Ramneek Chhatwal
CPSC 2150
HW 5

Project 5 Report

Requirements:

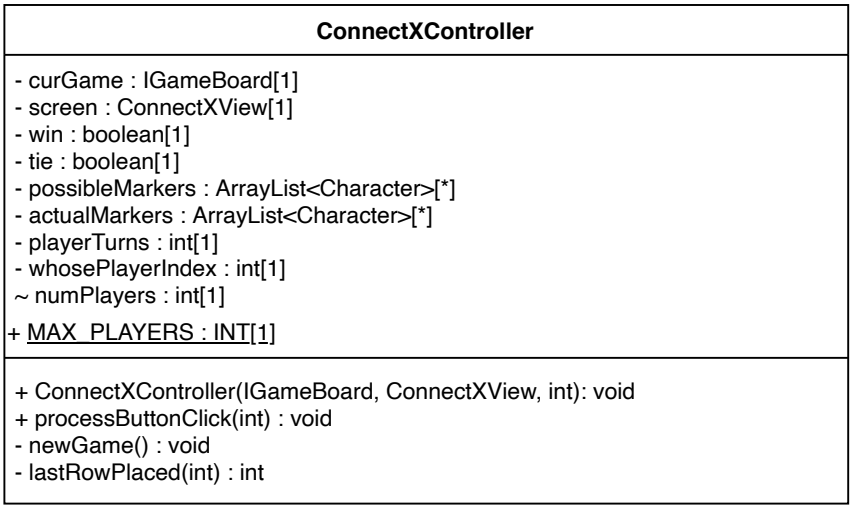
Functional Requirements:

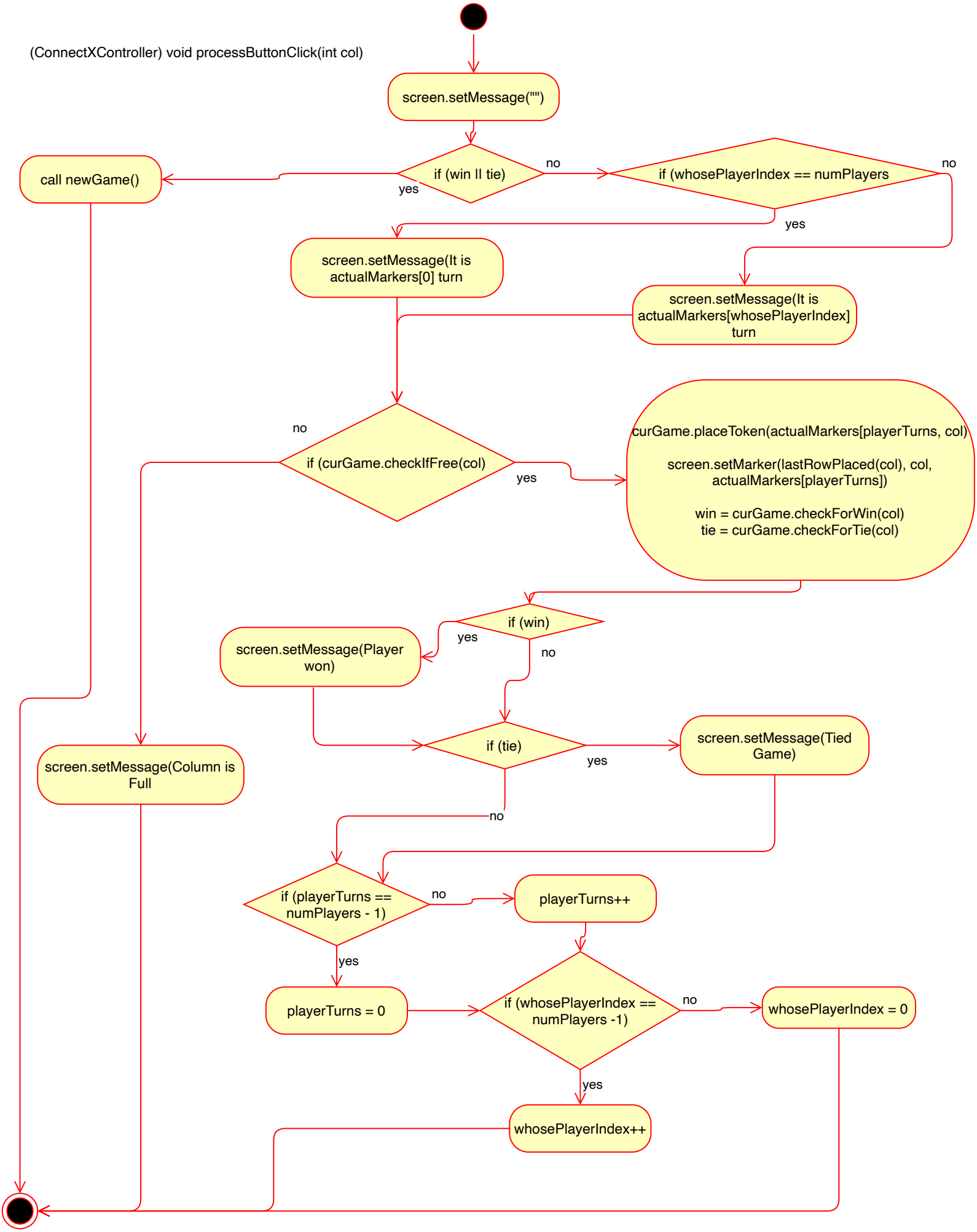
- As a player, I will be able to select the number of players that I want to play in the same game to include everyone I'm around in my game.
- As a player, I will be able to change the size of the grid to make the game bigger or smaller to account for how long I want to play the game.
- As a player, I will be able to select the number of markers in a row to win, to make the game more challenging.
- As a player, I can click a column to insert a marker in a way to win the game horizontally, so I can win.
- As a player, I can click a column to insert a marker in a way to win the game vertically, so I can win.
- As a player, I can click a column to insert a marker in a way to win the game diagonally, so I can win.
- As a player, I will be able to take turns with other players in the game to make the game more competitive and to follow the rules of ConnectX.
- As a player, I can see if I have tied with another player, so that we can know that outcome of the game.
- As a player, I can click the screen after a finished game (win or tie) to play another game.
- As a player, I am able to exit out of the screen to stop playing.

Non-Functional Requirements:

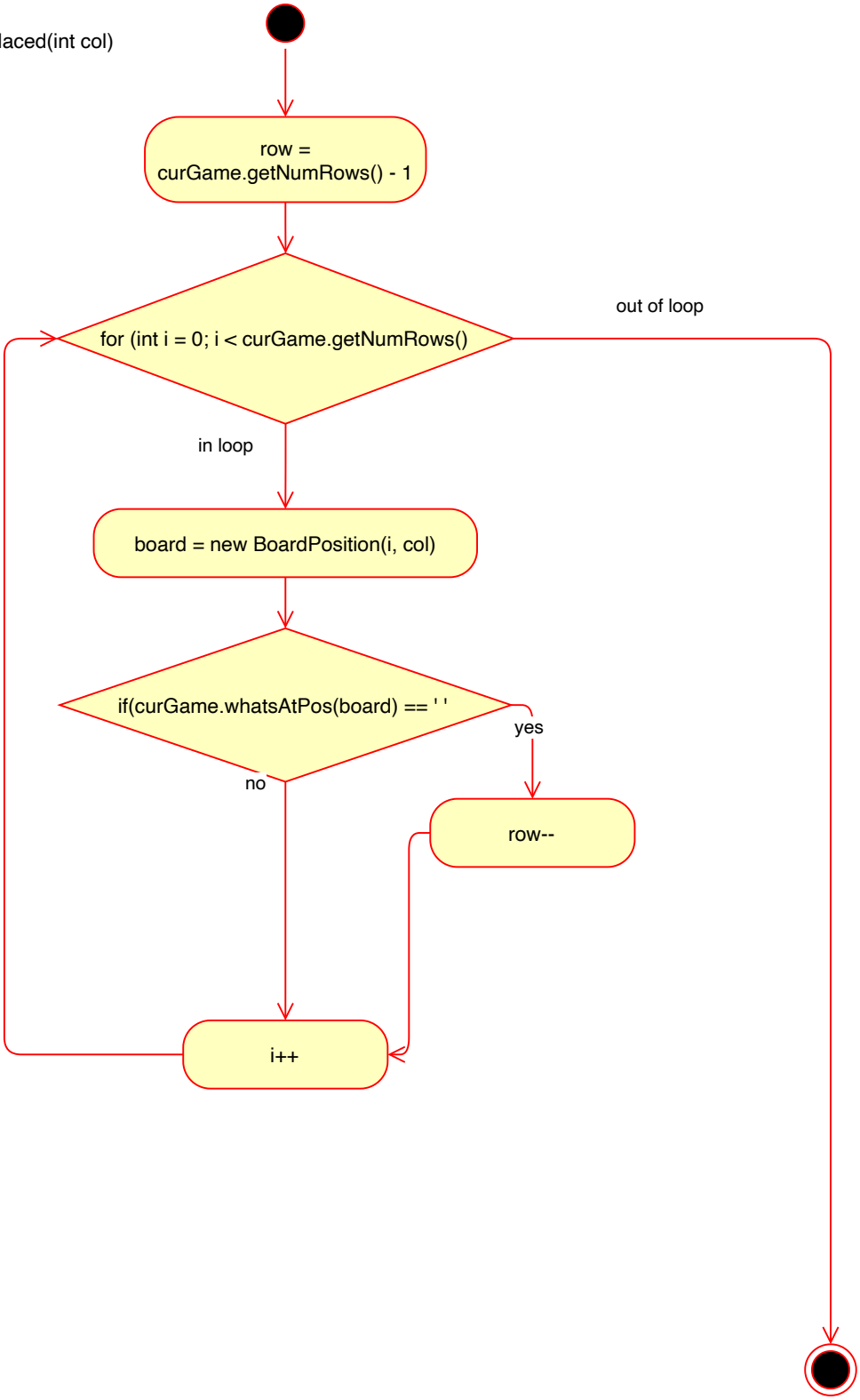
- Program will compile using Java 8
- Program will be able to run on the School of Computing Machines
- Pass all JUnit 4.10 tests
- Must allow first player to have first turn
- [0,0] is the bottom-left position of the board
- Must have a memory efficient implementation
- Must have a run-time efficient implementation
- Must support up to a 100x100 game board
- Must support a minimum of a 3x3 game board
- Must run using GUI
- Does not use command line for interfacing with user
- Follows Model View Controller architectural pattern
- Waits for events to occur and uses even driven programming

