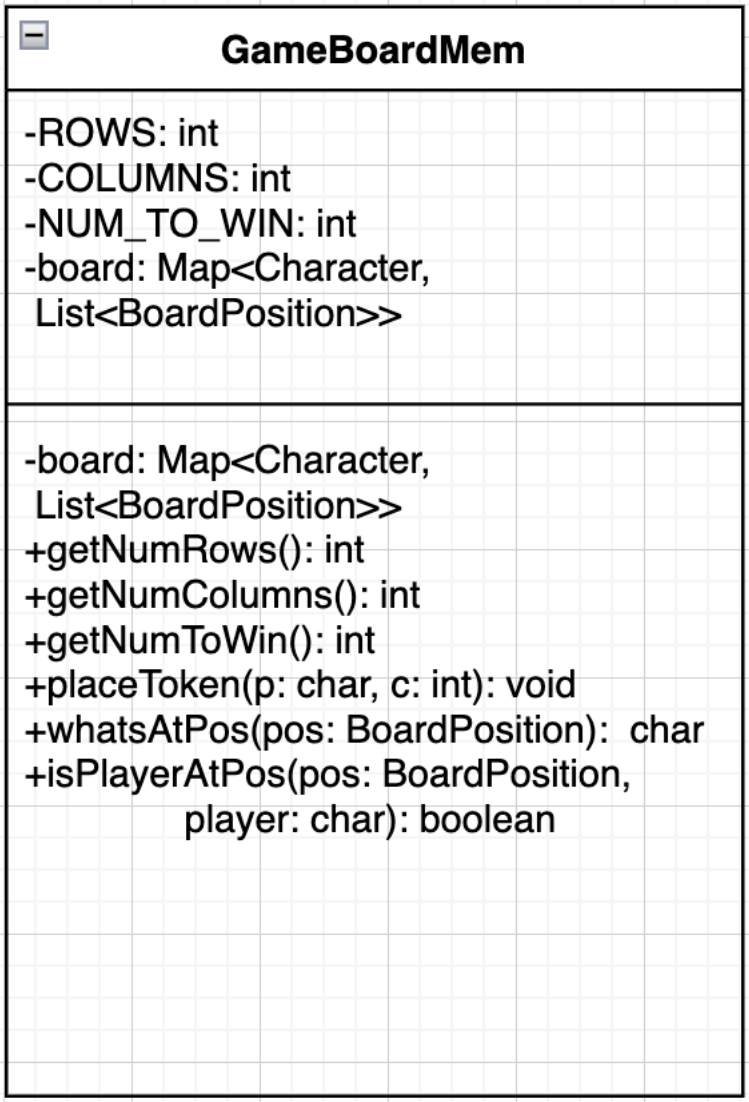

isplayeratpos
get value at position pos
if value equals player
if true
return true
return false


AbsGameBoard
create an empty string
loop through each row of gameboard array
loop through each column
append the current char
append a new character at end of string
append a horizontal line
append a new line char at end of string
return string

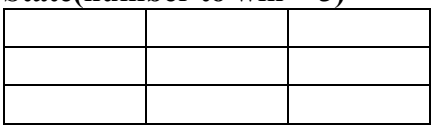

gamescreen
initialize GameBoard
initialize Player X as current player
initialize game status as ongoing
while true
print players turn
prompt for column selection
return to validation loop
error message/prompt for column seletion
if full
if column is full
within bounds
while column is not in bounds
not within bounds
error message/prompt for column seleciton
if not full
returns to validation loop
place token on gameboard
checkHorizWin()
checkVertWin()
checkDiagWin()
if win
win
win message prompt to play again
if yes to play again
reset gameboard
no
not a win
checkIfTie
if tie
tie
tie message/prompt to play again
not a tie
print gameboard
switch to next player

