Project 4: ConnectX
Abigail Poropatich
CPSC 2150

Functional Requirements:

1. As a user, I can have a maximum of 100 columns
2. As a user, I can have a minimum of 3 columns
3. As a user, my token has to be placed at the bottom of the board
4. As a user, I can place my token in any column
5. As a user, I can have up to 10 players
6. As a user, if I place my token in a full column, I will be asked to choose an available column
7. As a user, I must have a minimum of 2 players
8. As a user, if I have filled all spaces on the board, it is a tie
9. As a user, I can have a maximum of 25 in a row to win
10. As a user, I can have a minimum of 3 in a row to win
11. As a user, if I place my token in a non-existent, I will be asked to choose an available column
12. As a user, when there is a win, I will choose whether to play again or exit
13. As a user, when there is a tie, I will choose whether to play again or exit
14. As a user, I must enter ‘N’ or ‘n’ in order to exit a game
15. As a user, I must enter ‘Y’ or ‘y’ in order to replay a game
16. As a user, I must chose between a fast game or a memory-efficient game
17. As a user, I must enter ‘F’ or ‘f’ in order to play a fast game
18. As a user, I must enter ‘M’ or ‘m’ in order to play a memory-efficient game
19. As a user, if I place a token that is not mine, then I will be prompted again
20. As a user, I can win if I place the number of tokens in a row horizontally
21. As a user, I can win if I place the number of tokens in a row diagonally
22. As a user, I can win if I place the number of tokens in a row vertically
23. As a user, I can pick any character on the keyboard to represent my token
24. As a user, if I choose a token that has already been chosen, I will be asked to choose an available one

Non-functional requirements

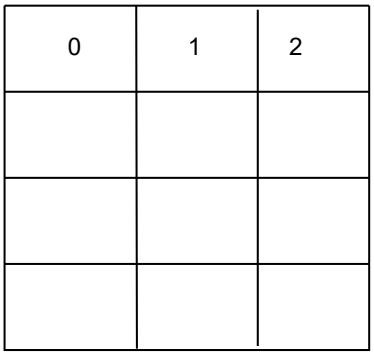
1. Must compile and run on Unix
2. Must be written in Java
3. Must be written in IntelliJ
4. (0,0) must be the bottom left of the board
5. Player 1 must always go first
6. Amount of tokens in a row must always be between 3 and 25, inclusive

7. Amount of columns must be between 3 and 100 inclusive
8. Amount of rows must be between 3 and 100 inclusive
9. Amount of players must be between 2 and 10 inclusive
10. Javadoc comments and contracts must be written for each method

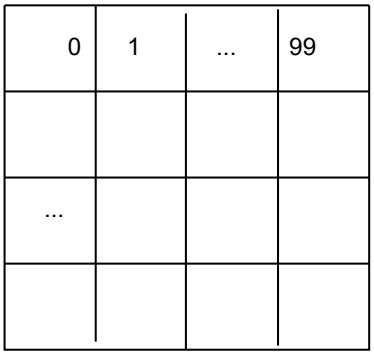
Deployment:

1. To compile the program, the user should enter the command ‘make’
2. To run the program, the user should enter the command ‘make run’
3. To clean the .class files from the program, the user should enter the command ‘make clean’
4. To compile the test cases, the user should enter the command ‘test’
5. To run the GameBoard test cases, the user should enter the command ‘make testGB’
6. To run the GameBoardMem test cases, the user should enter the command ‘make testGBmem’

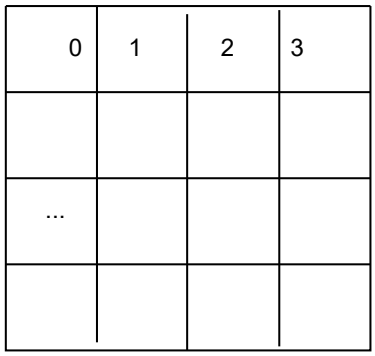
GameBoard(int NumRows, int NumCOlumns, int NumToWin)

Input:	Output state:	Reason:
NumRows = 3 NumColumns = 3 NumToWin = 3		This test case is unique because it checks to see if the constructor can initialize the board the smallest possible dimensions Function: testingConstructor_min

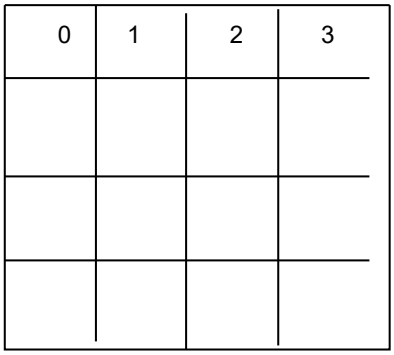
GameBoard(int NumRows, int NumColumns, int NumToWin)

Input:	Output state:	Reason:
NumRows = 100 NumColumns = 100 NumToWin = 25		This test case is unique because it checks to see if the constructor can initialize the board the largest possible dimensions Function: testingConstructor_max

GameBoard(int NumRows, int NumColumns, int NumToWin)

Input:	Output state:	Reason:
NumRows = 4 NumColumns = 3 NumToWin = 3		This test case is unique because it checks to see if the constructor can initialize the board to a rectangular shape rather than a perfect square Function: testingConstructor_abstract

boolean checkIfFree(int c)

