Julio Reyes

Professor Murali Sitaraman

CPSC 2150

February 7th 2021

Project Report

Requirements Analysis

Functional Requirements

1) As a user, I can view the current board, so that I know how the game is going.
2) As a user, I can enter a column position, so that I can place my piece in a specific position.
3) As a user, I can view the results of the game, so that I know who has won.
4) As a user, I can select how many rows I want my gameboard to have
5) As a user, I can select how many columns I want my gameboard to have
6) As a user, I can select how many players I want to play in the game
7) As a user, I can select my own unique player token
8) As a user, I can select whether I want a fast vs memory efficient game
9) As a user, I can be notified when I try to place my piece in an already filled column, so that I know an illegal move has been made.

10) As a user, I can have the option to play again after finishing a game, so that I can start a new game if I want to.
11) As a user, I can have the option to end the program after the game has finished, so that I can stop playing if I want to.
12) As a user, I can see when the game has ended in a tie, so that I know that no one has won.
13) As a user, I can see when a player has won due to placing five tokens in a row horizontally
14) As a user, I can see when a player has won due to placing five tokens in a row vertically
15) As a user, I can see when a player has won due to placing five tokens in a row diagonally
16) As a user, I can be given the option to place my token after my opponent's turn
17) As a user, I can be notified whenever a column I've chosen is already full.
18) As a user, I can be notified whenever I make a selection that is out of the bounds of the game board.
19) As a user, I can be notified if I choose a token that has already been chosen by another player

Non Functional Requirements

1) Must run on the Clemson School of Computing server.
2) Must be in Java.
3) Need to create UML class diagrams.
4) Need to create UML activity diagrams.

5) Need to create contracts for each method in my classes.
6) Create javadoc comments, specifying parameters, invariants, etc.
7) Game Board must be of user specified size
8) The bottom left of the board has coordinates [0, 0] and the top right of the board has coordinates [5, 8], depending on how many rows and columns the user wants to have (in this case it would be 6 rows and 9 columns.

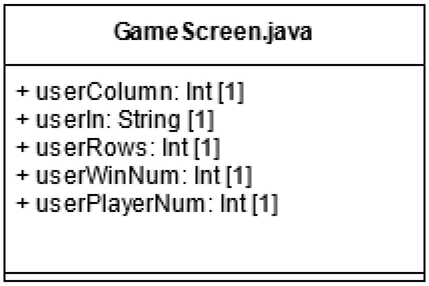
Deployment

Use “make” to compile all of the provided files. “make run” to run the actual program. “Make clean” can be used after running to remove any compiled class files. For our tests, “make test” will compile all of the test cases, while “make testGB” will run the tests for GameBoard, and “make testGBMem” will run the tests for GameBoardMem

Design

UML Class Diagrams

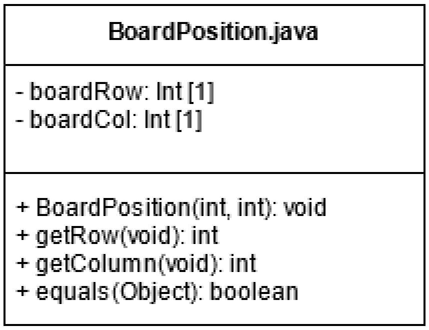
`GameScreen.java`



GameScreen.java

+ userColumn: Int [1]
+ userIn: String [1]
+ userRows: Int [1]
+ userWinNum: Int [1]
+ userPlayerNum: Int [1]

`BoardPosition.java`

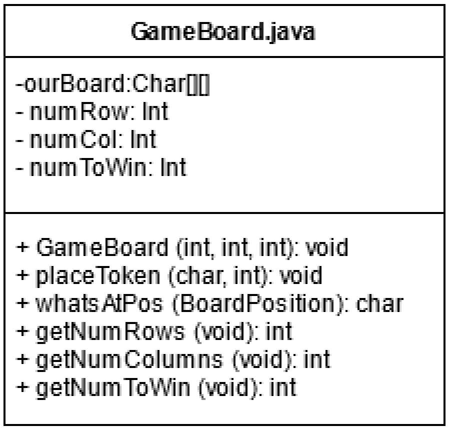


BoardPosition.java

- boardRow: Int [1]
- boardCol: Int [1]

+ BoardPosition(int, int): void
+ getRow(void): int
+ getColumn(void): int
+ equals(Object): boolean

`GameBoard.java`

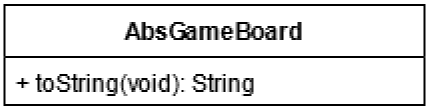


GameBoard.java

-ourBoard:Char[][]
- numRow: Int
- numCol: Int
- numToWin: Int

+ GameBoard (int, int, int): void
+ placeToken (char, int): void
+ whatsAtPos (BoardPosition): char
+ getNumRows (void): int
+ getNumColumns (void): int
+ getNumToWin (void): int

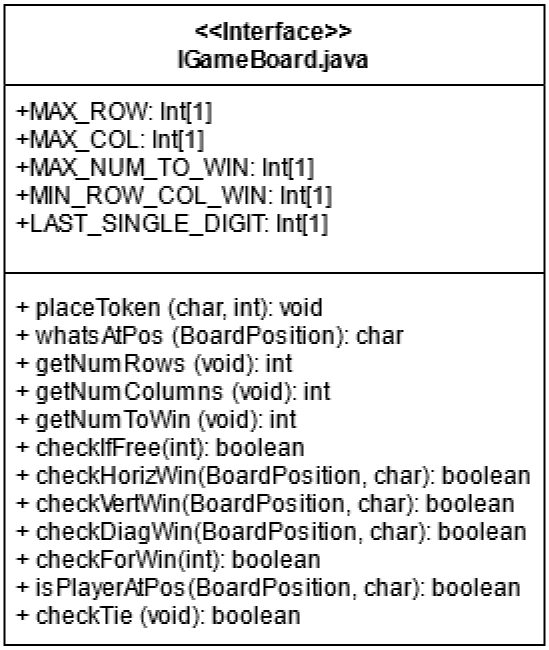
`AbsGameBoard.java`



AbsGameBoard

+ toString(void): String

IGameBoard.java

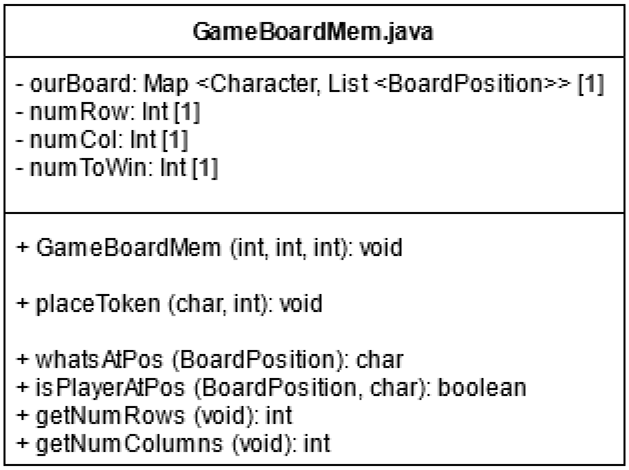


<<Interface>>
IGameBoard.java

+MAX_ROW: Int[1]
+MAX_COL: Int[1]
+MAX_NUM_TO_WIN: Int[1]
+MIN_ROW_COL_WIN: Int[1]
+LAST_SINGLE_DIGIT: Int[1]

+ placeToken (char, int): void
+ whatsAtPos (BoardPosition): char
+ getNumRows (void): int
+ getNumColumns (void): int
+ getNumToWin (void): int
+ checkIfFree(int): boolean
+ checkHorizWin(BoardPosition, char): boolean
+ checkVertWin(BoardPosition, char): boolean
+ checkDiagWin(BoardPosition, char): boolean
+ checkForWin(int): boolean
+ isPlayerAtPos(BoardPosition, char): boolean
+ checkTie (void): boolean

GameBoardMem.java



GameBoardMem.java

- ourBoard: Map <Character, List <BoardPosition>> [1]
- numRow: Int [1]
- numCol: Int [1]
- numToWin: Int [1]