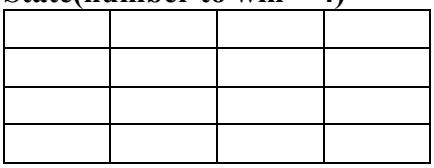

IGameBoard
+ checkIfFree(c:int):bool
+ placeToken(p:char, c:int)
+ checkForWin(c:int):bool
+ checkTie():bool
+ checkHorizWin(pos:BoardPosition, p:char):bool
+ checkVertWin(pos:BoardPosition, p:char):bool
+ isPlayerAtPos(pos:BoardPosition, player:char):bool
+ getRow():int
+ getColumn():int
+ equals(obj:Object):bool
+ getNumRows():int
+ getNumColumns():int
+ getNumToWin():int

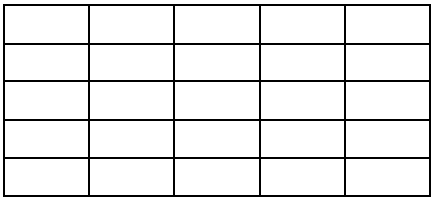

IGameBoard
+ checkIfFree(c:int):bool
+ placeToken(p:char, c:int)
+ checkForWin(c:int):bool
+ checkTie():bool
+ checkHorizWin(pos:BoardPosition, p:char):bool
+ checkVertWin(pos:BoardPosition, p:char):bool
+ isPlayerAtPos(pos:BoardPosition, player:char):bool
+ getRow():int
+ getColumn():int
+ equals(obj:Object):bool
+ getNumRows():int
+ getNumColumns():int
+ getNumToWin():int
AbsGameBoard
+ toString()
GameBoard
-Board position pos;
-p:char
-player:char
+GameBoard()
+checkIfFree(int c) :bool
+place Token(char p,int c):void
+checkForWin(int c): bool +checkTie():bool
+checkHorizWin(BoardPosition pos, char p): bool
+checkVertWin(BoardPosition pos, char p):bool
+checkDiagWin(BoardPosition pos, char p) :bool
+whatsAtPost(BoardPosition pos): char
+isPlayerAtPos(BoardPosition pos, char player) :bool


GameBoardMem
-ROWS: int
-COLUMNS: int
-NUM_TO_WIN: int
-board: Map<Character,
List<BoardPosition>>
-board: Map<Character,
List<BoardPosition>>
+getNumRows(): int
+getNumColumns(): int
+getNumToWin(): int
+placeToken(p: char, c: int): void
+whatsAtPos(pos: BoardPosition): char
+isPlayerAtPos(pos: BoardPosition,
player: char): boolean

`Constructor GameBoard() GameboardMem()`

Input:	Output:	Reason:
numRows = 3 **numColumns = 3** **numToWin = 3**	**State(number to win = 3)**  **Function Name:** testConstructor_01	This test case is unique because it tests the constructor for creating a standard 3x3 board with a winning condition of 3 tokens in a row.

Input:	Output:	Reason:
numRows = 4 **numColumns = 4** **numToWin = 4**	**State(number to win = 4)**  **Function Name:** testConstructor_02	This test case is unique because it tests the constructor for creating a 4x4 game board with a winning condition of 4 tokens in a row. This tests the constructor's ability to handle different board sizes.

Input:	Output:	Reason:
numRows = 5 **numColumns = 5** **numToWin = 3**	**State(number to win = 3)**  **Function Name:** testConstructor_03	This test case is unique because it tests the constructor for creating a 5x5 game board with a winning condition of only 3 tokens in a row. This tests the constructor's ability to handle different winning conditions in relation to the board size.

`boolean checkIfFree(int c)`

Input:	**Output:**	**Reason:**
State(number to win = 4) **o** **x** **o** **x** **o** **x** **o** **o** x x x o o o x x c = 0;	checkIfFree = false state of the board is unchanged **Function Name:** testCheckIfFree_01	This test case is unique because the game board is completely full, and the function should check if there is any available space in the specified column (0). Since the column is full, the function should return false.