Input:	Output state:	Reason:
	checkIfFree(1) == true Board remains unchanged	This test case is unique because the board is empty Function: testingCheckIfFree_empty

boolean checkIfFree(int c)

Input:

0	1	2	3
X			
O			
X			

Output state:
checkIfFree(0) == false

Board remains unchanged

Reason:

This test case is unique because the column is full

Function:
testingCheckIfFree_full

boolean checkIfFree(int c)

Input:

0	1	2	3
X			
O		O	

Output state:
checkIfFree(2) == true

Board remains unchanged

Reason:

This test case is unique because its tests a board with characters already in the column being tested

Function:
testingCheckIfFree_some

boolean checkHorizWin(BoardPosition pos, char p)

Input state: pos = <0,1>, p = 'O'

0	1	2	3

State: board.checkNumToWin() = 3;

Output:
checkHorizWin(pos, p) == false

Board remains unchanged

Reason:

This test case is unique because the board is empty

Function:
testingCheckHorizWin_empty

boolean checkHorizWin(BoardPosition pos, char p)

Input state: pos = <0,1>, p = 'X'

0	1	2	3
	X		

State: board.checkNumToWin() = 3;

Output:
checkHorizWin(pos, p) == false

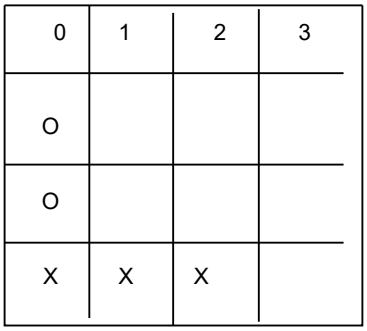
Board remains unchanged

Reason:

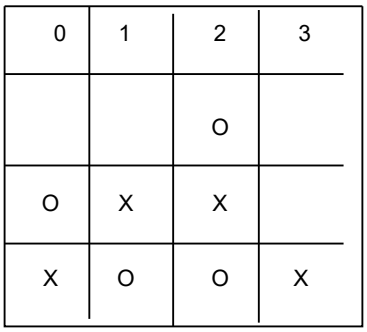
This test case is unique because there is only one character on the board

Function:
testingCheckHorizWin_one

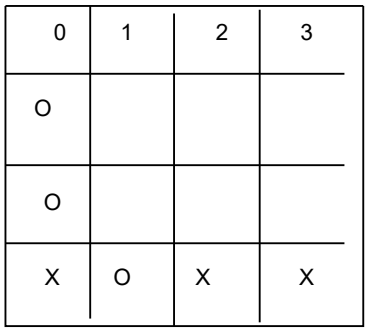
boolean checkHorizWin(BoardPosition pos, char p)

Input state: pos = <0,2>, p = 'X'	Output:	Reason:
	checkHorizWin(pos, p) == true Board remains unchanged	This test case is unique because there is a horizontal winner Function: testingCheckHorizWin_win
State: board.checkNumToWin() = 3;		

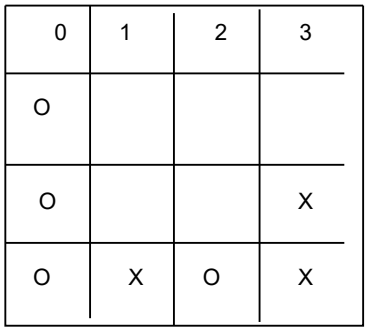
boolean checkHorizWin(BoardPosition pos, char p)

Input state: pos = <0,3>, p = 'X'	Output:	Reason:
	checkHorizWin(pos, p) == false Board remains unchanged	This test case is unique because there is no winner, but enough tokens place in a row to win, the are just not the same consecutive piece Function: testingCheckHorizWin_fauxwin
State: board.checkNumToWin() = 4;		

boolean checkVertWin(BoardPosition pos, char p)

Input state: pos = <3,0>, p = 'O'	Output:	Reason:
	checkVertWin(pos, p) == false Board remains unchanged	This test case is unique because there is no winner, but enough tokens place in a row to win, the are just not the same consecutive piece Function: testingCheckVertWin_fauxwin
State: board.checkNumToWin() = 3;		

boolean checkVertWin(BoardPosition pos, char p)

Input state: pos = <2,0>, p = 'O'	Output:	Reason:
	checkVertWin(pos, p) == true Board remains unchanged	This test case is unique because there is a winner Function: testingCheckVertWin_win
State: board.checkNumToWin() = 3;		

boolean checkVertWin(BoardPosition pos, char p)

Input state: pos = <3,1>, p = 'X'

0	1	2	3
	X		
O	X		
O	X		
O	X	O	

State: board.checkNumToWin() = 4;

Output:

checkVertWin(pos, p) == true

Board remains unchanged

Reason:

This test case is unique because there is a winner for a larger win value and a possible win for a lower win value right next to it

Function: testingCheckVertWin_winComp

boolean checkVertWin(BoardPosition pos, char p)

Input state: pos = <3,1>, p = 'X'

0	1	2	3
O			
O			
O	O		
X	X	X	X

State: board.checkNumToWin() = 4;

Output:

checkVertWin(pos, p) == false

Board remains unchanged

Reason:

This test case is unique because there is a winner for a different win condition, but not vertical.