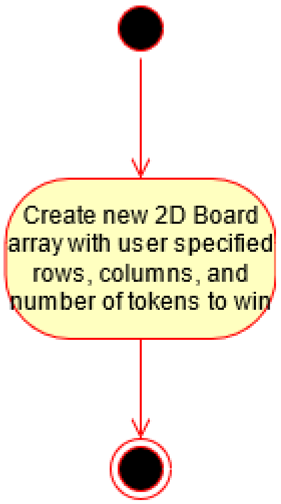
+ GameBoardMem (int, int, int): void

+ placeToken (char, int): void

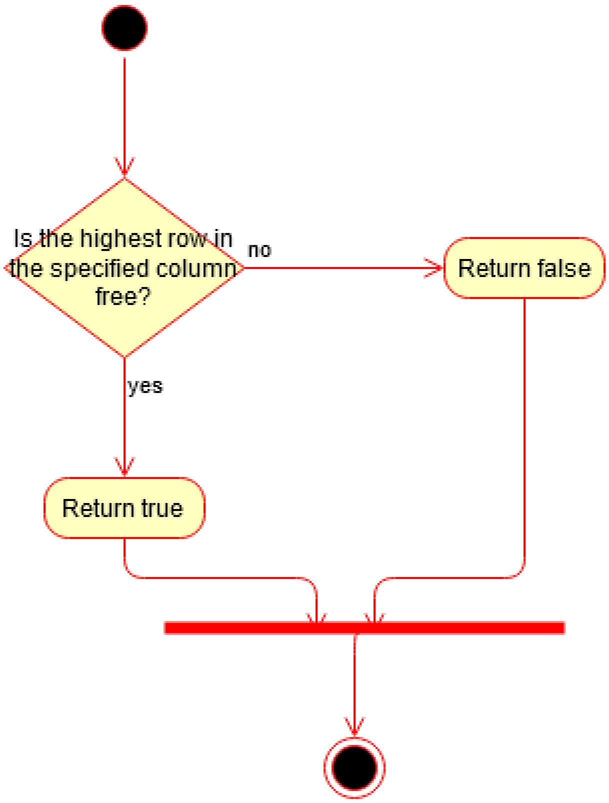
+ whatsAtPos (BoardPosition): char
+ isPlayerAtPos (BoardPosition, char): boolean
+ getNumRows (void): int
+ getNumColumns (void): int

UML Activity Diagrams

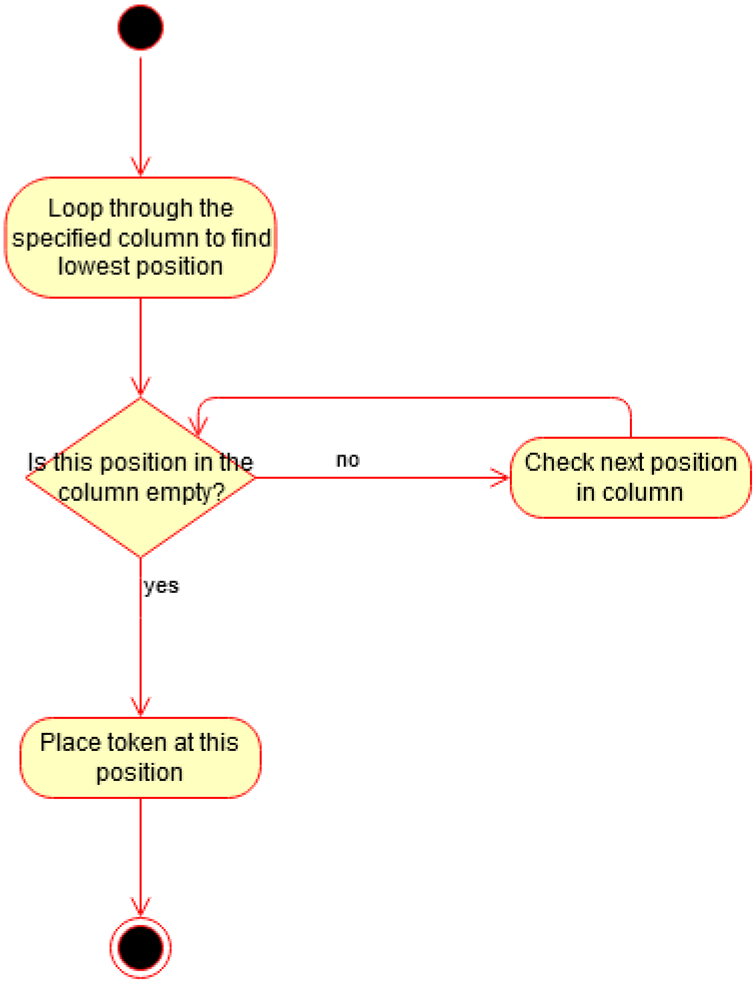
GameBoard.java - GameBoard()


Create new 2D Board array with user specified rows, columns, and number of tokens to win

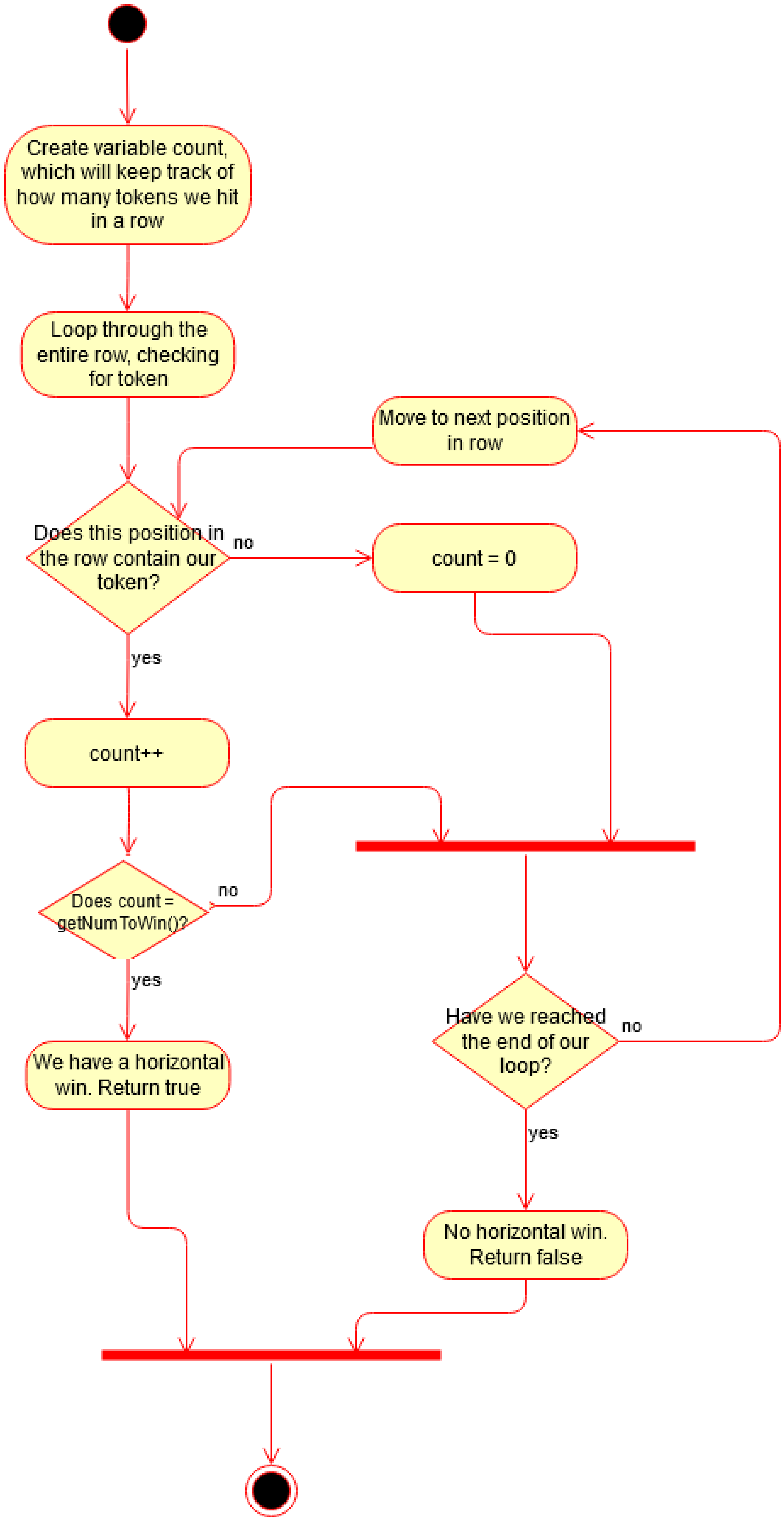
IGameBoard - checkIfFree()