Input:	**Output:**	**Reason:**
State(number to win = 4) **x** **o** **x** **x** **x** **o** **o** x x x o o o x x c = 1;	checkIfFree = true state of the board is unchanged **Function Name:** testCheckIfFree_02	This test case is unique because the game board has an empty spot in the specified column (1). The function should check if there is any available space in that column. Since the column has an empty spot, the function should return true.

Input:	**Output:**	**Reason:**
State(number to win = 4) **x** **o** **x** **x** **o** **o** x x x o o o x x c = 0;	checkIfFree = true state of the board is unchanged **Function Name:** testCheckIfFree_03	This test case is unique because the game board has a partially filled specified column (0) with only one 'x' marker. The function should check if there is any available space in that column. Since the column has empty spots, the function should return true.

`boolean checkHorizWin(BoardPosition pos, char p)`

Input:	**Output:**	**Reason:**
State:(number to win = 3) x x x o o x x pos.getRow = 1 pos.getCol = 2 p = 'x'	checkHorizWin = true state of the board is unchanged **Function Name:** testCheckHorizWin_01	This test case is unique and distinct because the last x was placed at the end of the string so the function needs to be able to count the x's to the left

Input:	**Output:**	**Reason:**
State:(number to win = 4) o o o o x x x o pos.getRow = 1 pos.getCol = 2 p = 'o'	checkHorizWin = true state of the board is unchanged **Function Name:** testCheckHorizWin_02	This test case is unique and distinct because the last o was placed in the middle of the string of 4 consecutive o's as opposed to on the end, so the function needs to count o's on the right and left

Input:	**Output:**	**Reason:**
State:(number to win = 4) **o o o** **x x x o x** pos.getRow = 0 pos.getCol = 4 p = 'x'	checkHorizWin = false state of the board is unchanged **Function Name:** testCheckHorizWin_03	This test case is unique because the 'x' has tokens to the left, but they are not consecutive. The function needs to count the number of consecutive 'x's to determine if there is a win condition. In this case, there are only three consecutive 'x's and o's, so there is no win condition.

Input:	Output:	Reason:
State:(number to win = 4) pos.getRow = 0 pos.getCol = 3 p = 'x'	checkHorizWin = true state of the board is unchanged **Function Name:** testCheckHorizWin_04	This test case is unique because the 'x' just placed is in the middle of a string of four consecutive 'x's, and the function needs to check for a win condition by counting the number of 'x's to the left and right of the newly placed 'x'.

o		**o**			
o	**o**	**x**	**x**	**x**	**x**

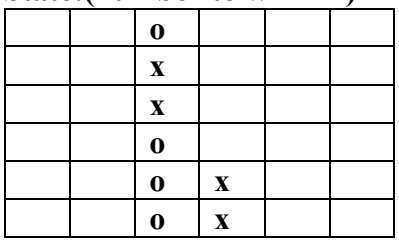
`boolean checkVertWin(BoardPosition pos, char p)`

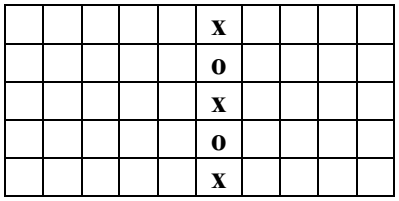
Input:	Output:	Reason:
State:(number to win = 3) pos.getRow = 2 pos.getCol = 1 p = 'x'	checkVertWin = true state of the board is unchanged **Function Name:** testCheckVertWin_01	This test case is unique because the 'x' just placed is part of a string of three consecutive 'x's in the second column from the left, and the function needs to count the number of 'x's below to determine a win condition. Since there are three 'x's in a row vertically, **checkVertWin** should return **true** for a win condition.

	x	
o	**x**	
o	**x**	

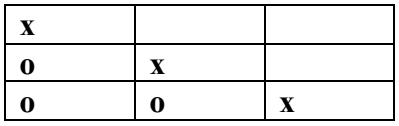
Input:	Output:	Reason:
State:(number to win = 5) pos.getRow = 4 pos.getCol = 5 p = 'x'	checkVertWin = true state of the board is unchanged **Function Name:** testCheckVertWin_02	This test case is unique because the 'x' just placed is part of a string of five consecutive 'x's in a column, and the function needs to count the number of consecutive 'x's to determine a win condition. In this case, there are five consecutive 'x's so checkVertWin should return true.