Function: testingCheckVertWin_fauxwin

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,0>, p = 'X'

0	1	2	3

State: board.checkNumToWin() = 3;

Output:

checkDiagWin(pos, p) == false

Board remains unchanged

Reason:

This test case is unique because the board is empty

Function: testingCheckDiagWin_empty

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,2>, p = 'X'

0	1	2	3
X		O	
O	X		
X	O	O	X

State: board.checkNumToWin() = 3;

Output:

checkDiagWin(pos, p) == false

Board remains unchanged

Reason:

This test case is unique because there is no winner, but enough tokens place in a row to win, the are just not the same consecutive piece

Function: testingCheckDiagWin_nowin

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,3>, p = 'O'

0	1	2	3
			O
X		O	X
X	O	O	O
O	X	X	X

State: board.checkNumToWin() = 4;

Output:

checkDiagWin(pos, p) == true

Board remains unchanged

Reason:

This test case is unique because there is a winner

Function:
testingCheckDiagWin_win

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,3>, p = 'O'

0	1	2	3
O			
O			
O	X		
O	X	X	X

State: board.checkNumToWin() = 4;

Output:

checkDiagWin(pos, p) == false

Board remains unchanged

Reason:

This test case is unique because there is a winner for a different win condition, but not diagonal.

Function:
testingCheckDiagWin_fauxwin

boolean checkTie()

Input state: pos = <0,0>, p = 'X'

0	1	2	3

State: board.checkNumToWin() = 3;

Output:

checkTie() == false

Board remains unchanged

Reason:

This test case is unique because the board is empty

Function:
testingCheckTie_empty

boolean checkTie()

Input state: pos = <0,2>, p = 'X'

0	1	2	3
X		O	
O	X		
X	O	O	X

State: board.checkNumToWin() = 3;

Output:

checkTie() == false

Board remains unchanged

Reason:

This test case is unique because there is not enough tokens on the board to have a tie

Function:
testingCheckTie_notenough

boolean checkTie()

Input state: pos = <0,3>, p = 'O'

0	1	2	3
O	X	O	O
X	O	O	X
X	X	O	O
O	X	X	X

State: board.checkNumToWin() = 4;

Output:

checkTie() == true

Board remains unchanged

Reason:

This test case is unique because there is a tie

Function:
testingCheckTie_tie

boolean checkTie()

Input state: pos = <0,3>, p = 'O'

0	1	2	3
X		O	X
O	O	X	O
O	X	X	O
X	O	X	O

State: board.checkNumToWin() = 4;

Output:

checkTie() == false

Board remains unchanged

Reason:

This test case is unique because there is a diagonal win

Function:
testingCheckTie_notie

char whatsAtPos(BoardPosition pos)

Input state: pos = <0,0>

0	1	2	3

Output:

whatsAtPos(pos) = ' '

Board remains unchanged

Reason:

This test case is unique because the board is empty

Function:
testingWhatsAtPos_empty

char whatsAtPos(BoardPosition pos)

Input state: pos = <0,0>

0	1	2	3
		X	O
O	O	O	X

Output:

whatsAtPos(pos) = 'O'

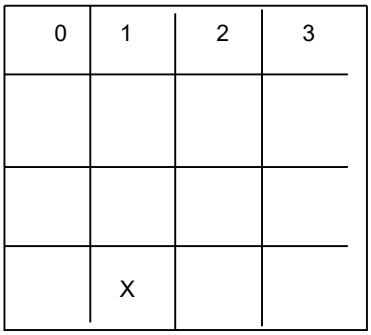
Board remains unchanged

Reason:

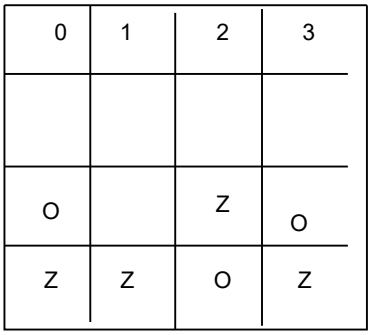
This test case is unique because there is a character in a unique position

Function:
testingCheckTie_char1

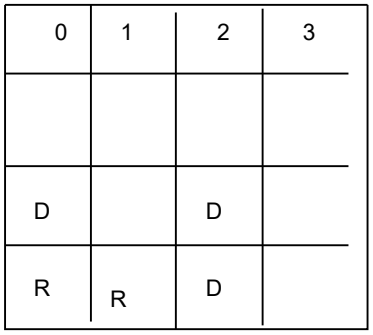
char whatsAtPos(BoardPosition pos)

Input state: pos = <0,1>	Output:	Reason:
	whatsAtPos(pos) = 'X' Board remains unchanged	This test case is unique because there is a character in a unique position Function: testingCheckTie_charX

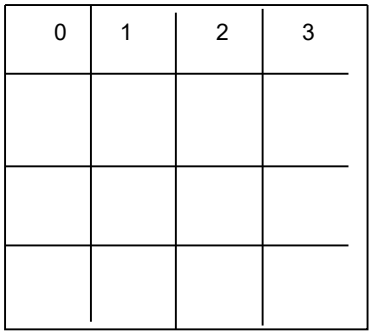
char whatsAtPos(BoardPosition pos)