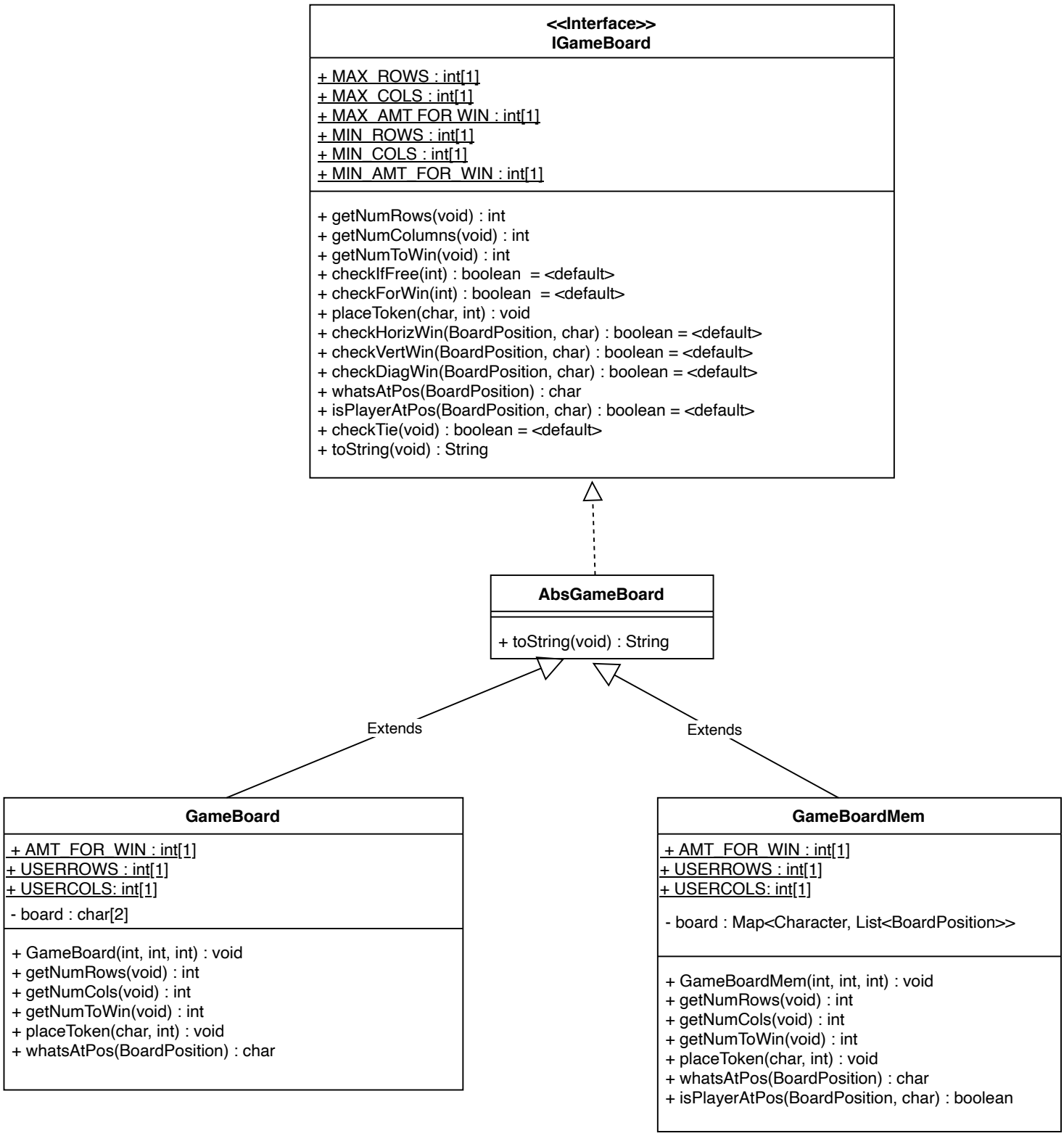
Design:


ConnectXController
- curGame : IGameBoard[1]
- screen : ConnectXView[1]
- win : boolean[1]
- tie : boolean[1]
- possibleMarkers : ArrayList<Character>[*]
- actualMarkers : ArrayList<Character>[*]
- playerTurns : int[1]
- whosePlayerIndex : int[1]
~ numPlayers : int[1]
+ MAX_PLAYERS : INT[1]
+ ConnectXController(IGameBoard, ConnectXView, int): void
+ processButtonClick(int) : void
- newGame() : void
- lastRowPlaced(int) : int

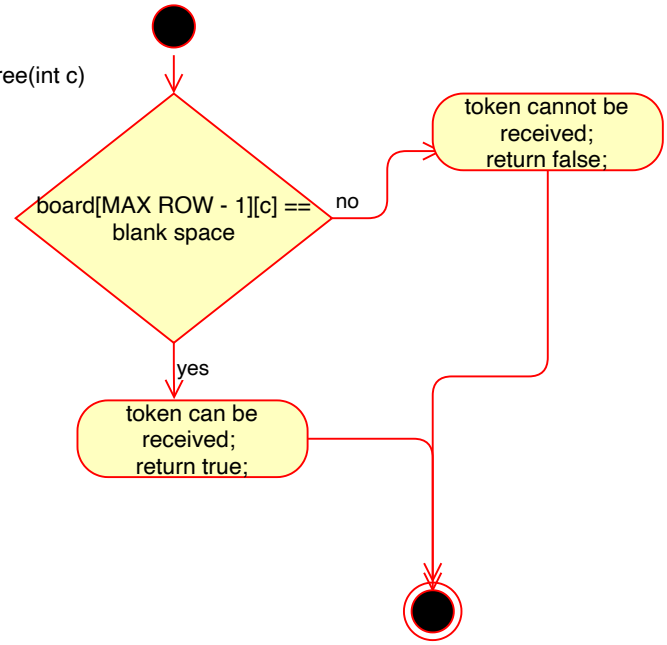

(ConnectXController) void processButtonClick(int col)
screen.setMessage("")
if (win II tie)
yes
no
call newGame()
if (whosePlayerIndex == numPlayers
no
yes
screen.setMessage(It is actualMarkers[0] turn
screen.setMessage(It is actualMarkers[whosePlayerIndex] turn
no
if (curGame.checkIfFree(col)
yes
curGame.placeToken(actualMarkers[playerTurns, col)
screen.setMarker(lastRowPlaced(col), col, actualMarkers[playerTurns])
win = curGame.checkForWin(col)
tie = curGame.checkForTie(col)
if (win)
yes
no
screen.setMessage(Player won)
if (tie)
yes
screen.setMessage(Tied Game)
no
screen.setMessage(Column is Full
if (playerTurns == numPlayers - 1)
no
playerTurns++
yes
playerTurns = 0
if (whosePlayerIndex == numPlayers -1)
no
whosePlayerIndex = 0
yes
whosePlayerIndex++

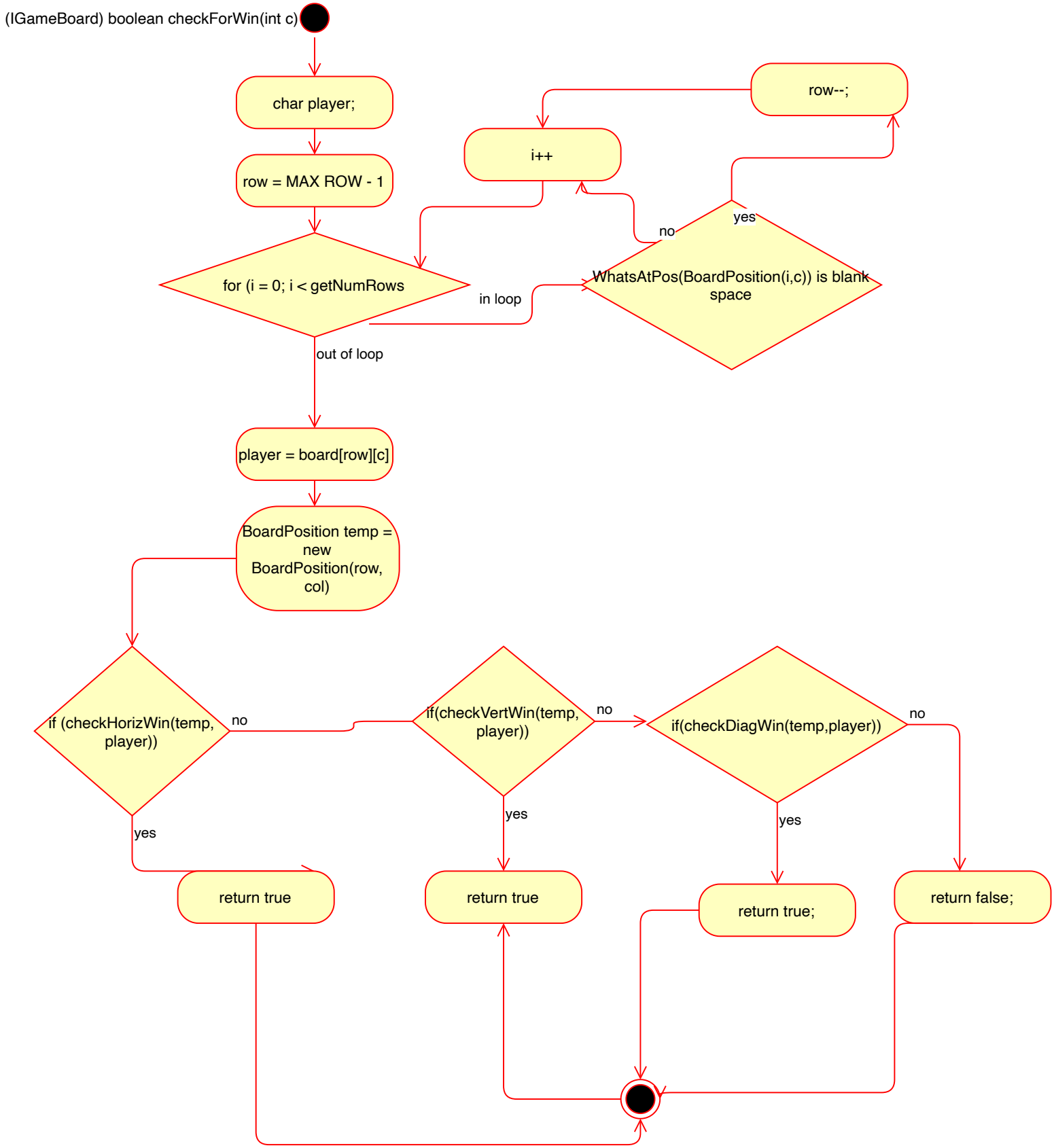
(ConnectXController) int lastRowPlaced(int col)