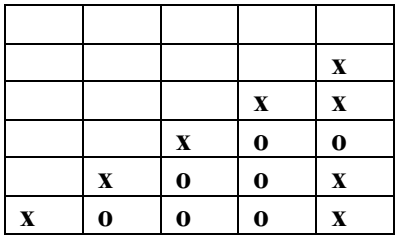
					x	
					x	
s	**s**	**f**	**o**	**o**	**x**	
o	**f**	**f**	**f**	**s**	**x**	
o	**f**	**o**	**s**	**s**	**x**	**o**

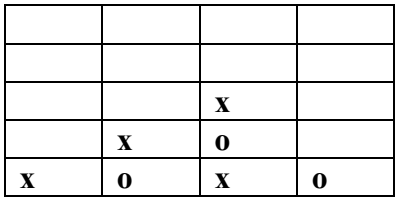
Input:	Output:	Reason:
State:(number to win = 4)  pos.getRow = 5 pos.getCol = 2 p = ‘o’	checkVertWin = false state of the board is unchanged **Function Name:** testCheckVertWin_03	This test case is unique because the 'o' just placed is not part of a string of four consecutive 'o's, and the function should return **false** for a win condition. Although there are four 'o's in the column, they are not in a consecutive pattern, so **checkVertWin** should return **false**.

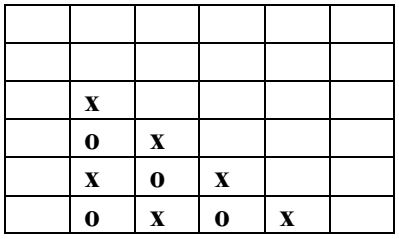
Input:	Output:	Reason:
State(number to win = 5)  pos.getRow = 4 pos.getCol = 5 p = ‘x’	checkVertWin = false state of the board is unchanged **Function Name:** testCheckVertWin_04	This test case is unique because the 'x' just placed is at the top of a string of five consecutive tokens and the function needs to check for a win condition by counting the number of 'x's. Since there are no 5 consecutive tokens checkVertWin should return false.

```
public default boolean checkDiagWin(BoardPosition pos, char p)
```

Input:	Output:	Reason:
State(number to win = 3)  pos.getRow = 3 pos.getCol = 1 p = ‘x’	checkDiagWin = true state of the board is unchanged **Function Name:** testCheckDiagWin _01	This test case is unique because the 'x' just placed is in the middle of a string of three consecutive 'x's in a diagonal direction. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since there are three consecutive 'x's in the top-left to bottom-right diagonal direction, **checkDiagWin** should **return true.**

Input:	Output:	Reason:
State(number to win = 5)  pos.getRow = 2 pos.getCol = 2 p = 'x'	checkDiagWin = true state of the board is unchanged **Function Name:** testCheckDiagWin _02	This test case is unique because the 'x' just placed is in the middle of a diagonal string of five consecutive tokens, and the function needs to check for a win condition by counting the number of 'x's in both diagonal directions. There are five consecutive 'x's in the top-left to bottom-right diagonal direction, so checkDiagWin should return true

Input:	Output:	Reason:
State(number to win = 3)  pos.getRow = 0 pos.getCol = 0 p = 'x'	checkDiagWin = true state of the board is unchanged **Function Name:** testCheckDiagWin _03	This test case is unique because the 'x' just placed is at the beginning of a string of three consecutive 'x's in a diagonal direction. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since there are three consecutive 'x's in the top-right to bottom-left diagonal direction, checkDiagWin should return true

Input:	Output:	Reason:
State(number to win = 4)  pos.getRow = 3 pos.getCol = 1 p = 'x'	checkDiagWin = true state of the board is unchanged **Function Name:** testCheckDiagWin _04	This test case is unique because the 'x' just placed is at the beginning of a string of four consecutive 'x's in a diagonal direction. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since there are four consecutive 'x's in the top-right to bottom-left diagonal direction, checkDiagWin should return true.