Input state: pos = <0,0>	Output:	Reason:
	whatsAtPos(pos) = 'Z' Board remains unchanged	This test case is unique because there is a unique character (something other than X/O) in a unique position Function: testingCheckTie_charUnique

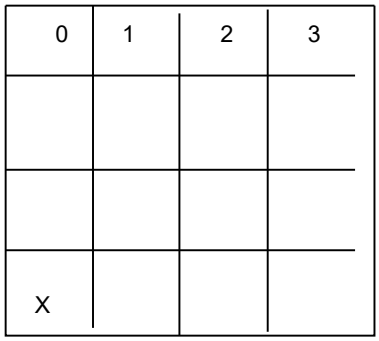
char whatsAtPos(BoardPosition pos)

Input state: pos = <1,2>	Output:	Reason:
	whatsAtPos(pos) = 'D' Board remains unchanged	This test case is unique because it makes sure that the function can find something not on the bottom row Function: testingWhatsAtPos_higherRow

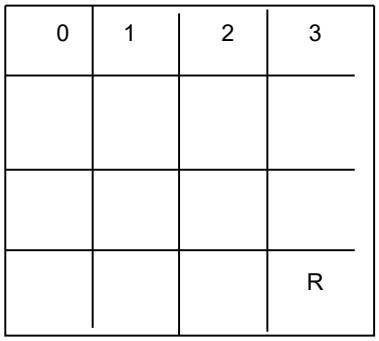
boolean isPlayerAtPos(BoardPosition pos, char p)

Input state: pos = <1,2>, p = 'X'	Output:	Reason:
	isPlayerAtPos(pos, p) = false Board remains unchanged	This test case is unique because the board is empty Function: testingIsPlayerAtPos_empty

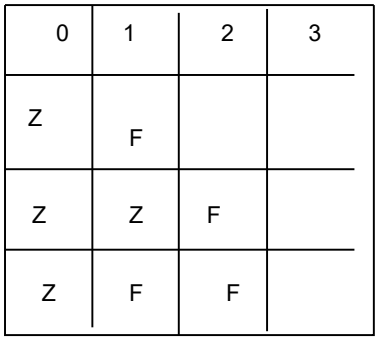
boolean isPlayerAtPos(BoardPosition pos, char p)

Input state: pos = <0,0>, p = 'X'	Output:	Reason:
	isPlayerAtPos(pos, p) = true Board remains unchanged	This test case is unique because the character is at the input position Function: testingIsPlayerAtPos_foundX

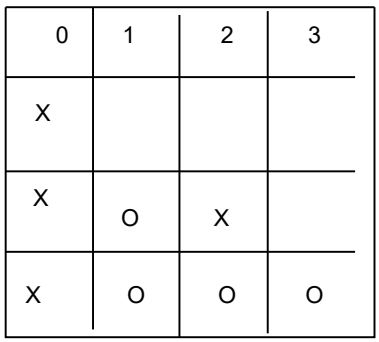
boolean isPlayerAtPos(BoardPosition pos, char p)

Input state: pos = <1,2>, p = 'R'	Output:	Reason:
	isPlayerAtPos(pos, p) = false Board remains unchanged	This test case is unique because the correct token is on the board, but not in the input position Function: testingIsPlayerAtPos_diffPos

boolean isPlayerAtPos(BoardPosition pos, char p)

Input state: pos = <0,1>, p = 'F'	Output:	Reason:
	isPlayerAtPos(pos, p) = true Board remains unchanged	This test case is unique because the correct character is found Function: testingIsPlayerAtPos_uniqueChar

boolean isPlayerAtPos(BoardPosition pos, char p)

Input state: pos = <2,1>, p = 'O'	Output:	Reason:
	isPlayerAtPos(pos, p) = true Board remains unchanged	This test case is unique because the correct character is found and is not located on a border Function: testingIsPlayerAtPos_border

void placeToken(char p, int c)

Input state:

0	1	2	3

pos = <0,0>, player = X

Output state:

0	1	2	3
X			

Reason:

This test case is unique because a character is being placed in an empty column

Function:
testingPlaceToken_empty

void placeToken(char p, int c)

Input state:

0	1	2	3
O	X	O	
O	X	X	

pos = <0,3>, player = O

Output state:

0	1	2	3
O	X	O	
O	X	X	O

Reason:

This test case is unique because a character is being placed in an empty column on a board with other tokens

Function:
testingPlaceToken_modPlace

void placeToken(char p, int c)

Input state:

0	1	2	3
X			

pos = <1,0>, player = O

Output state:

0	1	2	3
O			
X			

Reason:

This test case is unique because a character is being placed on top of something else

Function:
testingPlaceToken_top

void placeToken(char p, int c)

Input state:

0	1	2	3
O			
X			

pos = <2,0>, player = X

Output state:

0	1	2	3
X			
O			
X			

Reason:

This test case is unique because placing the token will make the column full

Function:
testingPlaceToken_fill

void placeToken(char p, int c)

Input state:

0	1	2	3
X	O	X	
O	X	O	X
X	O	X	O

pos = <2,3>, player = O

Output state:

0	1	2	3
X	O	X	O
O	X	O	X
X	O	X	O

Reason:

This test case is unique because a character is the last token to be placed

Function: testingPlaceToken_completeBoard

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,2>, p = 'X'