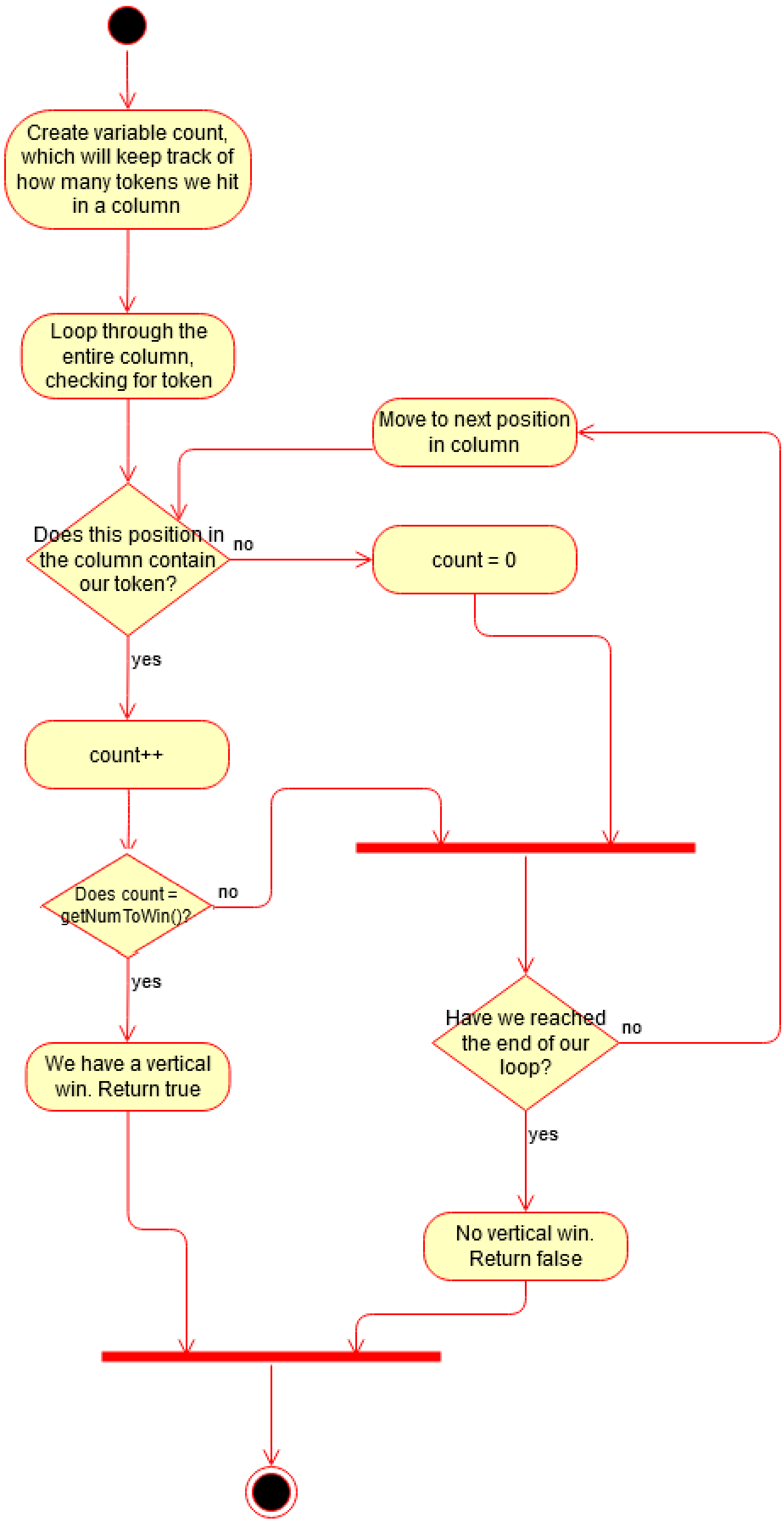
GameBoard - placeToken()


Loop through the specified column to find lowest position
Is this position in the column empty?
no
Check next position in column
yes
Place token at this position

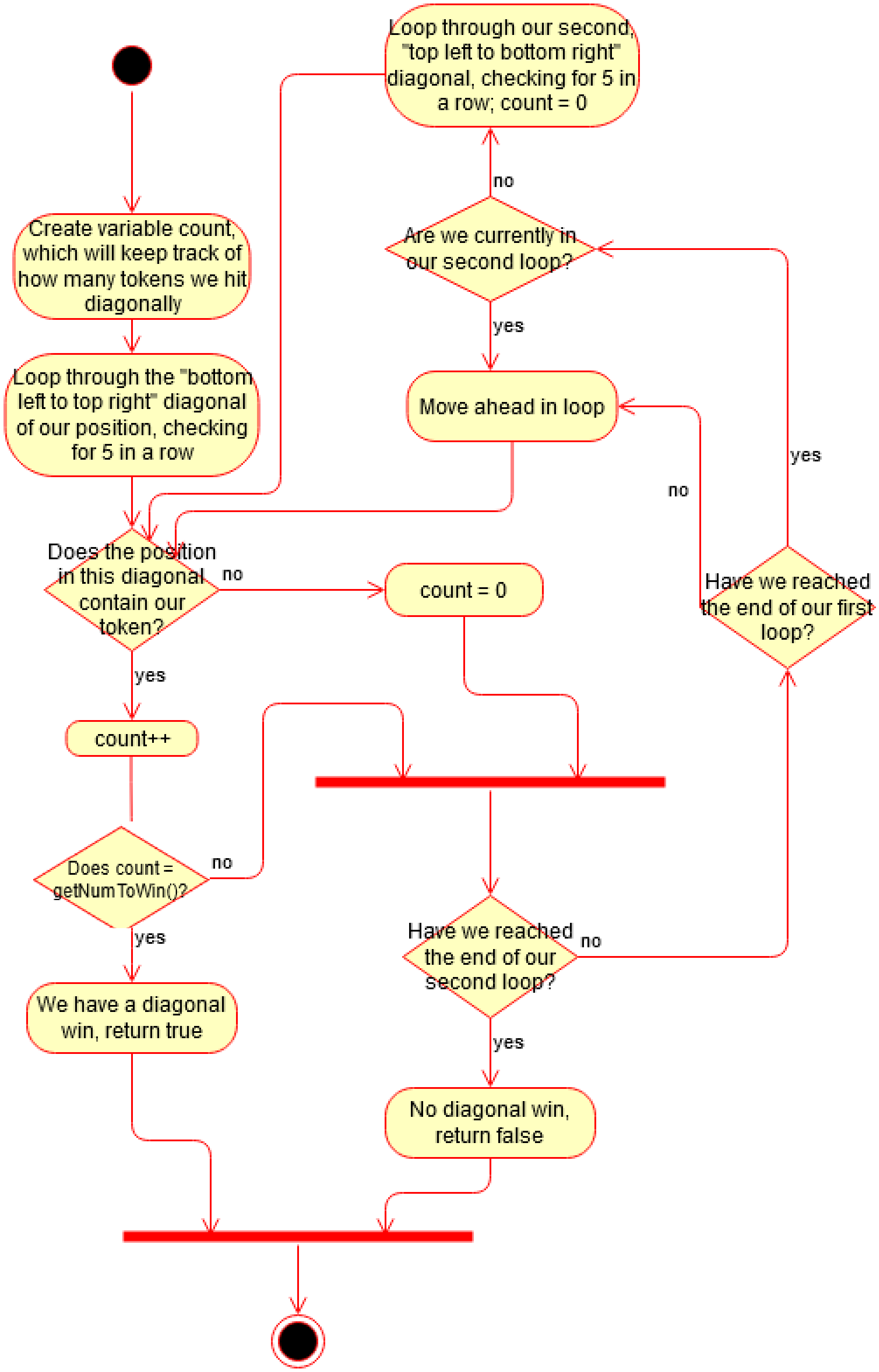
IGameBoard - checkHorizWin()