Input: **State(number to win = 4)** 	**Output:** checkDiagWin = false state of the board is unchanged **Function Name:** testCheckDiagWin _05	**Reason:** This test case is unique because the 'x' just placed is part of a diagonal string of 'x's, but the number of consecutive 'x's is not enough to meet the win condition. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since there are only three consecutive 'x's in the top-right to bottom-left diagonal direction and the number to win is 4, checkDiagWin should return false.

						x
					o	x
				o	o	x
			o	x	x	x
	o	x	o	x	x	o

pos.getRow = 0
pos.getCol = 1
p = ‘x’

Input: **State(number to win = 5)** 	**Output:** checkDiagWin = false state of the board is unchanged **Function Name:** testCheckDiagWin _06	**Reason:** This test case is unique because the 'x' just placed is part of a diagonal string of 'x's, but the number of consecutive 'x's is not enough to meet the win condition. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since there are only four consecutive 'x's in the top-left to bottom-right diagonal direction and the number to win is 5, checkDiagWin should return false

			x					
			o	x				
			o	o	x			
		o	x	o	o	x		

pos.getRow = 3
pos.getCol = 3
p = ‘x’

Input:	Output:	Reason:
State(number to win = 7) |						| |---|---|---|---|---|---|---| |						| |						| |						| |			**o**			| |		**o**	**o**	**x**		| |	**o**	**x**	**o**	**x**	**x**	| | **o**	**x**	**o**	**x**	**o**	**o**	**x** | pos.getRow = 3 pos.getCol = 3 p = 'x'	checkDiagWin = false state of the board is unchanged **Function Name:** testCheckDiagWin _07	This test case is unique because the 'x' just placed is part of a diagonal string of 'x's, but the number of consecutive 'x's is not enough to meet the win condition. The function needs to check for a win condition by counting the number of 'x's in both diagonal directions (top-left to bottom-right and top-right to bottom-left). Since the longest diagonal string of consecutive 'x's is four, and the number to win is 7, checkDiagWin should return false.

`boolean checkTie()`

Input:	Output:	Reason:
State(number to win = 4) |				| |---|---|---|---| |			| |			| |		x	| | o	o	x	x		checkTie = false state of the board is unchanged **Function Name:** testCheckTie_01	This test case is unique because the game board has several empty spots, and it is easy to identify that the game is not tied. The function should return false as there are still available moves.

Input:	Output:	Reason:
State(number to win = 4) |	**x**	**o**	**x** | | **o**	**o**	**o**	**x** | | x	x	x	o | | o	o	x	x |	checkTie = false state of the board is unchanged **Function Name:** testCheckTie_02	This test case is unique because the game board appears to be almost full, with only one empty spot. However, since there is still an available move, the function should return false, indicating that the game is not tied.

Input:	Output:	Reason:
State(number to win = 4) | **o**	**x**	**o**	**x** | | **o**	**x**	**o**	**o** | | x	x	x	o | | o	o	x	x |	checkTie = true state of the board is unchanged **Function Name:** testCheckTie_03	This test case is unique because the game board is completely full, and there are no available moves left. The function should return true, indicating that the game is tied.

Input:	Output:	Reason:
State(number to win = 4) | **x**			| | **o**	**x**	**o**	| | x	o	x	o | | o	x	o	o |	checkTie = false state of the board is unchanged **Function Name:** testCheckTie_04	This test case is unique because the game board has an irregular distribution of 'x' and 'o' markers, but there are still empty spots available. The function should return false, indicating that the game is not tied.

```
char whatsAtPos(BoardPosition pos)
```

Input:	Output:	Reason:
State(number to win = 4) **x** | **o** | **x** | **o** **o** | **x** | **o** | **x** x | x | o | o o | x | o | x BoardPosition (row = 0, col = 0)	whatsAtPos = 'o' state of the board is unchanged **Function Name:** testCheckWhatsAtPos_01	This test case is unique because the function should return the marker at the specified BoardPosition (row = 0, col = 0). In this case, the marker is 'o', so the function should return 'o'.