0	1	2	3
	X	O	
	O	X	
X	O	O	X

State: board.checkNumToWin() = 3;

Output:

checkDiagWin(pos, p) == true

Board remains unchanged

Reason:

This test case is unique because there is a winner

Function: testingCheckDiagWin_winleft

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,0>, p = 'X'

0	1	2	3
X			
O	X		
O	X	X	O

State: board.checkNumToWin() = 3;

Output:

checkDiagWin(pos, p) == true

Board remains unchanged

Reason:

This test case is unique because there is a winner

Function: testingCheckDiagWin_winleft2

boolean checkDiagWin(BoardPosition pos, char p)

Input state: pos = <0,2>, p = 'X'

0	1	2	3
R		O	
		O	R
R	O	O	R

State: board.checkNumToWin() = 3;

Output:

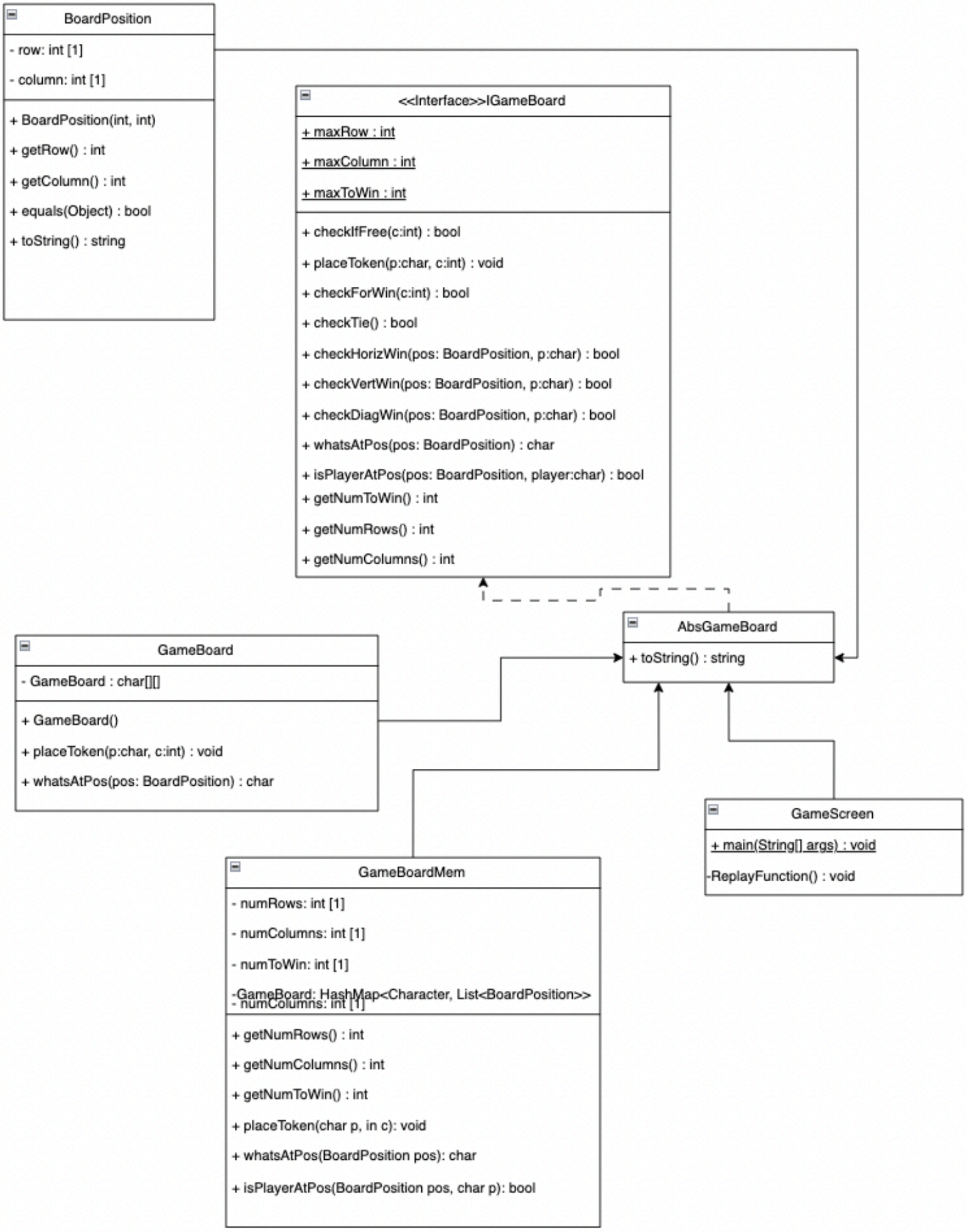
checkDiagWin(pos, p) == false

Board remains unchanged

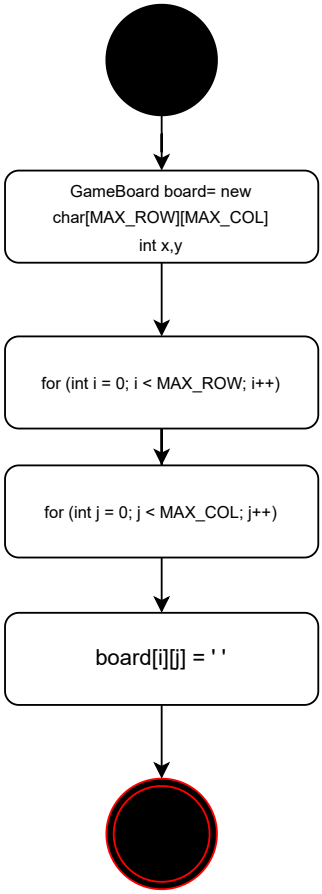
Reason:

This test case is unique because there is no diagonally winner, but there is a win for a different condition

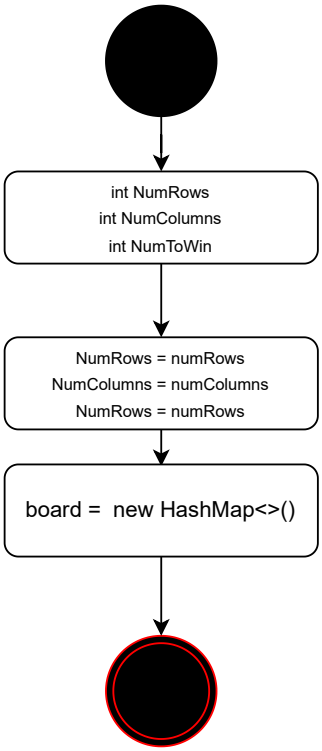
Function: testingCheckDiagWin_noWin2

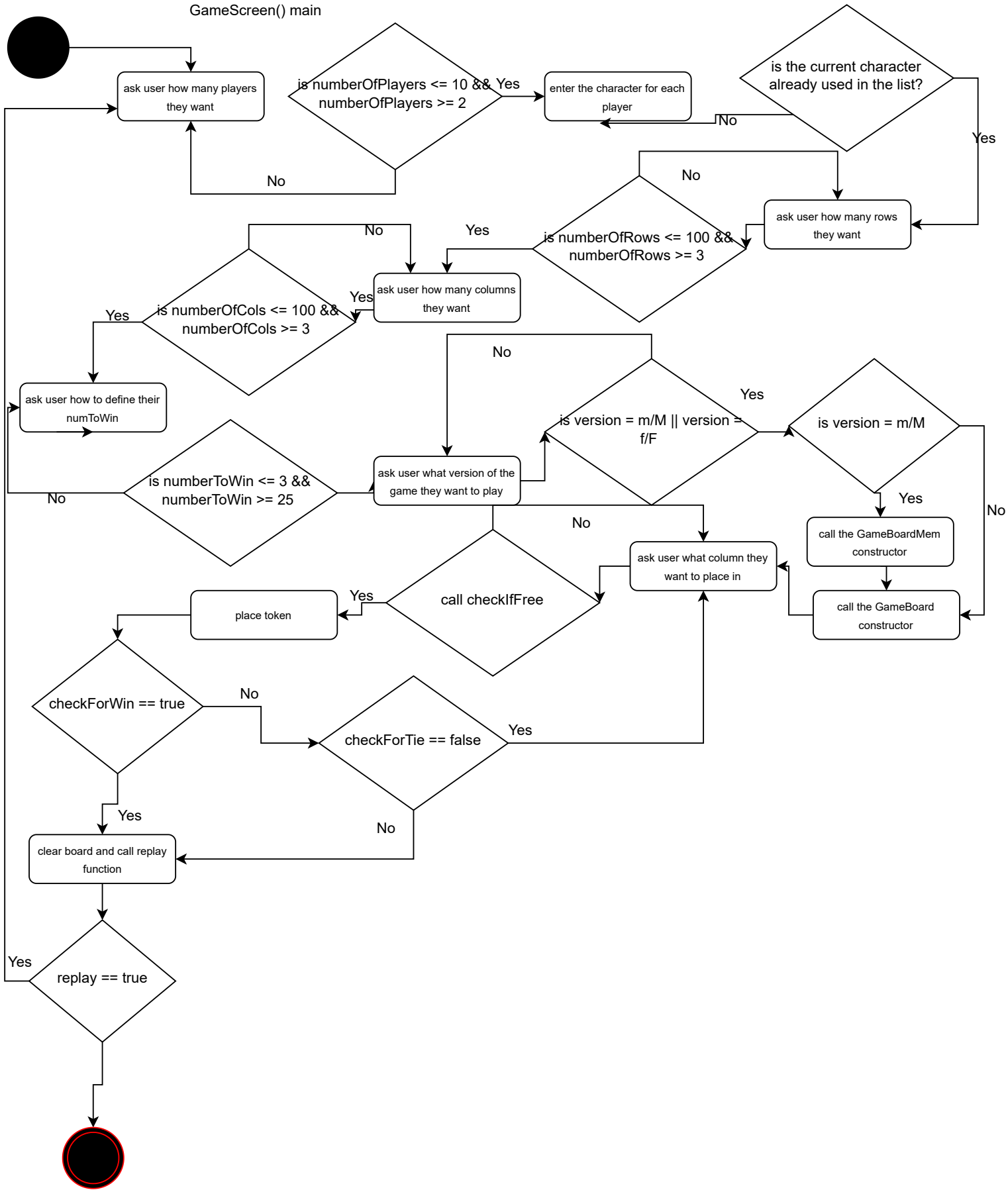

BoardPosition
- row: int [1]
- column: int [1]
+ BoardPosition(int, int)
+ getRow() : int
+ getColumn() : int
+ equals(Object) : bool
+ toString() : string
<<Interface>>IGameBoard
+ maxRow : int
+ maxColumn : int
+ maxToWin : int
+ checkIfFree(c:int) : bool
+ placeToken(p:char, c:int) : void
+ checkForWin(c:int) : bool
+ checkTie() : bool
+ checkHorizWin(pos: BoardPosition, p:char) : bool
+ checkVertWin(pos: BoardPosition, p:char) : bool
+ checkDiagWin(pos: BoardPosition, p:char) : bool
+ whatsAtPos(pos: BoardPosition) : char
+ isPlayerAtPos(pos: BoardPosition, player:char) : bool
+ getNumToWin() : int
+ getNumRows() : int
+ getNumColumns() : int
AbsGameBoard
+ toString() : string
GameBoard
- GameBoard : char[][]
+ GameBoard()
+ placeToken(p:char, c:int) : void
+ whatsAtPos(pos: BoardPosition) : char
GameScreen
+ main(String[] args) : void
-ReplayFunction() : void
GameBoardMem
- numRows: int [1]
- numColumns: int [1]
- numToWin: int [1]
-GameBoard: HashMap<Character, List<BoardPosition>>
- numColumns: int [1]
+ getNumRows() : int
+ getNumColumns() : int
+ getNumToWin() : int
+ placeToken(char p, in c): void
+ whatsAtPos(BoardPosition pos): char
+ isPlayerAtPos(BoardPosition pos, char p): bool