Create variable count, which will keep track of how many tokens we hit in a row
Loop through the entire row, checking for token
Move to next position in row
Does this position in the row contain our token?
no
count = 0
yes
count++
Does count = getNumToWin()?
no
yes
We have a horizontal win. Return true
Have we reached the end of our loop?
no
yes
No horizontal win. Return false

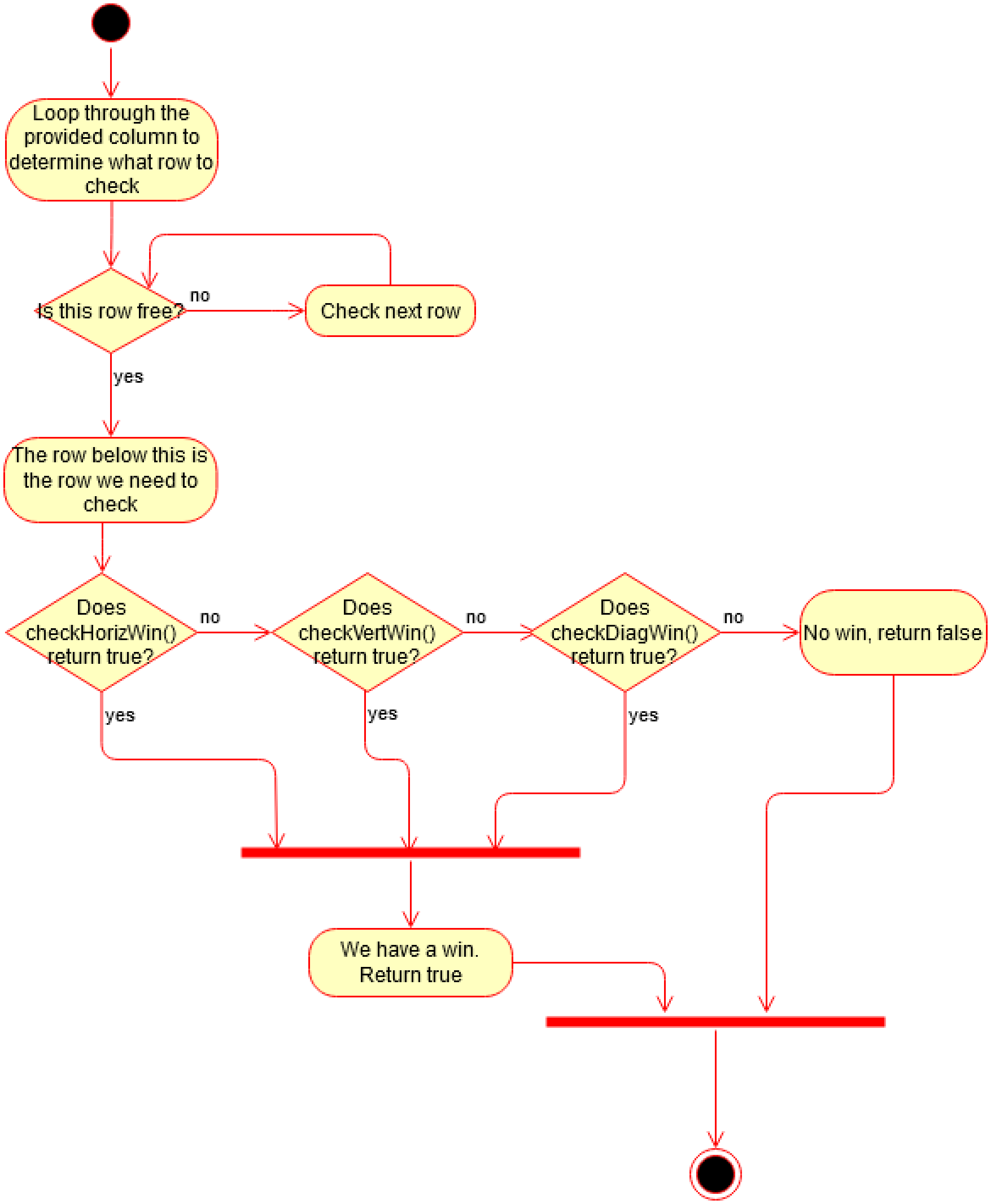
IGameBoard - checkVertWin()


Create variable count, which will keep track of how many tokens we hit in a column
Loop through the entire column, checking for token
Move to next position in column
Does this position in the column contain our token?
no
count = 0
yes
count++
Does count = getNumToWin()?
no
yes
We have a vertical win. Return true
Have we reached the end of our loop?
no
yes
No vertical win. Return false

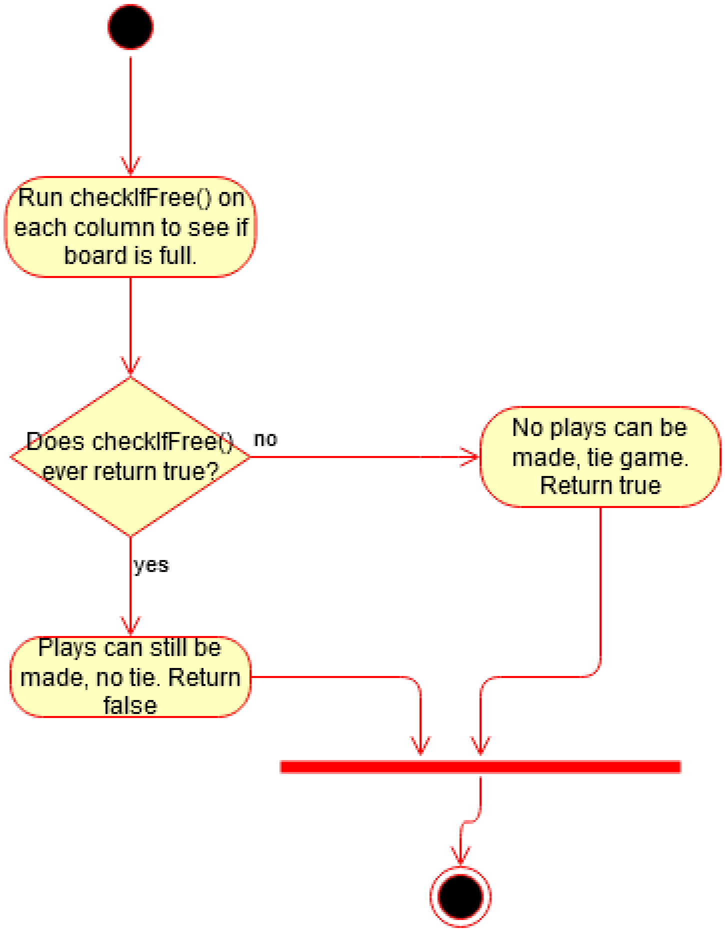
IGameBoard - checkDiagWin()


Loop through our second "top left to bottom right" diagonal, checking for 5 in a row; count = 0
no
Are we currently in our second loop?
yes
Move ahead in loop
Create variable count, which will keep track of how many tokens we hit diagonally
Loop through the "bottom left to top right" diagonal of our position, checking for 5 in a row
yes
no
Does the position in this diagonal contain our token?
no
count = 0
Have we reached the end of our first loop?
yes
count++
Does count = getNumToWin()?
no
yes
Have we reached the end of our second loop?
no
We have a diagonal win, return true
yes
No diagonal win, return false

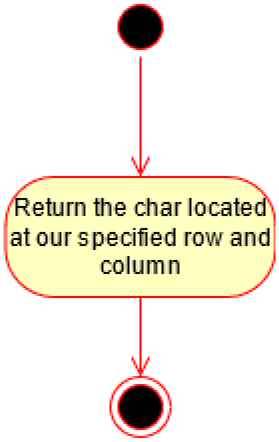
IGameBoard - checkForWin()


Loop through the provided column to determine what row to check
Is this row free?
no
Check next row
yes
The row below this is the row we need to check
Does checkHorizWin() return true?
no
Does checkVertWin() return true?
no
Does checkDiagWin() return true?
no
No win, return false
yes
yes
yes
We have a win. Return true