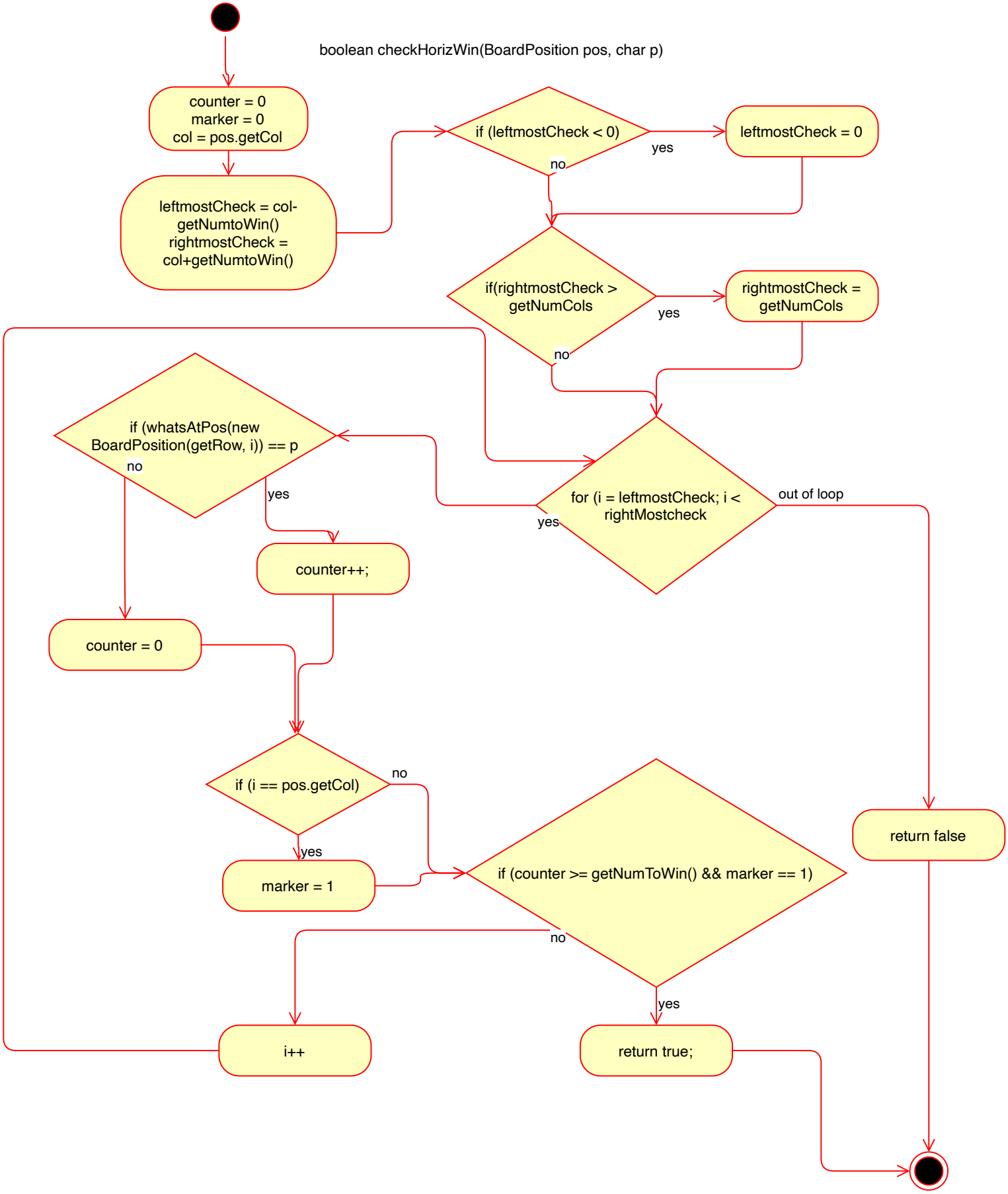


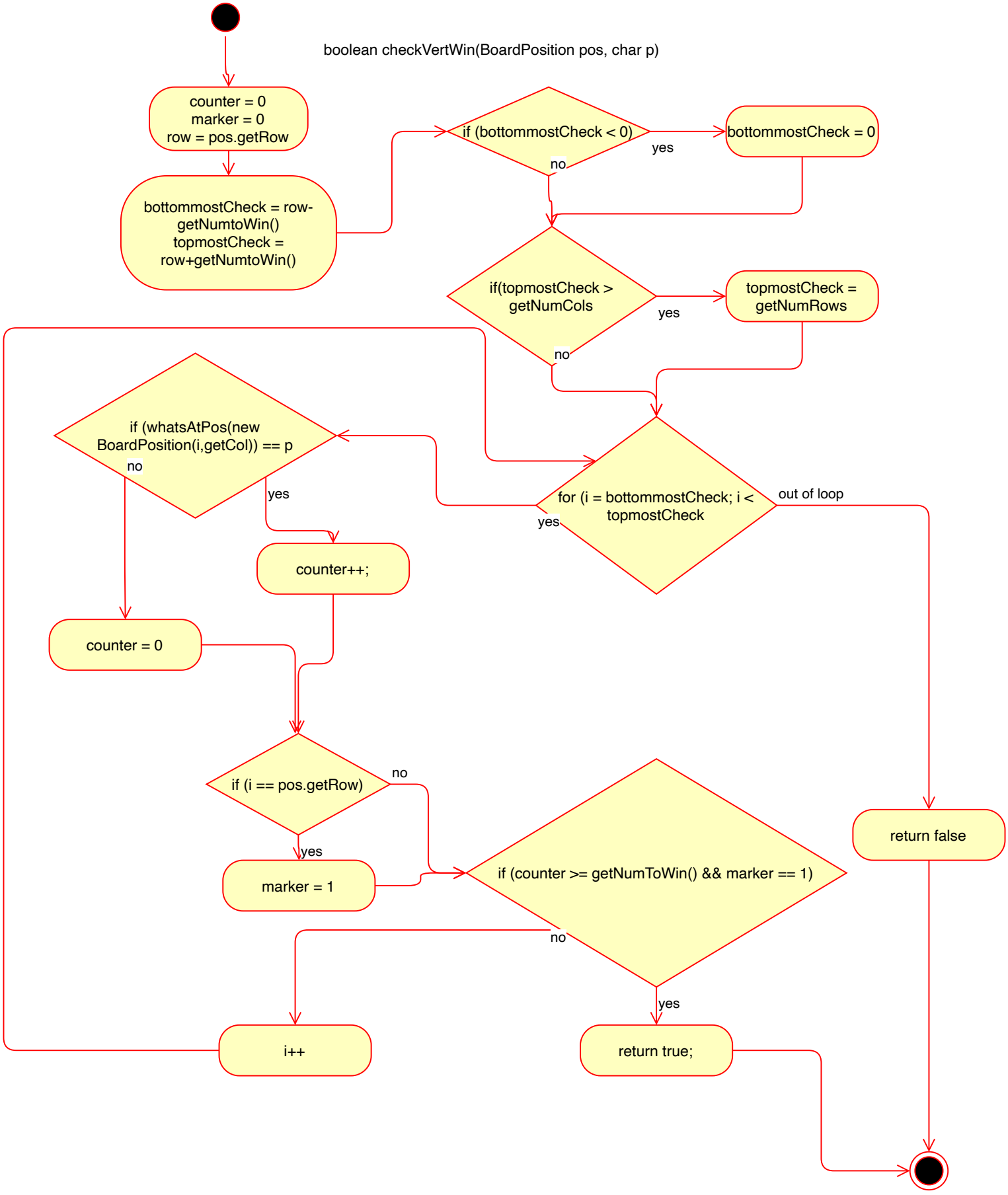

<<Interface>>
IGameBoard
+ MAX_ROWS : int[1]
+ MAX_COLS : int[1]
+ MAX_AMT FOR WIN : int[1]
+ MIN_ROWS : int[1]
+ MIN_COLS : int[1]
+ MIN_AMT_FOR_WIN : int[1]
+ getNumRows(void) : int
+ getNumColumns(void) : int
+ getNumToWin(void) : int
+ checkIfFree(int) : boolean = <default>
+ checkForWin(int) : boolean = <default>
+ placeToken(char, int) : void
+ checkHorizWin(BoardPosition, char) : boolean = <default>
+ checkVertWin(BoardPosition, char) : boolean = <default>
+ checkDiagWin(BoardPosition, char) : boolean = <default>
+ whatsAtPos(BoardPosition) : char
+ isPlayerAtPos(BoardPosition, char) : boolean = <default>
+ checkTie(void) : boolean = <default>
+ toString(void) : String
AbsGameBoard
+ toString(void) : String
Extends
Extends
GameBoard
+ AMT_FOR_WIN : int[1]
+ USERROWS : int[1]
+ USERCOLS: int[1]
- board : char[2]
+ GameBoard(int, int, int) : void
+ getNumRows(void) : int
+ getNumCols(void) : int
+ getNumToWin(void) : int
+ placeToken(char, int) : void
+ whatsAtPos(BoardPosition) : char
GameBoardMem
+ AMT_FOR_WIN : int[1]
+ USERROWS : int[1]
+ USERCOLS: int[1]
- board : Map<Character, List<BoardPosition>>
+ GameBoardMem(int, int, int) : void
+ getNumRows(void) : int
+ getNumCols(void) : int
+ getNumToWin(void) : int
+ placeToken(char, int) : void
+ whatsAtPos(BoardPosition) : char
+ isPlayerAtPos(BoardPosition, char) : boolean

(IGameBoard) boolean checkIfFree(int c)