Input:	Output:	Reason:
State(number to win = 4) **x** | **o** | **x** | **o** **o** | **x** | **o** | **x** x | o | x | o o | x | o | x BoardPosition (row = 2, col = 1)	whatsAtPos = 'x' state of the board is unchanged **Function Name:** testCheckWhatsAtPos_02	This test case is unique because the function should return the marker at the specified BoardPosition (row = 2, col = 1). In this case, the marker is 'x', so the function should return 'x'.

Input:	Output:	Reason:
State(number to win = 4) **x** | **o** | | **o** **o** | **o** | **o** | **x** x | o | x | o o | x | o | x BoardPosition (row = 3 col = 2)	whatsAtPos = ' ' state of the board is unchanged **Function Name:** testCheckWhatsAtPos_03	This test case is unique because the function should return the marker at the specified BoardPosition (row = 3, col = 2). In this case, there is no marker, so the function should return ' ' (empty space).

Input:	Output:	Reason:
State(number to win = 4) **x** | **o** | | **o** **o** | **x** | | **x** x | o | | o o | x | | x BoardPosition (row = 0, col = 2)	whatsAtPos = ' ' state of the board is unchanged **Function Name:** testCheckWhatsAtPos_04	This test case is unique because the function should return the marker at the specified BoardPosition (row = 0, col = 2). In this case, the marker is ' ', so the function should return ' '(empty space) This is also a unique case because of how the whole board is full except one column.

Input:	Output:	Reason:
State(number to win = 4) **x** | | | | **o** | | | | x | | | | o | | | | BoardPosition (row = 3, col = 3)	whatsAtPos = ' ' state of the board is unchanged **Function Name:** testCheckWhatsAtPos_05	This test case is unique because the function should return the marker at the specified BoardPosition (row = 3, col = 3). In this case, the marker is ' ', so the function should return ' '(empty space). This is also a unique case because of how the whole board is empty except one column.

`boolean isPlayerAtPos(BoardPosition pos, char player)`

Input:	**Output:**	**Reason:**
State(number to win = 4) **x** | **o** | **x** | **o** **o** | **x** | **o** | **x** x | o | x | o o | x | o | x BoardPosition (row = 0, col = 0), player = 'o'	isPlayerAtPos = true state of the board is unchanged **Function Name:** testCheckIsPlayerAtPos_01	This test case is unique because the function should return true if the specified player ('o') is at the specified BoardPosition (row = 0, col = 0). In this case, the player 'o' is at the given position, so the function should return true.

Input:	**Output:**	**Reason:**
State(number to win = 4) **x** | **o** | **x** | **o** **o** | **x** | **o** | **x** x | o | x | o o | x | o | x BoardPosition (row = 2, col = 1), player = 'x'	isPlayerAtPos = true state of the board is unchanged **Function Name:** testCheckIsPlayerAtPos_02	This test case is unique because the function should return true if the specified player ('x') is at the specified BoardPosition (row = 2, col = 1). In this case, the player 'x' is at the given position, so the function should return true.

Input:	**Output:**	**Reason:**
State(number to win = 4) | **o** | **o** | **x** **o** | **x** | **o** | **x** x | o | x | o o | x | o | x BoardPosition (row = 3, col = 0), player = 'o'	isPlayerAtPos = false state of the board is unchanged **Function Name:** testCheckIsPlayerAtPos_03	This test case is unique because the function should return false if the specified player ('o') is not at the specified BoardPosition (row = 3, col = 0). In this case, there is no marker at the given position, so the function should return false.