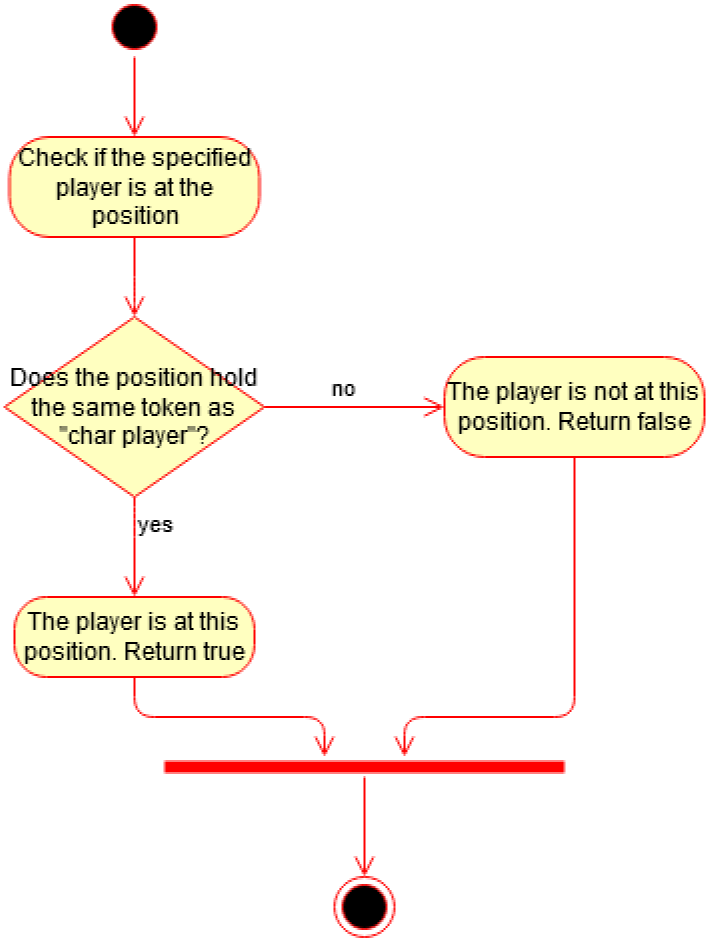
IGameboard - checkTie()


Run checkIfFree() on each column to see if board is full.
Does checkIfFree() ever return true?
no
No plays can be made, tie game. Return true
yes
Plays can still be made, no tie. Return false

Gameboard - whatsAtPos()

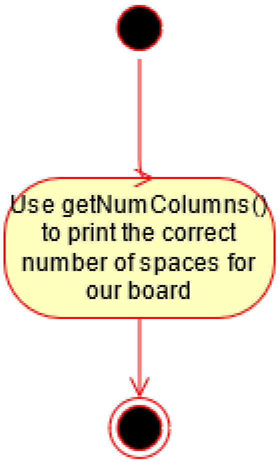


IGameBoard - isPlayerAtPos()

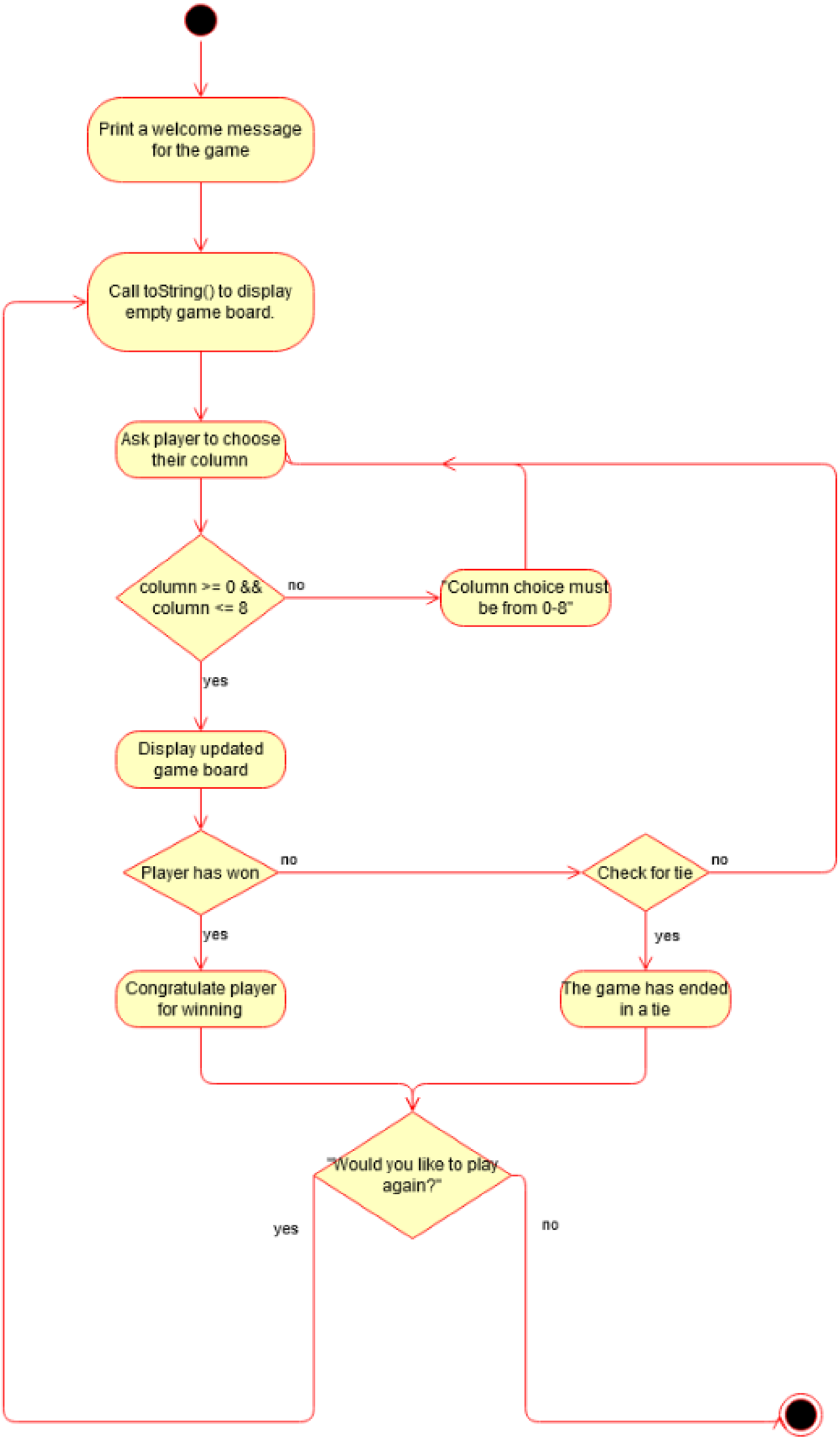


Check if the specified player is at the position
Does the position hold the same token as "char player"?
no
The player is not at this position. Return false
yes
The player is at this position. Return true

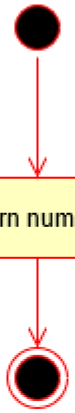
AbsGameBoard - toString()


Use getNumColumns() to print the correct number of spaces for our board

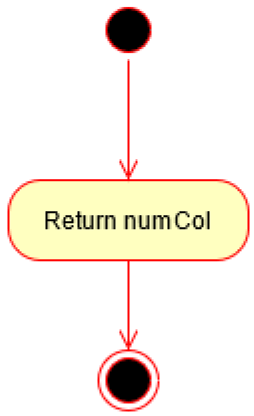
GameScreen.java - main()


Print a welcome message for the game
Call toString() to display empty game board.
Ask player to choose their column
column >= 0 && column <= 8
no
"Column choice must be from 0-8"
yes
Display updated game board
Player has won
no
Check for tie
no
yes
Congratulate player for winning
yes
The game has ended in a tie
"Would you like to play again?"
yes
no

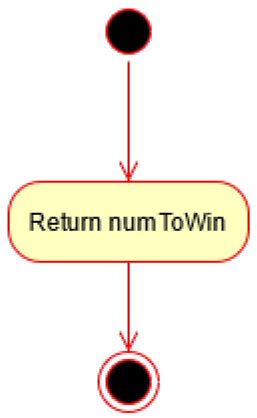
`GameBoard.java AND GameBoardMem.java - getNumRows()`