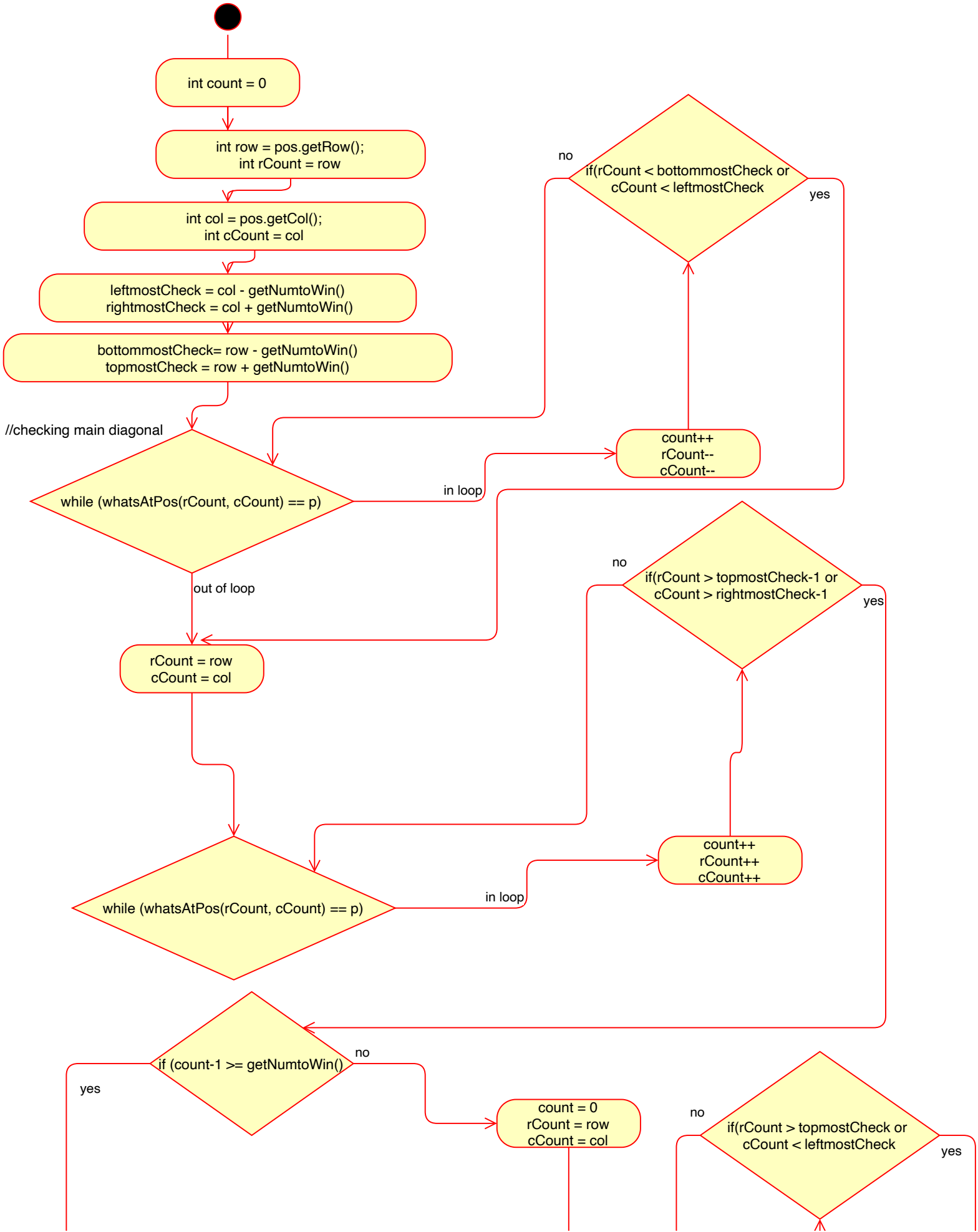


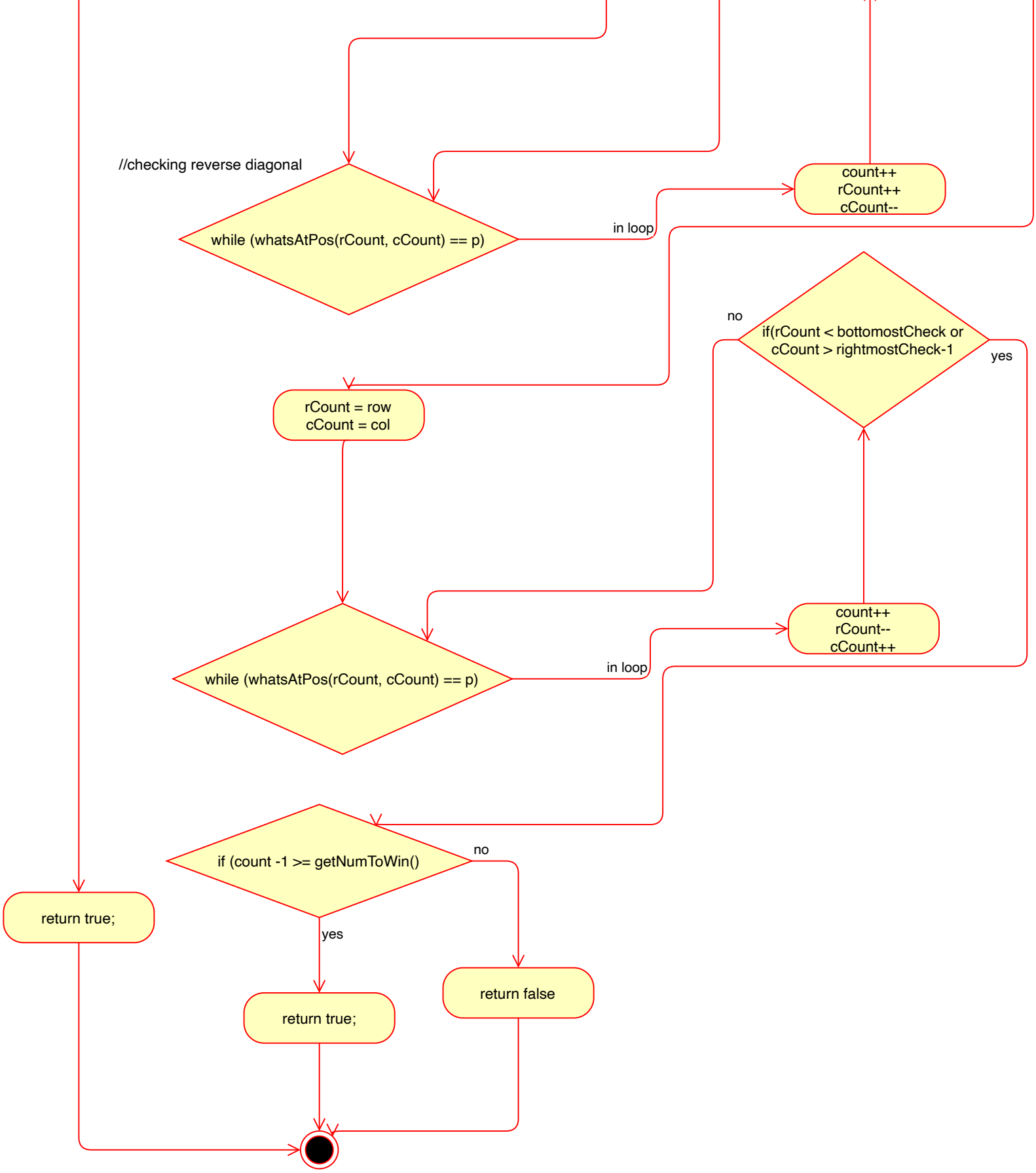

(IGameBoard) boolean checkForWin(int c)
char player;
row = MAX ROW - 1
for (i = 0; i < getNumRows
in loop
out of loop
WhatsAtPos(BoardPosition(i,c)) is blank space
no
yes
i++
row--;
player = board[row][c]
BoardPosition temp = new BoardPosition(row, col)
if (checkHorizWin(temp, player))
no
yes
return true
if(checkVertWin(temp, player))
no
yes
return true
if(checkDiagWin(temp,player))
no
yes
return true;
return false;


boolean checkHorizWin(BoardPosition pos, char p)
counter = 0
marker = 0
col = pos.getCol
leftmostCheck = col-getNumtoWin()
rightmostCheck = col+getNumtoWin()
if (leftmostCheck < 0)
yes
leftmostCheck = 0
no
if(rightmostCheck > getNumCols
yes
rightmostCheck = getNumCols
no
for (i = leftmostCheck; i < rightMostcheck
yes
out of loop
if (whatsAtPos(new BoardPosition(getRow, i)) == p
no
yes
counter++;
counter = 0
if (i == pos.getCol)
no
yes
marker = 1
if (counter >= getNumToWin() && marker == 1)
no
yes
return true;
return false
i++

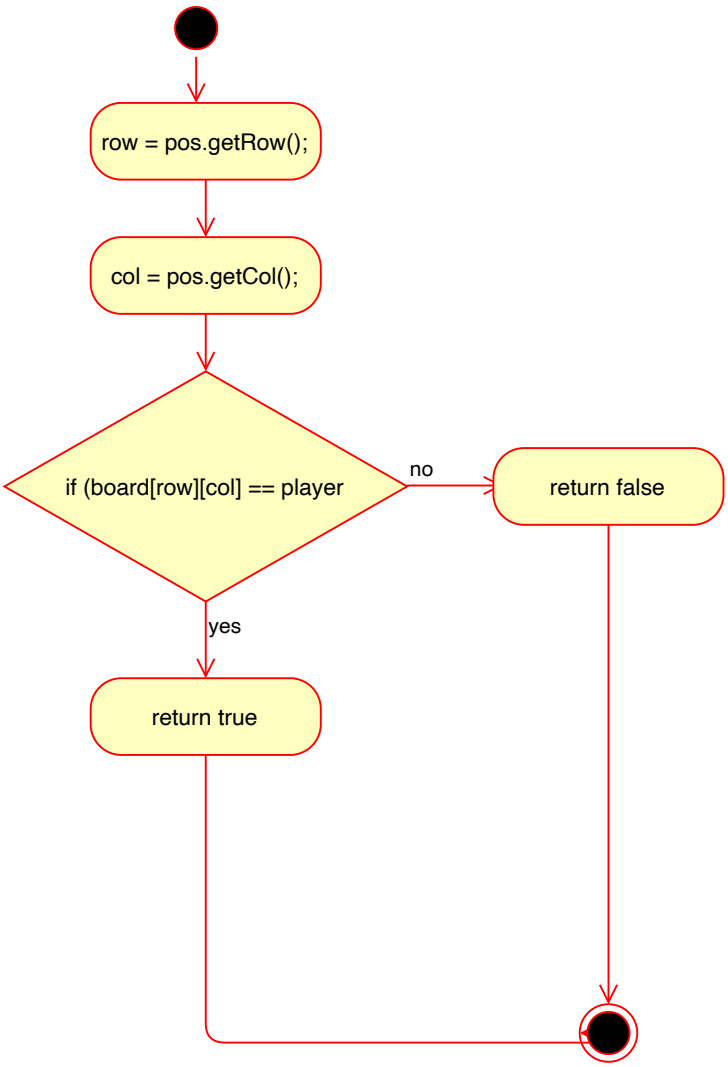

boolean checkVertWin(BoardPosition pos, char p)
counter = 0
marker = 0
row = pos.getRow
bottommostCheck = row-getNumtoWin()
topmostCheck = row+getNumtoWin()
if (bottommostCheck < 0)
yes
bottommostCheck = 0
no
if(topmostCheck > getNumCols
yes
topmostCheck = getNumRows
no
for (i = bottommostCheck; i < topmostCheck
yes
out of loop
if (whatsAtPos(new BoardPosition(i,getCol)) == p
no
yes
counter++;
counter = 0
if (i == pos.getRow)
no
yes
marker = 1
if (counter >= getNumToWin() && marker == 1)
no
yes
i++
return true;
return false

(GameBoard) boolean checkDiagWin(BoardPosition pos, char p)

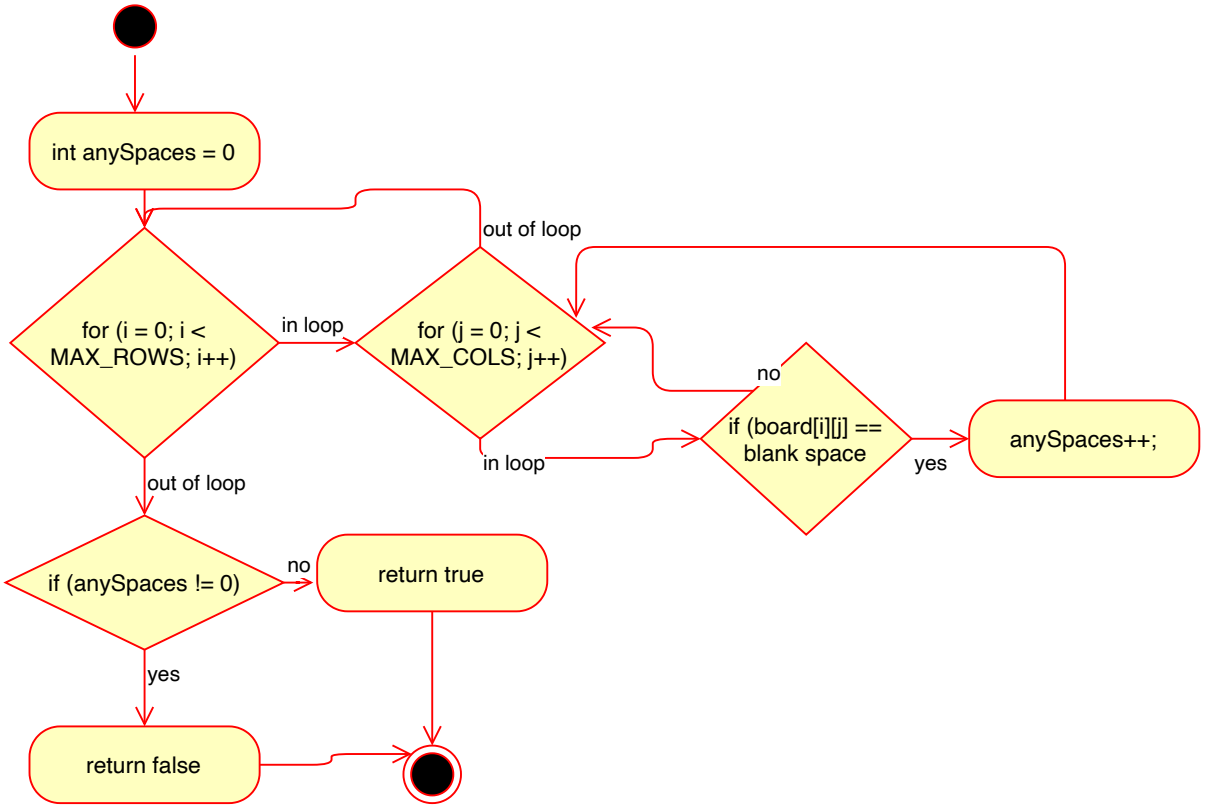



//checking reverse diagonal
while (whatsAtPos(rCount, cCount) == p)
in loop
count++
rCount++
cCount--
rCount = row
cCount = col
no
if(rCount < bottomostCheck or
cCount > rightmostCheck-1
yes
while (whatsAtPos(rCount, cCount) == p)
in loop
count++
rCount--
cCount++
if (count -1 >= getNumToWin()
no
yes
return true;
return false
return true;