Input:	Output:	Reason:
State(number to win = 4) |			| |---|---|---|---| |	**x**	**o**	| | x | x | o | | | o | x | o | x | BoardPosition (row = 1, col = 2), player = 'x'	isPlayerAtPos = false state of the board is unchanged **Function Name:** testCheckIsPlayerAtPos_04	This test case is unique because the function should return false if the specified player ('x') is not at the specified BoardPosition (row = 1, col = 2). In this case, the player 'o' is at the given position, so the function should return false.

Input:	Output:	Reason:
State(number to win = 4) |			| |---|---|---|---| |		**x**	| |		**x**	| |		x	| |		o	| BoardPosition (row = 3, col = 3), player = 'o'	isPlayerAtPos = false state of the board is unchanged **Function Name:** testCheckIsPlayerAtPos_05	This test case is unique because the function should return false if the specified player ('o') is not at the specified BoardPosition (row = 3, col = 3). In this case, the player 'x' is at the given position, so the function should return false.

`void placeToken(char p, int c)`

Input:	Output:	Reason:
State: |					| |---|---|---|---|---| |				| |				| |				| |		**o**		| | **x** | **o** | **x** | | | p = 'x' c = 1	State: |				| |---|---|---|---|---| |				| |				| |				| |	x | o | | | | x | o | x | | | **Function Name:** testPlace_token_01	This test case is unique because I am placing my marker in a column that was not empty, and also was not close to be being full.

Input:	**Output:**	**Reason:**
State: |					| |---|---|---|---|---| | **x** |			| | **o** |			| | **o** |			| | **o** |			| p = 'x' c = 0	State: | x |			| |---|---|---|---|---| | x |			| | o |			| | o |			| | o |			| **Function Name:** testPlace_token_02	This test case is unique because it involves placing the 'x' marker in a column that was not empty, and also was not close to being full. Additionally, this placement completely fills row 1 of the game board, not allowing any more tokens to be placed in that row.

Input:	**Output:**	**Reason:**
State: |					| |---|---|---|---|---| |				| |				| |				| |				| | **o** | **x** | | **x** | **o** | p = 'x' c = 2	State: |				| |---|---|---|---|---| |				| |				| |				| |				| | **o** | **x** | **x** | **x** | **o** | **Function Name:** testPlace_token_03	This test case is unique because I am placing my marker in a column that was empty, and also was not close to be being full and also completely filling row 0 of the gameboard not allowing anymore tokens to be placed on that row.

Input:	**Output:**	**Reason:**
State: | **x** |				| |---|---|---|---|---| | **x** | **x** |		| | **o** | **o** |		| | **o** | **x** | **x** | **x** | | | **o** | **x** | **x** | **x** | **x** | p = 'x' c = 2	State: | **x** |			| |---|---|---|---|---| | **x** | **x** |		| | **o** | **o** | **x** |	| | **o** | **x** | **x** | **x** | | | **o** | **x** | **x** | **x** | **x** | **Function Name:** testPlace_token_04	This test case is unique because it involves placing the 'x' marker in a column that was not empty and has a mix of 'x' and 'o' markers. The board is also not close to being full, and the placement results in a more balanced distribution of tokens across the rows, allowing for more strategic play in future moves. Also this completes a diagonal row of x's

<table>
<tr>
<td>Input:

State:
<table>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td></td></tr>
<tr><td>o</td><td>x</td><td>o</td><td>x</td><td>o</td></tr>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>o</td><td>x</td><td>o</td><td>x</td><td>o</td></tr>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
</table>

p = ‘x’
c = 4</td>
<td>Output:

State:
<table>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>o</td><td>x</td><td>o</td><td>x</td><td>o</td></tr>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>o</td><td>x</td><td>o</td><td>x</td><td>o</td></tr>
<tr><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
</table>

Function Name:
testPlace_token_05</td>
<td>Reason:

This test case is unique because it involves placing the 'x' marker in a column that is almost full, with a mix of 'x' and 'o' markers, and has only one spot left for the token to be placed. After placing the 'x' token, the game board becomes completely full.</td>
</tr>
</table>