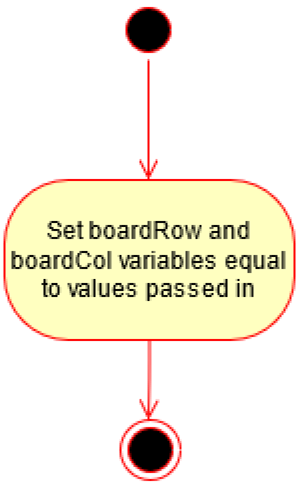
GameBoard.java AND GameBoardMem.java - getNumColumns()



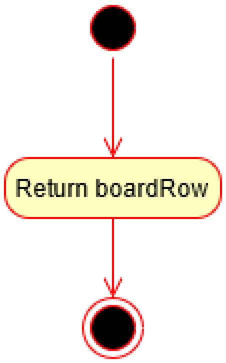
GameBoard.java - getNumToWin()



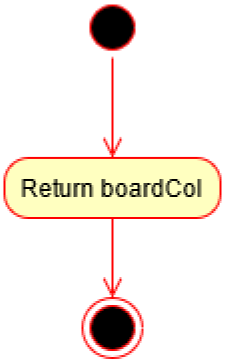
BoardPosition.java - BoardPosition()


Set boardRow and boardCol variables equal to values passed in

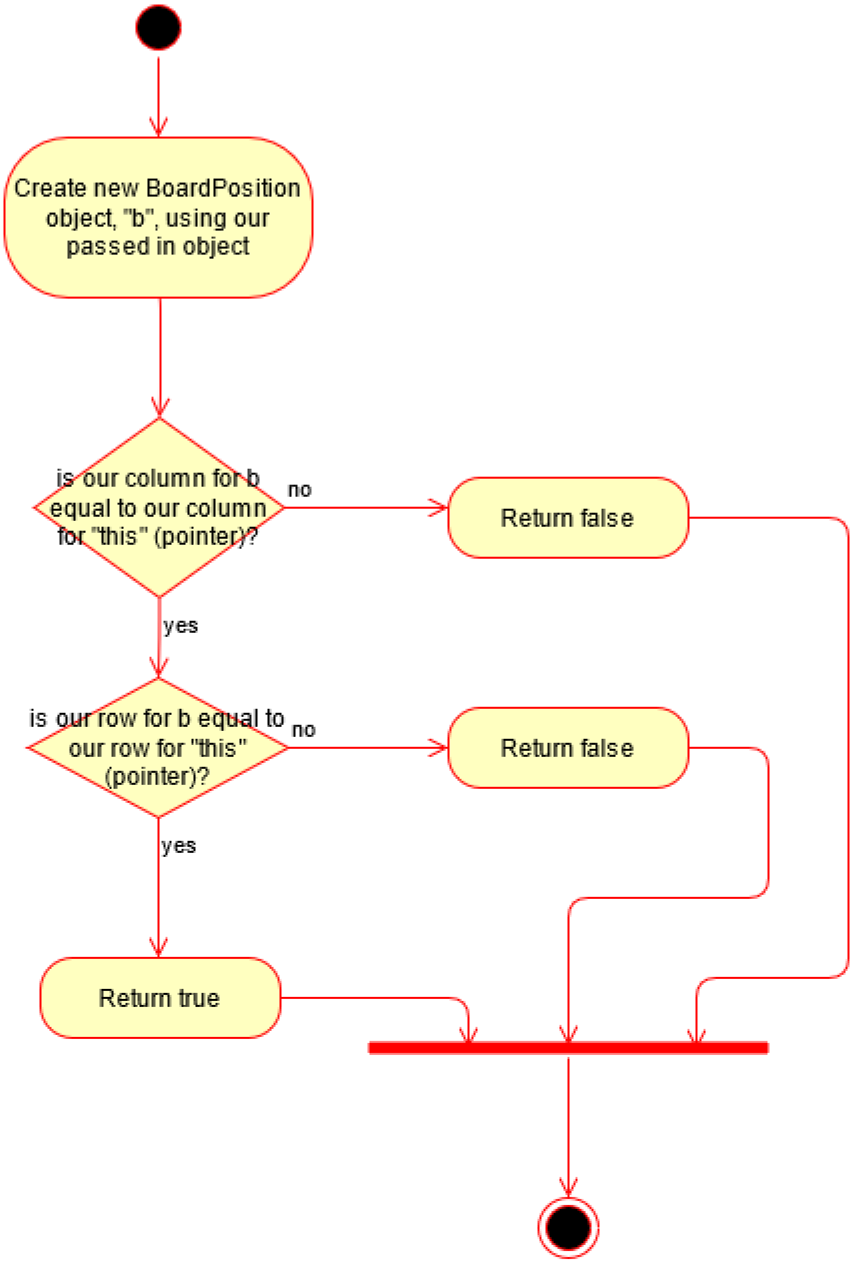
`BoardPosition.java - getRow()`



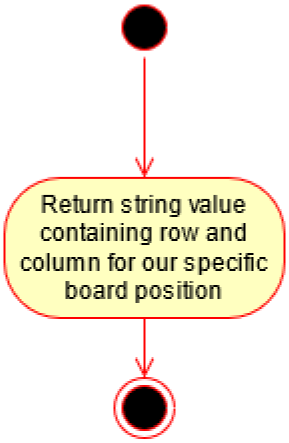
BoardPosition.java - getColumn()



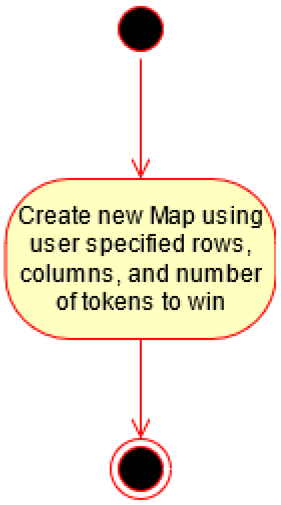
BoardPosition.java - equals()


Create new BoardPosition object, "b", using our passed in object
is our column for b equal to our column for "this" (pointer)?
no
Return false
yes
is our row for b equal to our row for "this" (pointer)?
no
Return false
yes
Return true