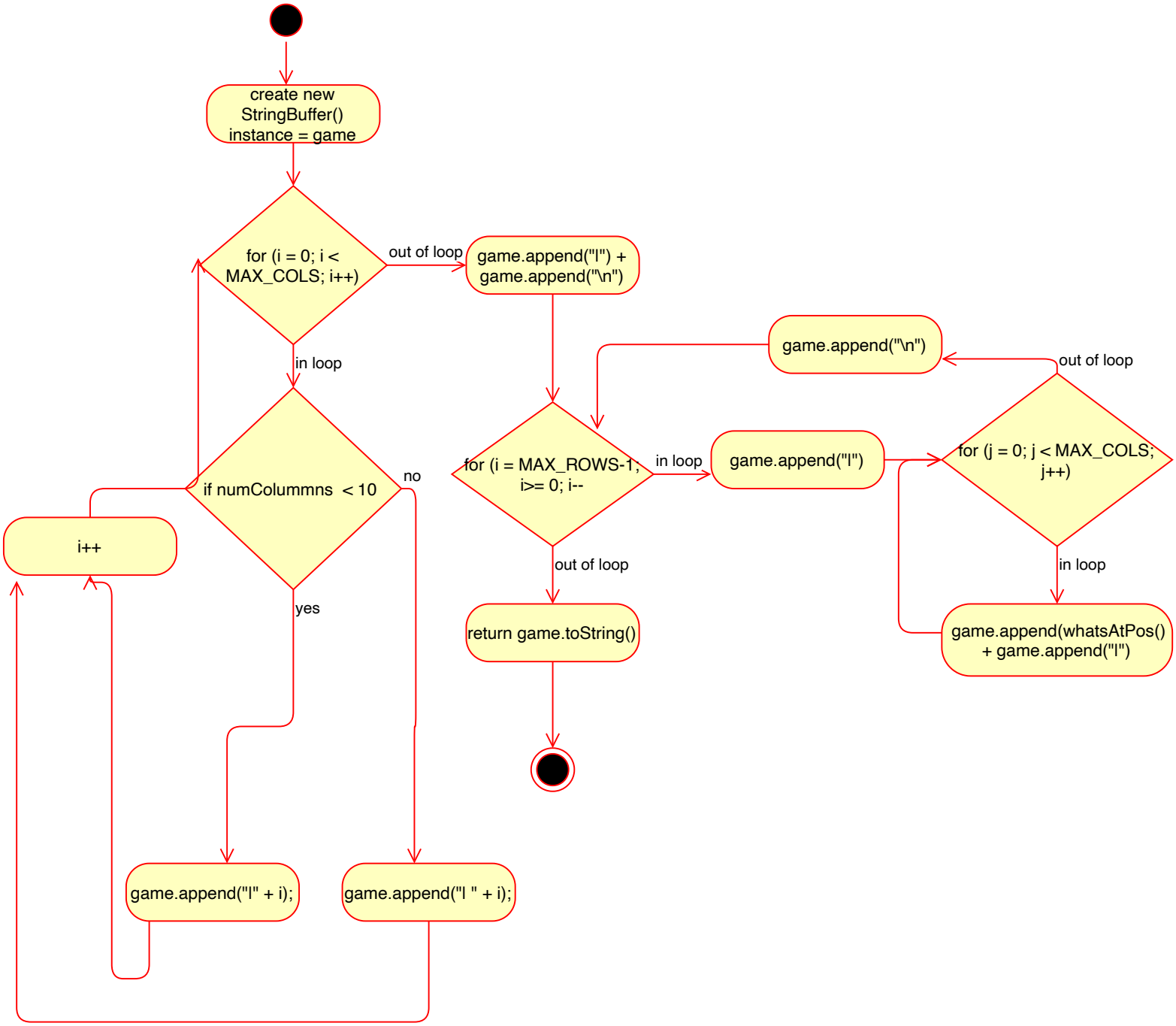
(IGameBoard) boolean isPlayerAtPos(BoardPosition pos, char player)

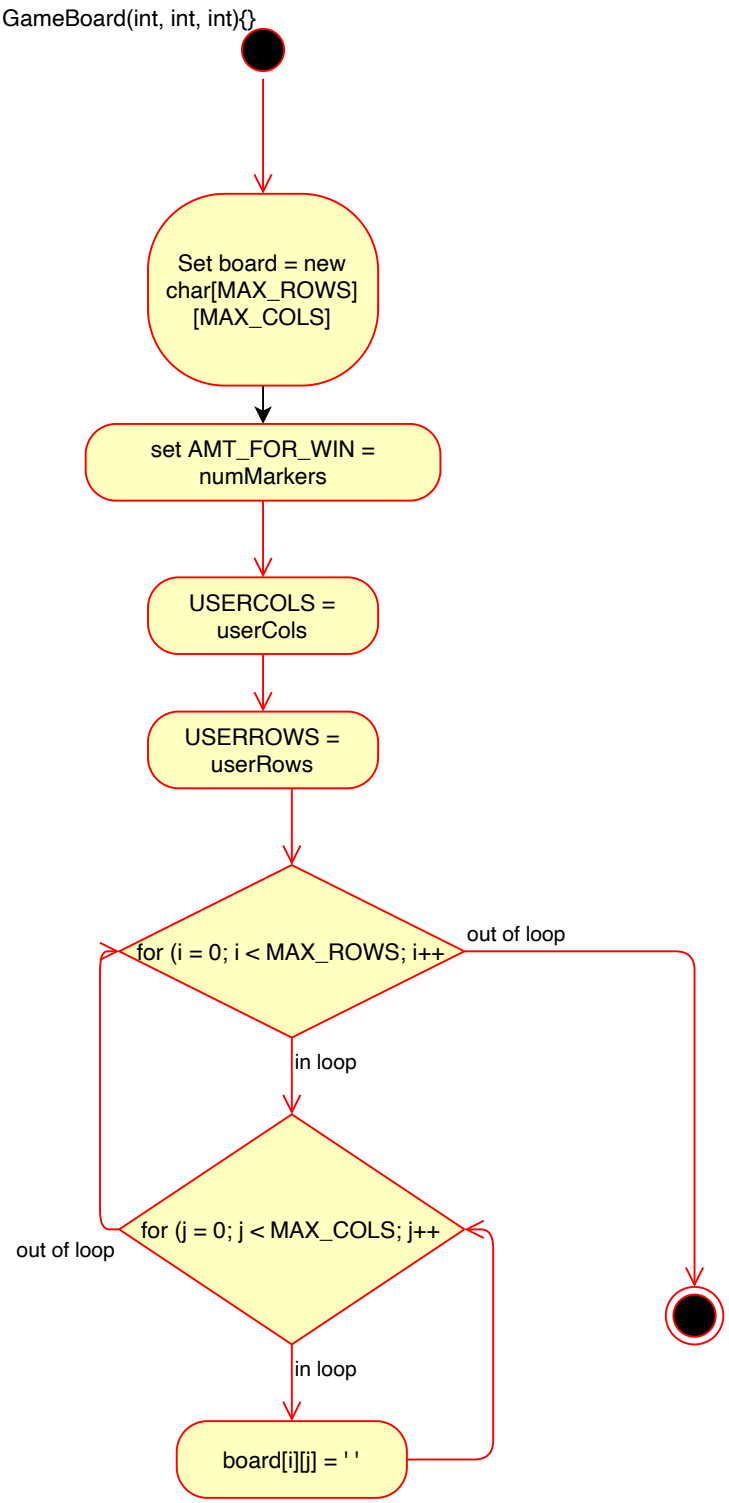


(IGameBoard) boolean checkTie()

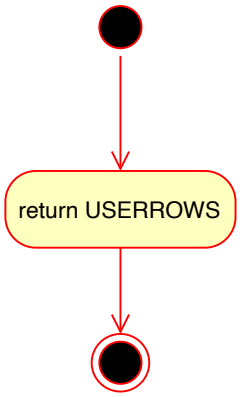


(absGameBoard) String toString()

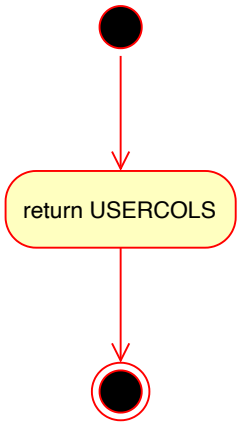



GameBoard(int, int, int){}
Set board = new char[MAX_ROWS] [MAX_COLS]
set AMT_FOR_WIN = numMarkers
USERCOLS = userCols
USERROWS = userRows
for (i = 0; i < MAX_ROWS; i++
out of loop
in loop
for (j = 0; j < MAX_COLS; j++
out of loop
in loop
board[i][j] = ' '

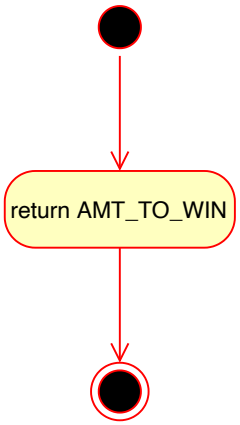
(GameBoard) int getNumRows()



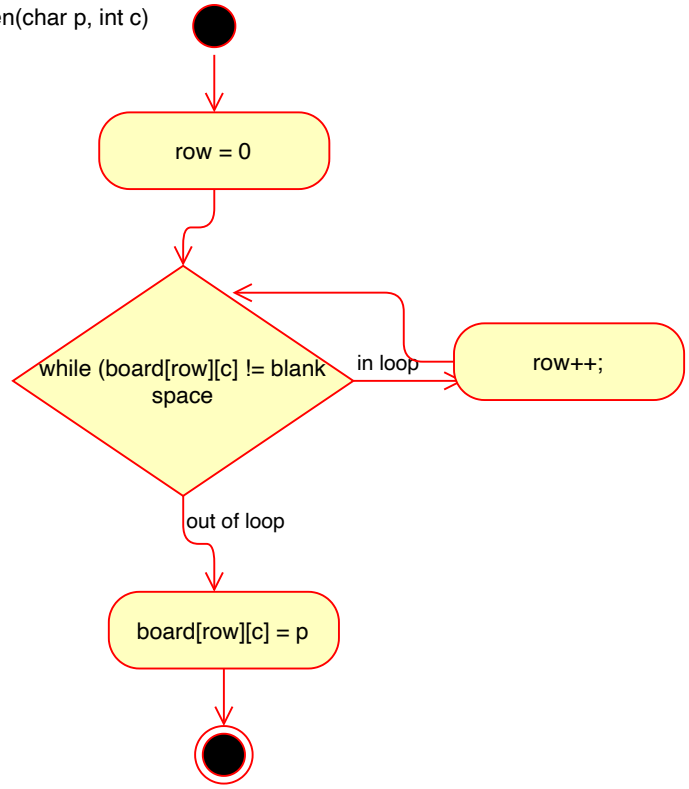
(GameBoard) int getNumColumns()



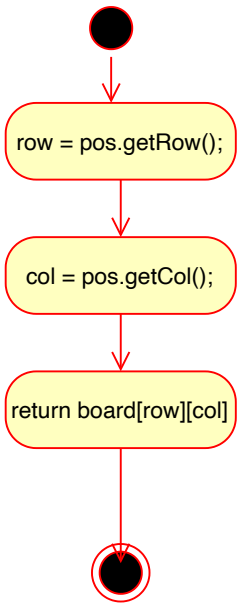
(GameBoard) int getNumToWin()



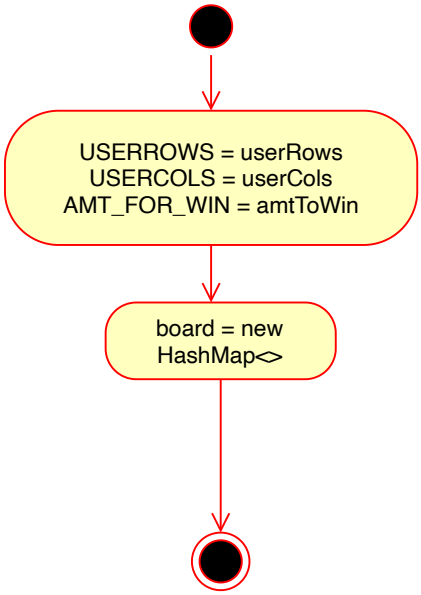
(GameBoard) void placeToken(char p, int c)