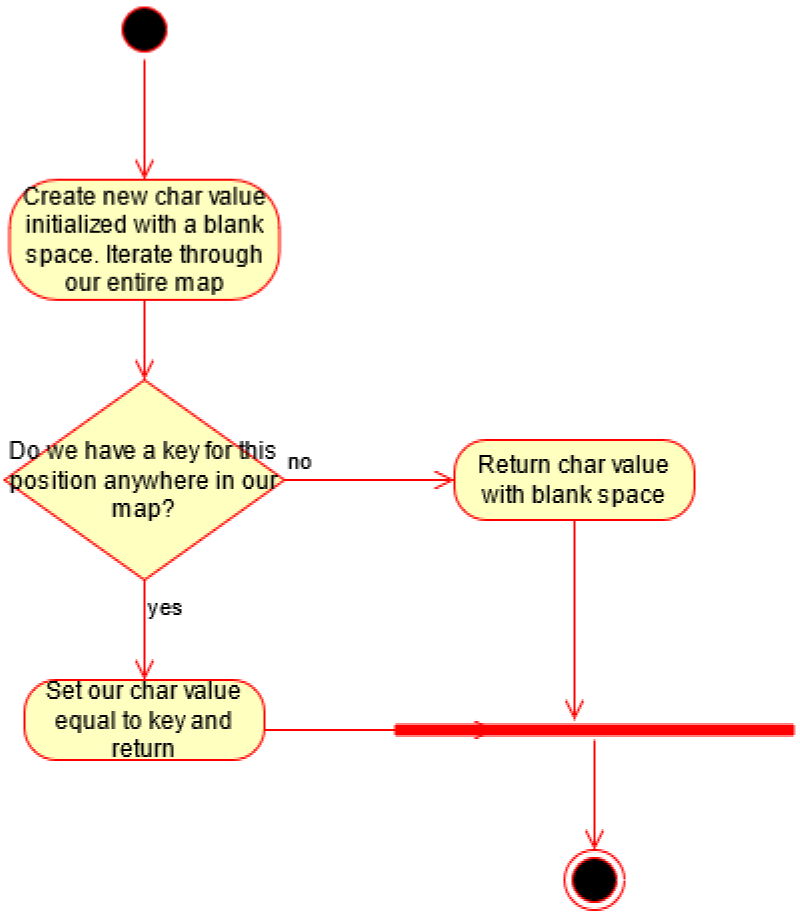
BoardPosition.java - toString


Return string value containing row and column for our specific board position

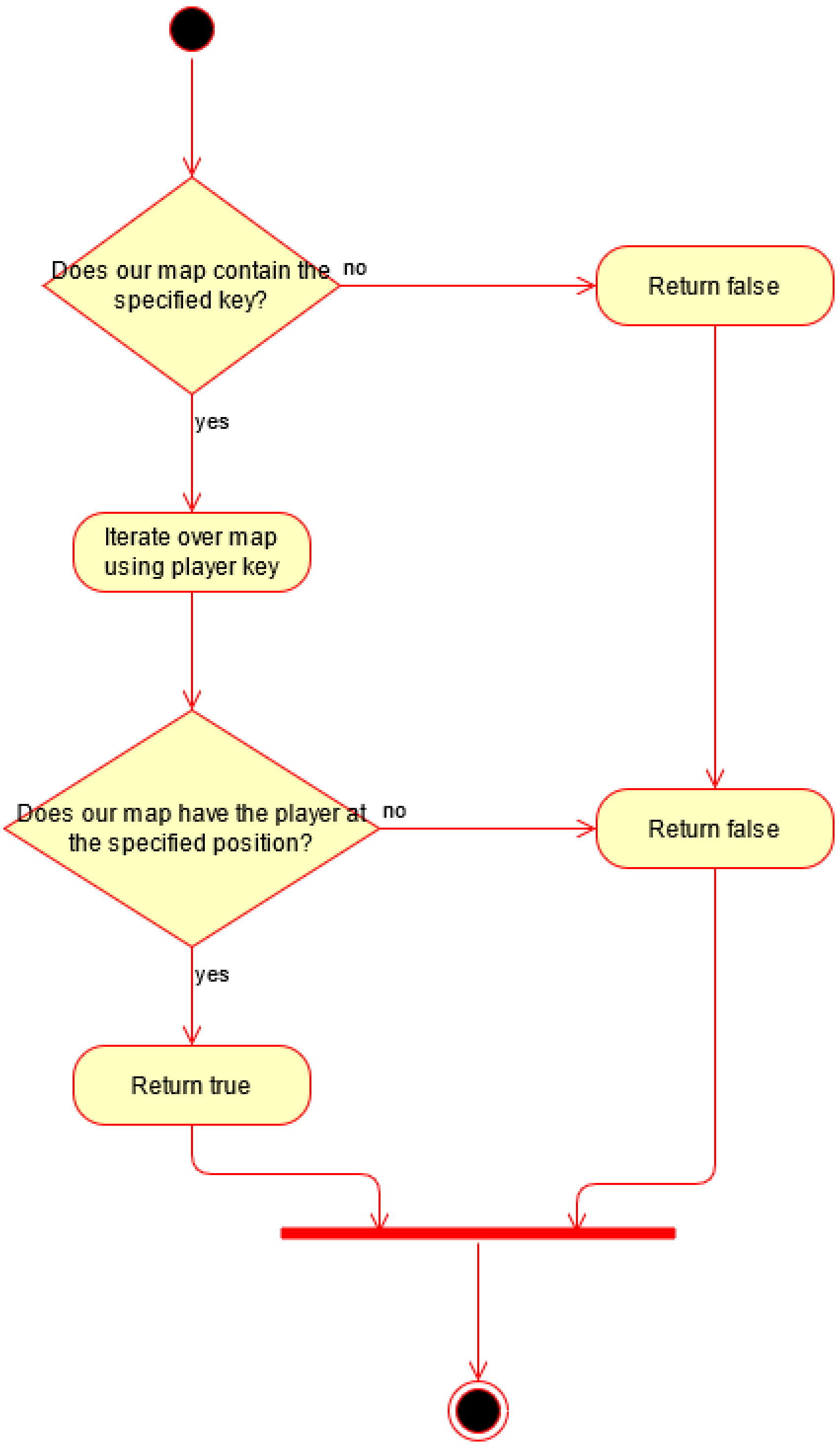
GameBoardMem.java - GameBoardMem()


Create new Map using user specified rows, columns, and number of tokens to win

GameBoardMem.java - whatsAtPos()


Create new char value initialized with a blank space. Iterate through our entire map
Do we have a key for this position anywhere in our map?
no
Return char value with blank space
yes
Set our char value equal to key and return

GameBoardMem - isPlayerAtPos()


Does our map contain the specified key?
no
Return false
yes
Iterate over map using player key
Does our map have the player at the specified position?
no
Return false
yes
Return true

Testing

GameBoard(int userRows, int userCols, int userNumToWin) / GameBoardMem(int userRows, int userCols, int userNumToWin)

<table>
<tr>
<td><b>Input:</b>
userRows = 100
userCols = 100
userNumToWin = 25</td>
<td><b>Output:</b>

Board = 100 * 100

board.getNumToWin() = 25</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are creating a board using the largest possible values for row, column, and wins
<b>Function Name:</b>
test_Constructor_Large</td>
</tr>
<tr>
<td><b>Input:</b>
State:
userRows = 100
userCols = 3
userNumToWin = 25</td>
<td><b>Output:</b>

Board = 100 * 3

board.getNumToWin() = 25</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are creating a board using the largest possible values for row and the smallest value for column
<b>Function Name:</b>
test_Constructor_Mix</td>
</tr>
<tr>
<td><b>Input:</b>
State:
userRows = 3
userCols = 3
userNumToWin = 3</td>
<td><b>Output:</b>
<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>
board.getNumToWin() = 3</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are creating a board using the smallest possible values for row, column, and wins
<b>Function Name:</b>
test_Constructor_Small</td>
</tr>
</table>

boolean checkIfFree(int c)

<table>
<tr>
<td><b>Input:</b>
State:

Empty Board</td>
<td><b>Output:</b>

checkIfFree(4) = true

Empty Board</td>
<td><b>Reason:</b>
This test case is unique and distinct because it tests for when checkIfFree should return true, with an empty board
<b>Function Name:</b>
test_CheckIfFree_empty</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
</table>
</td>
<td><b>Output:</b>

checkIfFree(4) = false

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because it tests a condition in which checkIfFree should return false, when the entire column is full
<b>Function Name:</b>
test_CheckIfFree_notFree</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>X</td><td></td><td></td></tr>
</table>
</td>
<td><b>Output:</b>

checkIfFree(2) = true

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which a token is in a column but that column is still free
<b>Function Name:</b>
test_CheckIfFree_tokenPresent</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td></td><td></td></tr>
</table>
pos = (0, 3)
p = ‘X’
numToWin = 3</td>
<td><b>Output:</b>
checkHorizWin = true
Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which a horizontal win has occurred
<b>Function Name:</b>
test_CheckHorizWin_winCase</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (1, 2)
p = ‘X’
numToWin = 3</td>
<td><b>Output:</b>
checkHorizWin = false
Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is NOT a horizontal win, because the board is empty
<b>Function Name:</b>
test_CheckHorizWin_noWinCase</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>O</td><td>X</td><td></td><td></td></tr>
</table></td>
<td><b>Output:</b>
checkHorizWin = false
Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which enough tokens exist for a horizontal win, but they are not the same tokens
<b>Function Name:</b>
test_CheckHorizWin_tokenMix</td>
</tr>
</table>

<table>
<tr><td>pos = (0, 3)
p = ‘X’
numToWin = 3</td><td></td><td></td></tr>
<tr><td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td></td><td></td></tr>
</table>
pos = (0, 3)
p = ‘X’
numToWin = 4</td><td><b>Output:</b>
checkHorizWin = false
Board stays the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there are tokens in a row present but not enough to win
<b>Function Name:</b>
test_CheckHorizWin_almostWinCase</td></tr>
</table>

boolean checkVertWin(BoardPosition pos, char p)