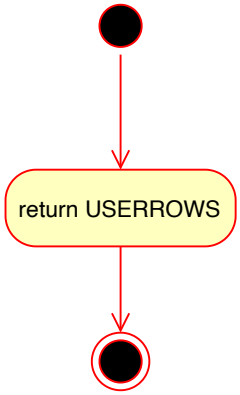
(GameBoard)char whatsAtPos



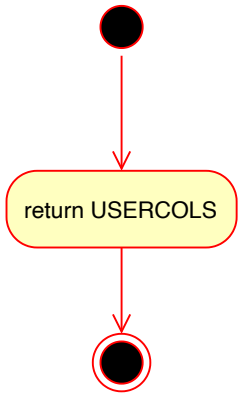
GameBoardMem(int, int, int, char[])



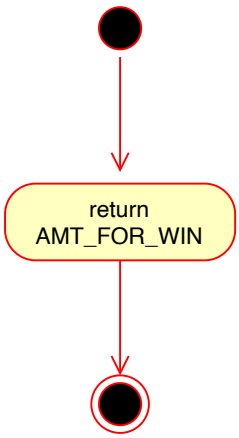
(GameBoardMem) int getNumRows()



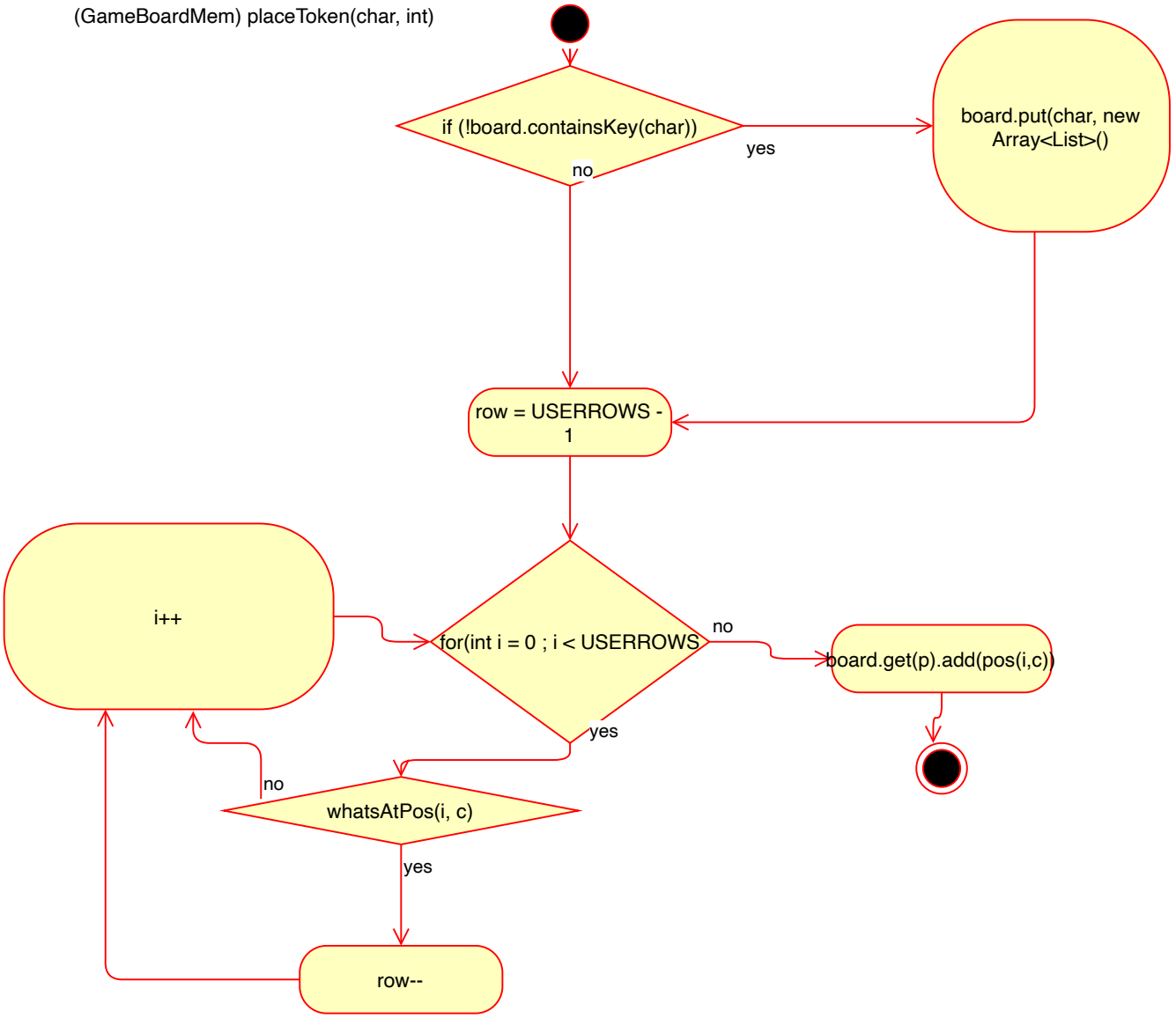
(GameBoardMem) int getNumColumns()



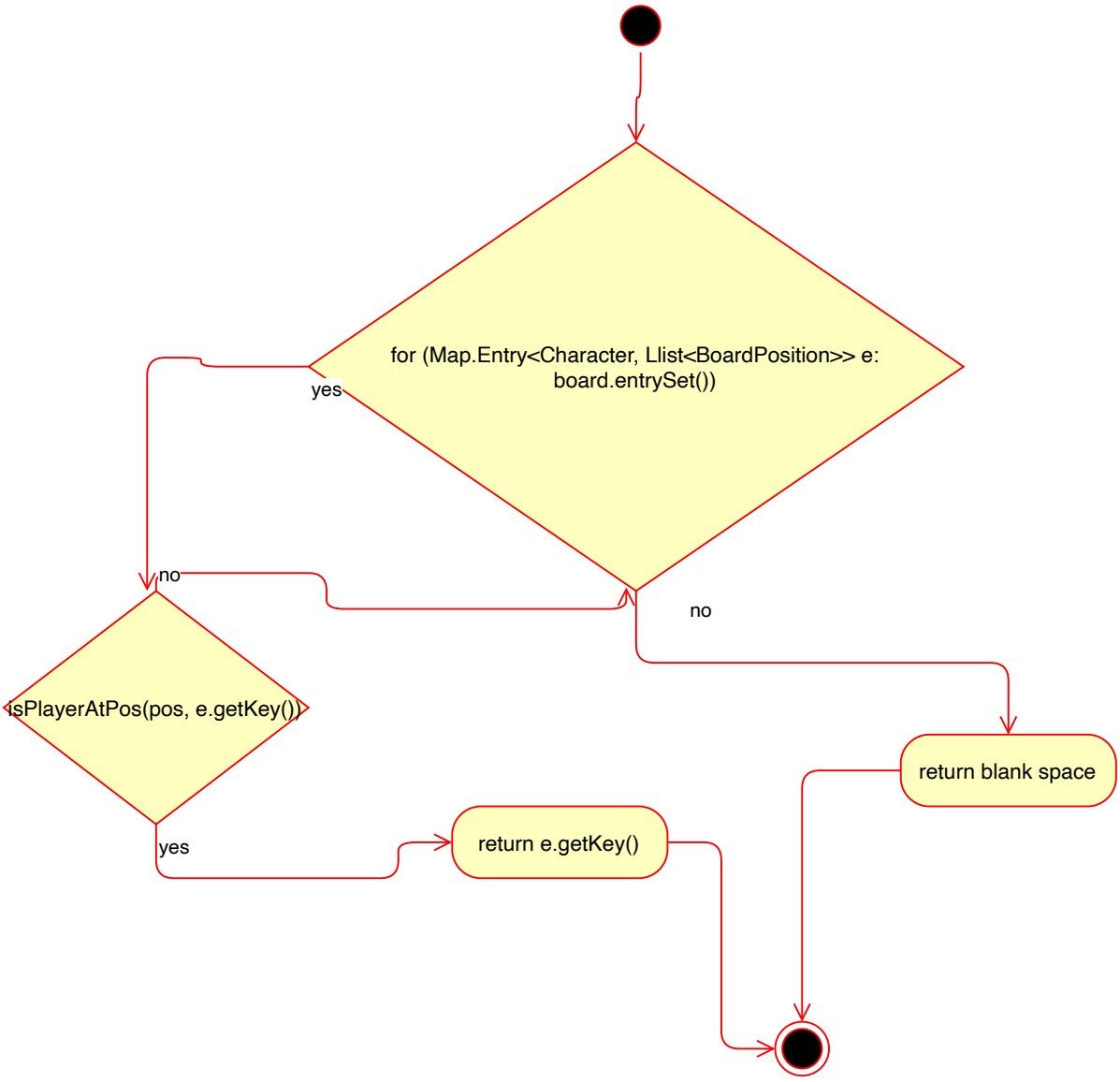
(GameBoardMem) int getNumToWin()



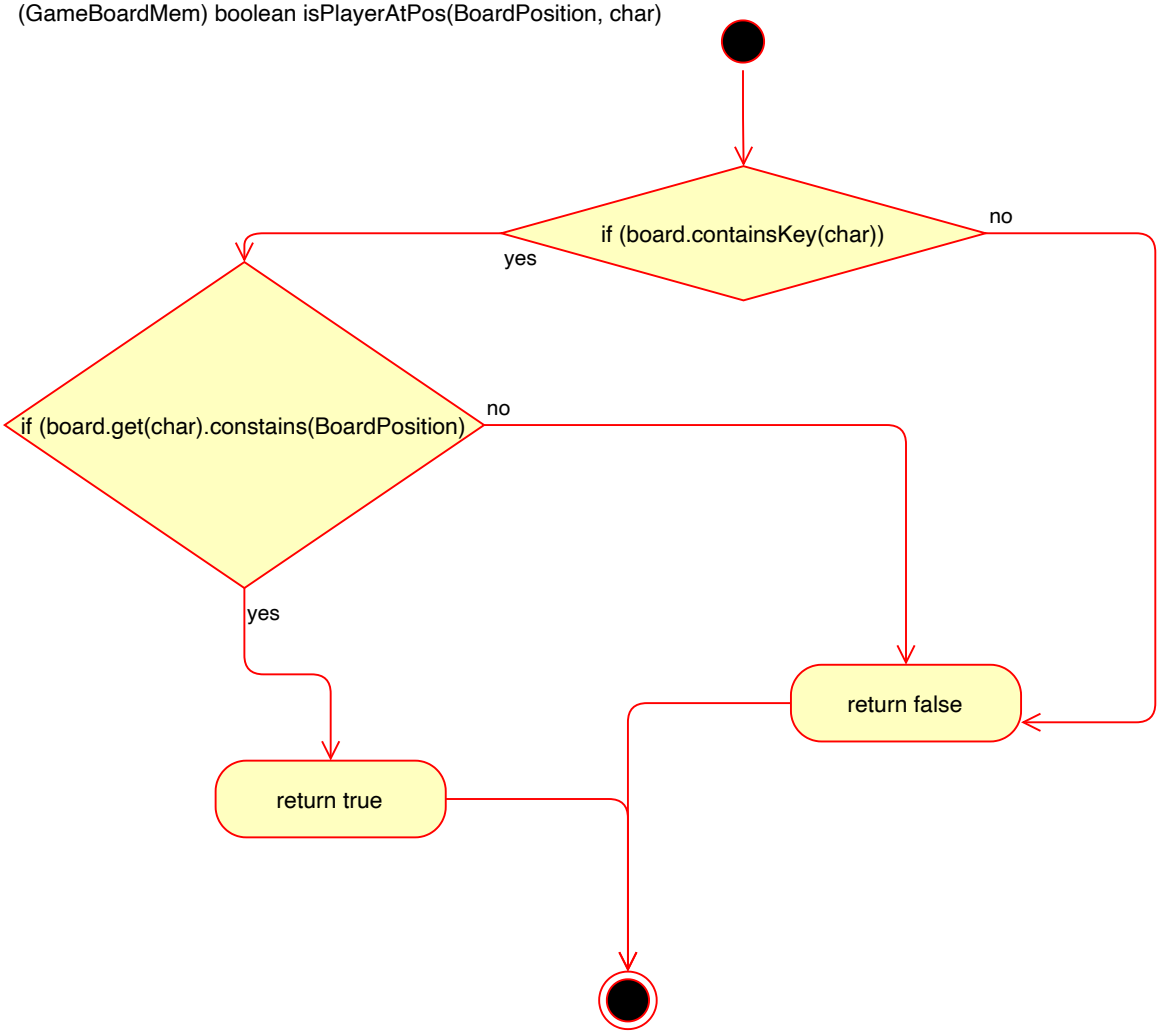
(GameBoardMem) placeToken(char, int)

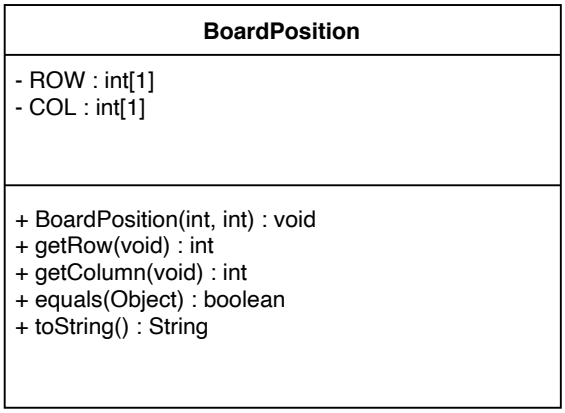


(GameBoardMem) whatsAtPos(BoardPosition)

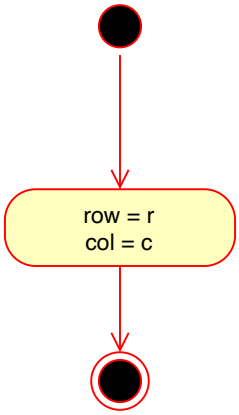


(GameBoardMem) boolean isPlayerAtPos(BoardPosition, char)

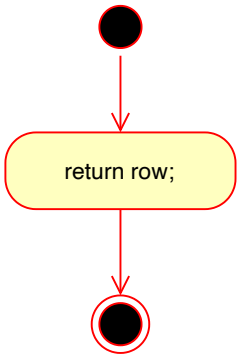



BoardPosition
- ROW : int[1]
- COL : int[1]
+ BoardPosition(int, int) : void
+ getRow(void) : int
+ getColumn(void) : int
+ equals(Object) : boolean
+ toString() : String

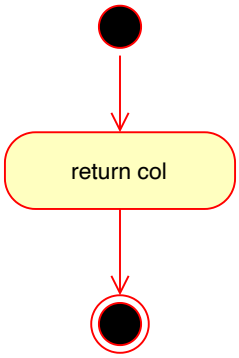
(BoardPosition)BoardPosition(int r, int c)



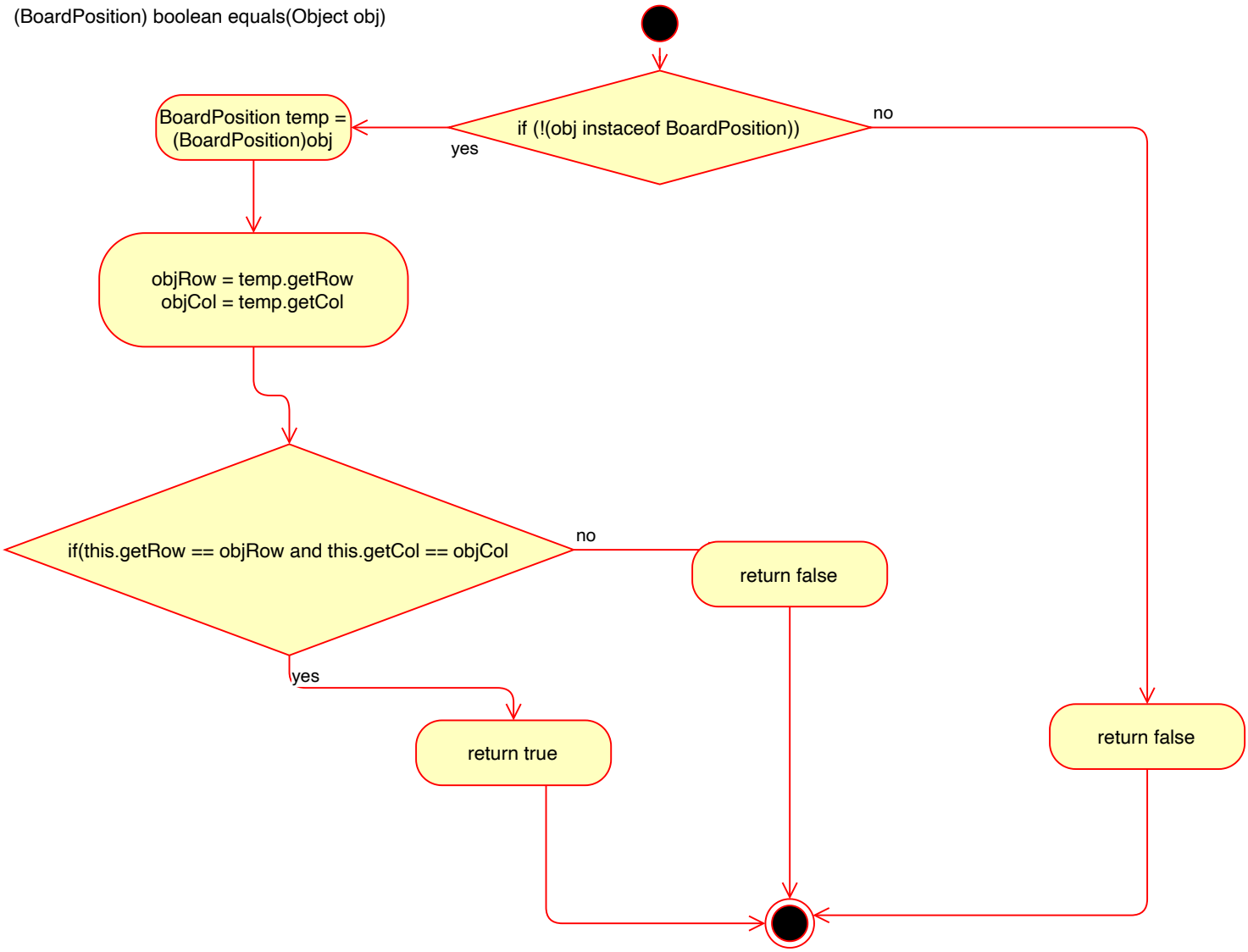
(BoardPosition) int getRow()



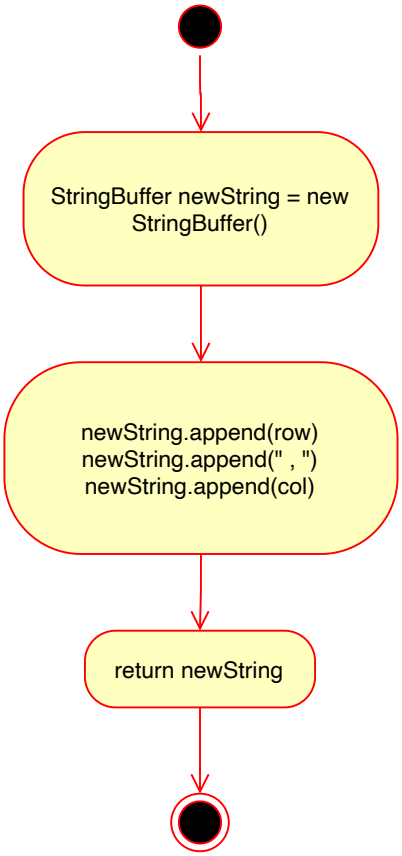
(BoardPosition) int getCol()



(BoardPosition) boolean equals(Object obj)



(BoardPosition) String toString()



Testing:

constructor(int r, int c, int num) (x3)

Input	Output	Reason

r = 12
c = 12
num = 4
	State:
	12x12 gameboard of blank spaces
	
	getNumRows = 12;
	getNumCols = 12;
	getNumToWin = 4;
		This function is unique and distinct because it represents a routine test case of a board that is not the minimum or maximum for rows, columns or amount to win.
		
		Function:
		testConstructor_not_extreme_inputs

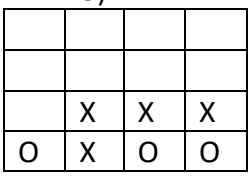
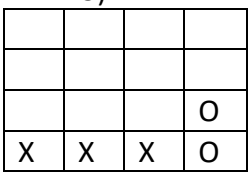
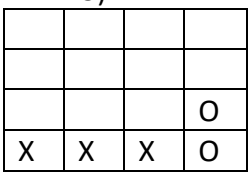
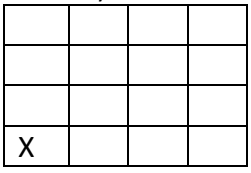
r = 100;
c = 100;
num = 25;
	State:
	100x100 gameboard of blank spaces
	
	getNumRows = 100;
	getNumCols = 100;
	getNumToWin = 25;
		This function is unique and distinct because it represents a boundary test case of a board that is the maximum for rows, columns and amount to win.
		
		Function:
		testConstructor_maximum_inputs

r = 3
c = 3
num = 3
	State:
	3x3 gameboard of blank spaces
	
	getNumRows = 3;
	getNumCols = 3;
	getNumToWin = 3;
		This function is unique and distinct because it represents a boundary test case of a board that is the maximum for rows, columns and amount to win.
		