<table>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
</table>
pos = (2, 1)
p = ‘X’
numToWin = 3</td>
<td><b>Output:</b>

checkVertWin = true

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which a vertical win has occurred

<b>Function Name:</b>
test_CheckVertWin_winCase</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (3, 2)
p = ‘X’
numToWin = 3</td>
<td><b>Output:</b>

checkVertWin = false

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is no vertical win
<b>Function Name:</b>
test_CheckVertWin_noWinCase</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td></tr>
</table>
pos = (3, 2)
p = ‘X’
numToWin = 4</td>
<td><b>Output:</b>

checkVertWin = false

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which a vertical win has almost occurred
<b>Function Name:</b>
test_CheckVertWin_almostWinCase</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td>O</td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr></table>
pos = (3, 2)
numToWin = 3</td><td><b>Output:</b>
checkVertWin = false
Board is the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which enough tokens exist for a vertical win, but they are not the same tokens
<b>Function Name:</b>
test_CheckVertWin_tokenMix</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr></table>
pos = (0, 0)
numToWin = 3</td><td><b>Output:</b>
checkDiagWin = false
Board is the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is no diagonal win
<b>Function Name:</b>
test_CheckDiagWin_noWinCase</td></tr>
<tr><td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td>X</td><td>O</td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>O</td><td></td><td></td></tr></table>
pos = (0, 0)
numToWin = 3</td><td><b>Output:</b>
checkDiagWin = true
Board is the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is a diagonal win that goes from bottom left to top right
<b>Function Name:</b>
test_CheckDiagWin_leftWinCase</td></tr>
</table>

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td><b>Input:</b>
State:<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td>X</td><td>O</td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>O</td><td></td><td></td></tr></table>
pos = (0, 0)
numToWin = 4</td><td><b>Output:</b>

checkDiagWin = false

Board is the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is ALMOST a diagonal win going from the bottom left to the top right
<b>Function Name:</b>
test_CheckDiagWin_almostLeftWinCase</td></tr>
<tr><td><b>Input:</b>
State:<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td>M</td><td>O</td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>O</td><td></td><td></td></tr></table>
pos = (0, 0)
numToWin = 3</td><td><b>Output:</b>

checkDiagWin = false

Board is the same</td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there are tokens in a row diagonally from left to right but they are not all the same token
<b>Function Name:</b>
test_CheckDiagWin_mixedTokensLeft</td></tr>
<tr><td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>O</td><td>X</td><td></td></tr></table>
pos = (0, 6)
p = ‘X’
numToWin = 3</td><td><b>Output:</b></td><td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is a diagonal win going from the bottom right to the top left
<b>Function Name:</b>
test_CheckDiagWin_rightWinCase</td></tr>
</table>

<table>
<tr><td></td><td></td><td></td></tr>
<tr>
<td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>O</td><td>X</td><td></td></tr></table>
pos = (0, 6)
p = ‘X’
numToWin = 4;</td>
<td><b>Output:</b>
checkDiagWin = false
Board stays the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is ALMOST a diagonal win going from the bottom right to the top left
<b>Function Name:</b>
test_CheckDiagWin_almostRightWinCase</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>M</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td>O</td><td>O</td><td>X</td><td></td></tr></table>
pos = (0, 6)
p = ‘X’
numToWin = 3;</td>
<td><b>Output:</b>
checkDiagWin = false
Board stays the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there are tokens in a row diagonally from right to left but they are not all the same token
<b>Function Name:</b>
test_CheckDiagWin_mixedTokensRight</td>
</tr>
</table>

boolean checkTie()

<table>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td><b>Output:</b>

checkTie = false

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which our board is empty and therefore cannot have a tie
<b>Function Name:</b>
test_CheckTie_boardEmpty</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
</table>
</td>
<td><b>Output:</b>

checkTie = false

Board is the same</td>
<td><b>Reason:</b>
This test case is distinct and unique because we are testing a condition in which our board has tokens but is not full, this ensures that are program is effectively checking every column
<b>Function Name:</b>
test_CheckTie_boardOccupied</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
</table>
</td>
<td><b>Output:</b>

checkTie = true

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which our board is full without any wins and should therefore result in a tie game
<b>Function Name:</b>
test_CheckTie_boardFull</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
</table>
</td>
<td><b>Output:</b>

checkTie = false

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which every space is filled but one. This will allow us to ensure every single space is being checked
<b>Function Name:</b>
test_CheckTie_oneSpace</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (0, 0)</td>
<td><b>Output:</b>

whatsAtPos = ‘ ‘

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which our board is empty and thus our method should always return false
<b>Function Name:</b>
test_WhatsAtPos_boardEmpty</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (0, 3)</td>
<td><b>Output:</b>

whatsAtPos = ‘X’

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is a player located at the specified position that we need to return
<b>Function Name:</b>
test_WhatAtPos_playerAtPlace</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (3, 3)</td>
<td><b>Output:</b>

whatsAtPos = ‘ ‘

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is a player located at position but not the one we are checking. This will ensure our function is checking the right location
<b>Function Name:</b>
test_WhatAtPos_playerAtWrongPlace</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>M</td><td></td><td></td><td></td><td></td></tr>
</table>
pos = (0, 3)</td>
<td><b>Output:</b>

whatsAtPos = ‘M’

Board is the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which our function needs to return a token that is not a ' ', 'X', or 'O
'<b>Function Name:</b>
test_WhatsAtPos_uniqueChar</td>
</tr>
<tr>
<td><b>Input:</b></td>
<td><b>Output:</b></td>
<td><b>Reason:</b></td>
</tr>
</table>

<table>
<tr>
<td>State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>M</td><td>X</td><td>O</td><td></td><td></td><td></td></tr></table>
pos = (0, 3)</td>
<td>whatsAtPos = ‘X’

Board is the same</td>
<td>This test case is unique and distinct because we are testing a condition in which there are multiple players on our board and our method must return the correct one
<b>Function Name:</b>
test_WhatsAtPos_multiplePlayers</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>
pos = (3, 5)
p = ‘X’</td>
<td><b>Output:</b>

isPlayerAtPos = false</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which our board is completely empty and should thus always return false when calling isPlayer
<b>Function Name:</b>
test_isPlayerAtPos_boardEmpty</td>
</tr>
<tr>
<td><b>Input:</b>
State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td><td></td></tr></table>
pos = (0, 3)
p = ‘X’</td>
<td><b>Output:</b>

isPlayerAtPos = true

Board stays the same</td>
<td><b>Reason:</b>
This test case is unique and distinct because we are testing a condition in which there is a player located at the specified position
<b>Function Name:</b>
test_IsPlayerAtPos_playerAtPlace</td>
</tr>
</table>

Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which our function needs to return a token that is not a ' ', 'X', or 'O' **Function Name:** test_IsPlayerAtPos_uniqueChar
Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which there are multiple players on our board **Function Name:** test_isPlayerAtPos_multiplePlayers
Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which there is a player located at position but not the one we are checking. This will ensure our function is checking the right location **Function Name:** test_isPlayerAtPos_playerAtWrongPlace

It was at this point I realized it was 9:40 and my best bet was check on SOC and submit. Below I was able to fill in my function names and reasons. I also have all my test cases complete. Hopefully this is enough for a decent grade, lord knows I could use one. Also, just to restate I do have all my reasons and function names done, and my inputs and outputs past this point really are basically the exact same as the ones above. I know, I know, points must be removed ☹ but with everything else complete hopefully this isn't too much of a deduction. Now I'm wondering if you have stopped reading up to this point lol. Now I'm realizing I'm running out of time to turn this in. I will do that now.

void placeToken(char p, int c)

Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which our placeToken function needs to fill the entire board **Function Name:** test_PlaceToken_boardFull
Input: State:	**Output:**	**Reason:** This test case is unique and distinct be we are testing a condition in

		which we need to fill our entire board using multiple different characters **Function Name:** test_PlaceToken_boardFullMultChars
Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which are placing a single token in the first column of our board **Function Name:** test_PlaceToken_firstColumn
Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which there are multiple players that need to be placed onto the board **Function Name:** test_PlaceToken_multiplePlayers
Input: State:	**Output:**	**Reason:** This test case is unique and distinct because we are testing a condition in which are placing a single token in the last column of our board **Function Name:** test_PlaceToken_finalColumn