		Function:
		testConstructor_minimum_inputs

boolean checkIfFree(int c) (x3)

Input	Output	Reason
State: c = 0	checkIfFree = true state of the board is unchanged	This test case is unique and distinct, because it represents a routine test case where the column is not empty but not full. Function: testCheckIfFree_middle_of_column
State: c = 1	checkIfFree = true state of the board is unchanged	This test case is unique and distinct because it represents a boundary test case of the first marker in the column and the board in general. Function: testCheckIfFree_beginning_of_column
State: c = 0	checkIfFree = false state of the board is unchanged	This test case is unique and distinct because it represents a boundary test case of the column already being full. Function: testCheckIfFree_column_already_full

boolean checkHorizWin(BoardPosition pos, char p) (x4)

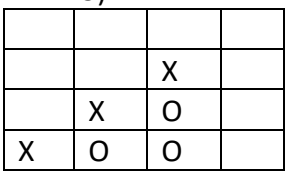
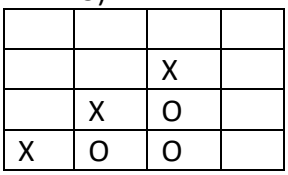
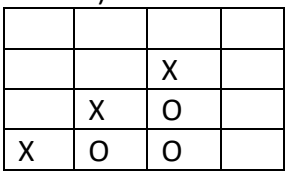
Input	Output	Reason
State: (number to win = 3)  pos = new BoardPosition(1,3) char = X	checkHorizWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed on the complete right of the string of 3 consecutive X's, so the function needs to count X's to the left. Function: testCheckHorizWin_win_last_marker_right
State: (number to win = 3)  pos = new BoardPosition(0,0) char = X	checkHorizWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed on the complete left of the string of 3 consecutive X's, so the function needs to count X's to the right. Function: testCheckHorizWin_win_last_marker_left
State: (number to win = 3)  pos = new BoardPosition (0,1) char = X	checkHorizWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed in the middle of the string of 3 consecutive X's, so the function needs to count X's to the left and right. Function: testCheckHorizWin_win_last_marker_middle
State: (number to win = 3)  pos = new BoardPosition(0,0) char = X	checkHorizWin = false state of the board is unchanged	This test case is unique and distinct because the last X was placed where it does not cause a string of 3 consecutive X's and therefore does not cause player X to win. Function: testCheckHorizWin_noWin_last_marker_first_in_string

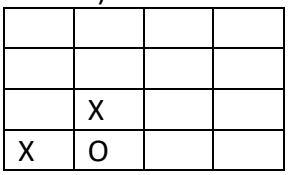
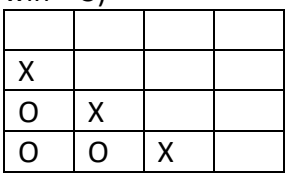
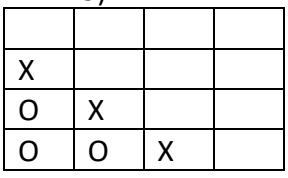
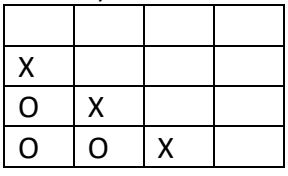
boolean checkVertWin(BoardPosition pos, char p) (x4)

Input	Output	Reason
State: (number to win = 3) | | | | | | | X | | | | | X | O | | | | X | O | | | pos = new BoardPosition(2, 0) char = X	checkVertWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed on the top of the string of 3 consecutive X's, so the function needs to count X's toward the bottom of the board. Function: testVertWin_win_last_marker_top
State: (number to win = 3) | | | | | | | | | | | | | | | | | | X | | | pos = new BoardPosition(0, 1) char = X	checkVertWin = false state of the board is unchanged	This test case is unique and distinct because the last X was placed in the beginning of a row such that it is the only X in a string of consecutive X's and therefore does not cause player X to win. Function: testVertWin_noWin_last_marker_first_in_string
State: (number to win = 3) | | | | | | | X | | | | | O | | | | | X | | | | pos = new BoardPosition(2, 0) char = X	checkVertWin = false state of the board is unchanged	This test case is unique and distinct because the last X was placed at the top of the column making a string of 3 consecutive characters. The characters in the string are not the same and therefore does not cause player X to win. Function: testVertWin_noWin_last_marker_last_in_non_consecutive_string
State: (number to win = 3) | X | O | X | O | | O | X | O | X | | O | X | O | X | | X | O | X | O |	checkVertWin = false state of the board is unchanged	This test case is unique and distinct because the last O was placed at the top of the column in the last row making the board completely full. There are no vertical wins in this board and should be a tie. Function:

pos = new BoardPosition(3, 3) char = O		testVertWin_noWin_last_marker_makes_full_board

boolean checkDiagWin(BoardPosition pos, char p) (x7)

Input	Output	Reason
State: (number to win = 3)  pos = new BoardPosition(2,2) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the top-rightmost of the string of 3 consecutive X's, so the function needs to count X's on the main diagonal toward the bottom-left of the board. Function: testDiagWin_win_last_marker_right_mainDiag
State: (number to win = 3)  pos = new BoardPosition(1,1) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the middle of a string of 3 consecutive X's, so the function needs to count X's on the main diagonal both toward the bottom-left and top-right of the board. Function: testDiagWin_win_last_marker_middle_mainDiag
State: (number to win = 3)  pos = new BoardPosition(0,0) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the bottom-leftmost of a string of 3 consecutive X's, so the function needs to count X's on the main diagonal toward the top-right of the board. Function: testDiagWin_win_last_marker_left_mainDiag

State: (number to win = 3)  pos = new BoardPosition(1,1) char = X	checkDiagWin = false state of the board is unchanged	This test case is unique and distinct because the last X was placed at the top-right of the string of 2 consecutive X's. There is not enough consecutive X's for a win, so there is not a win. Function: testDiagWin_noWin_last_marker_right_in_two
State: (number to win = 3)  pos = new BoardPosition(2,0) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the top-leftmost of the string of 3 consecutive X's, so the function needs to count X's on the reverse diagonal toward the bottom-right of the board. Function: testDiagWin_win_last_marker_left_reverseDiag
State: (number to win = 3)  pos = new BoardPosition(1,1) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the middle of the string of 3 consecutive X's, so the function needs to count X's on the reverse diagonal toward both the bottom-right and top-left of the board. Function: testDiagWin_win_last_marker_mid_reverseDiag
State: (number to win = 3)  pos = new BoardPosition(0,2) char = X	checkDiagWin = true state of the board is unchanged	This test case is unique and distinct because the last X was placed at the bottom-rightmost of the string of 3 consecutive X's, so the function needs to count X's on the reverse diagonal toward to top-left of the board. Function: testDiagWin_win_last_marker_right_reverseDiag

boolean checkTie() (x4)

<table>
<tr><th>Input</th><th>Output</th><th>Reason</th></tr>
<tr><td>State: (number to win = 3)
<table><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><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></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr></table></td><td>checkTie = true

state of the board is unchanged</td><td>This test case is unique and distinct because the last marker was placed such that the entire board is filled without a win, thus resulting in a tie.

Function: testCheckTie_tie_full_board</td></tr>
<tr><td>State: (number to win = 3)
<table><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td></td></tr></table></td><td>checkTie = false

state of the board is unchanged</td><td>This test case is unique and distinct because the last marker was placed such that the board not close to being filled but not empty, and the columns and rows are also not full.

Function: testCheckTie_nonFull_board</td></tr>
<tr><td>State: (number to win = 3)
<table><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr></table></td><td>checkTie = false

state of the board is unchanged</td><td>This test case is unique and distinct because the last marker was placed such that row 0 became full and the function needs to be aware that a full row doesn’t cause a tie.

Function: testCheckTie_fullRow</td></tr>
<tr><td>State: (number to win = 3)
<table><tr><td>X</td><td></td><td></td><td></td></tr><tr><td>O</td><td></td><td></td><td></td></tr><tr><td>O</td><td></td><td></td><td></td></tr><tr><td>X</td><td></td><td></td><td></td></tr></table></td><td>checkTie = false

state of the board is unchanged</td><td>This test case is unique and distinct because the last marker was placed such that column 0 became full and the function needs to be aware that a full column doesn’t cause a tie.

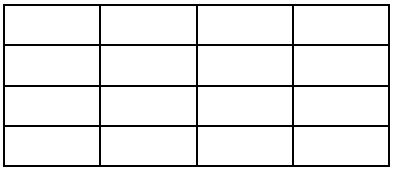
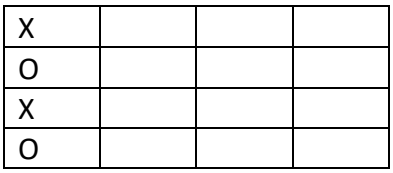
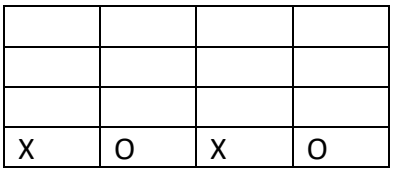
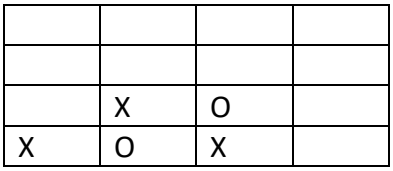
Function: testCheckTie_fullColumn</td></tr>
</table>

char whatsAtPos(BoardPosition pos) (x5)

Input	Output	Reason
State: pos = new BoardPosition(1,1)	whatsAtPos = ' ' state of the board is unchanged	This test case is unique and distinct because it is getting the position on a completely blank board and should return a blank space. Function: testWhatsAtPos_blank_board
State: X O X O pos = new BoardPosition(1,0)	whatsAtPos = 'X' state of the board is unchanged	This test case is unique and distinct because it is getting the position from a completely full column. Function: testWhatsAtPos_full_column
State: X O X O pos = new BoardPosition(0,1)	whatsAtPos = 'O' state of the board is unchanged	This test case is unique and distinct because it is getting the position from a completely full column. Function: testWhatsAtPos_full_row
State: X O X pos = new BoardPosition(1,1)	whatsAtPos = 'X' state of the board is unchanged	This test case is unique and distinct because it is getting a position that is in a column that is not empty, but not full and a row that is not empty, but not full. Function: testWhatsAtPos_nonFull_col_row
State: X O X O O X O X O X O X X O X O	whatsAtPos = 'X' state of the board is unchanged	This test case is unique and distinct because it is getting a position from a full board and at the top of a full column. Function:

pos = new BoardPosition(3,2)		testWhatsAtPos_full_board

boolean isPlayeratPos(BoardPosition pos, char player) (x5)

Input	Output	Reason
State:  pos = new BoardPosition(2,2) char = 'X'	isPlayeratPos = false state of the board is unchanged	This test case is unique and distinct because it is checking if there is a player in an empty board, which there is not. Function: testIsPlayerAtPos_empty_board
State:  pos = new BoardPosition(3,0) char = 'X'	isPlayeratPos = true state of the board is unchanged	This test case is unique and distinct because it is checking if there is a player, X, in the top position of a full column Function: testIsPlayerAtPos_full_col
State:  pos = new BoardPosition(0,3) char = 'O'	isPlayeratPos = true state of the board is unchanged	This test case is unique and distinct because it is checking if there is a player, O, in the last position of a full row. Function: testIsPlayerAtPos_full_row
State:  pos = new BoardPosition(1,1) char = 'X'	isPlayeratPos = true; state of the board is unchanged	This test case is unique and distinct because it is checking if there is a player, X, in a position that is not an empty or full row or column. Function: testIsPlayerAtPos_nonFull_row_col

State: X O X O O X O X O X O X X O X O pos = new BoardPosition(3,3) char = 'O'	isPlayerAtPos = true; state of the board is unchanged	This test case is unique and distinct because it is checking if there is a player, O, in a position that is in the top right of a full board. Function: testIsPlayerAtPos_full_board

void placeToken(char p, int c) (x5)

Input	Output	Reason
State: p = X c = 3	State: X	This test case is unique and distinct because it is adding a marker to an empty board and an empty column. Function: testPlaceToken_empty_board
State: O X O X p = X c = 2	State: X O X O X	This test case is unique and distinct because it is adding a marker to a column that is not empty, but not close to being full. Function: testPlaceToken_col_not_empty
State: X O X p = O c = 3	State: X O X O	This test case is unique and distinct because it is adding a marker to a row that will make the row full. Function: testPlaceToken_make_row_full
State: X O X O X	State: O X O X O X	This test case is unique and distinct because it is adding a marker to a column that will make the column full. Function:

<table>
<tr>
<td>p = O
c = 2</td>
<td></td>
<td>testPlaceToken_make_col_full</td>
</tr>
<tr>
<td>State:
<table><tr><td>X</td><td>O</td><td>X</td><td></td></tr><tr><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></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr></table>
p = O
c = 3</td>
<td>State:
<table><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><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></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td></tr></table></td>
<td>This test case is unique and distinct because it is adding a marker to a board that will make the board full.

Function: testPlaceToken_make_board_full</td>
</tr>
</table>

Deployment:

There is no makefile for this program. Use GUI to run program.