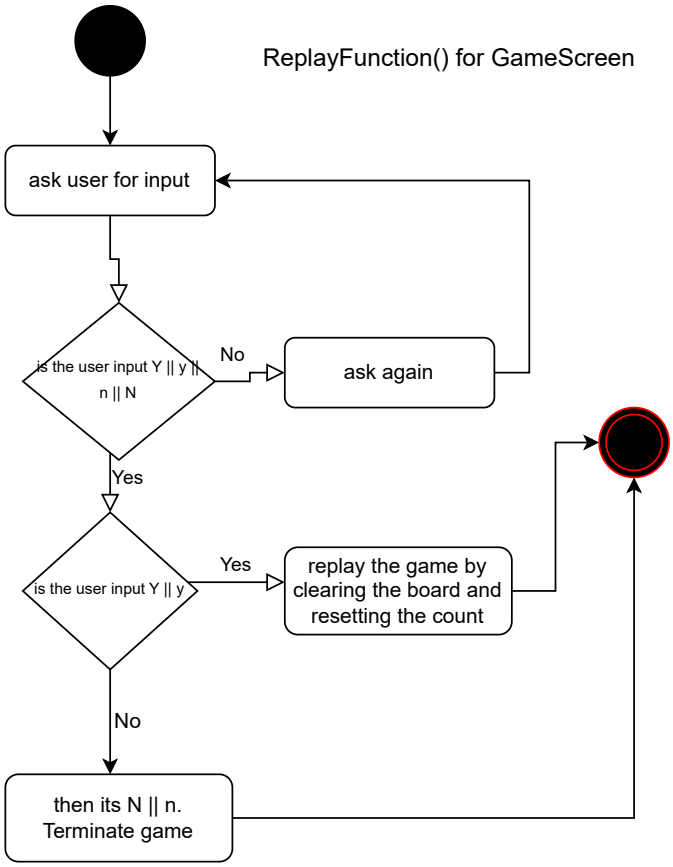
GameBoard() constructor



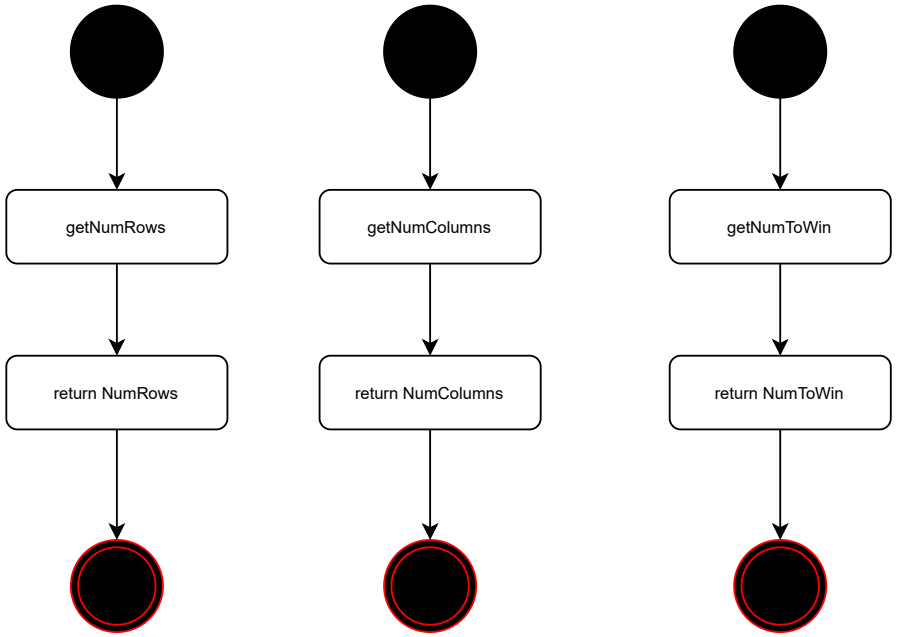
GameBoardMem() constructor



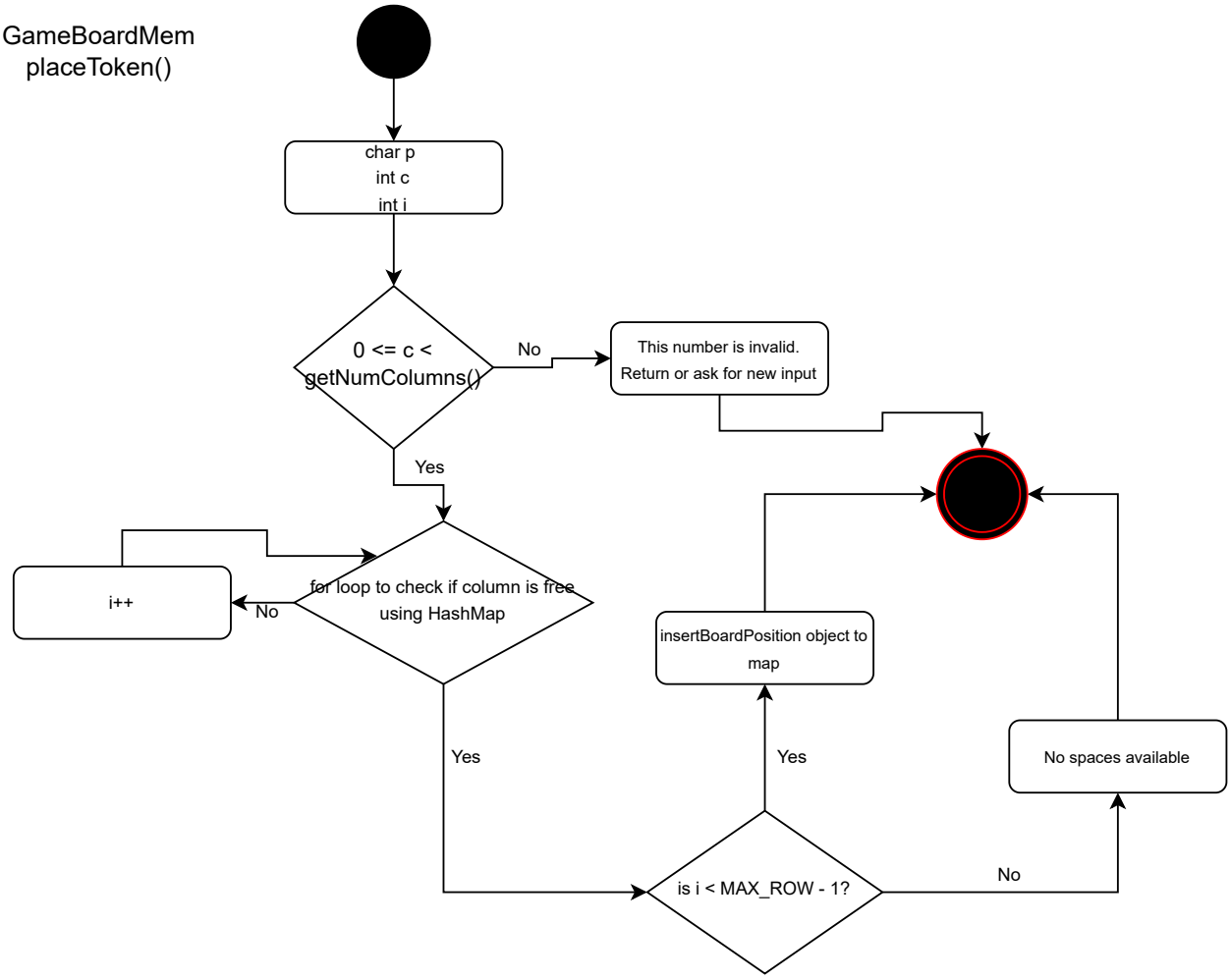

GameScreen() main
ask user how many players they want
is numberOfPlayers <= 10 && numberOfPlayers >= 2
Yes
No
enter the character for each player
is the current character already used in the list?
No
Yes
ask user how many rows they want
is numberOfRows <= 100 && numberOfRows >= 3
No
Yes
ask user how many columns they want
is numberOfCols <= 100 && numberOfCols >= 3
No
Yes
Yes
ask user how to define their numToWin
is numberToWin <= 3 && numberToWin >= 25
No
ask user what version of the game they want to play
No
is version = m/M || version = f/F
Yes
is version = m/M
Yes
No
call the GameBoardMem constructor
call the GameBoard constructor
ask user what column they want to place in
No
call checkIfFree
Yes
place token
checkForWin == true
No
checkForTie == false
Yes
Yes
No
clear board and call replay function
replay == true
Yes


ReplayFunction() for GameScreen
ask user for input
is the user input Y || y || n || N
No
ask again
Yes
is the user input Y || y
Yes
replay the game by clearing the board and resetting the count
No
then its N || n. Terminate game

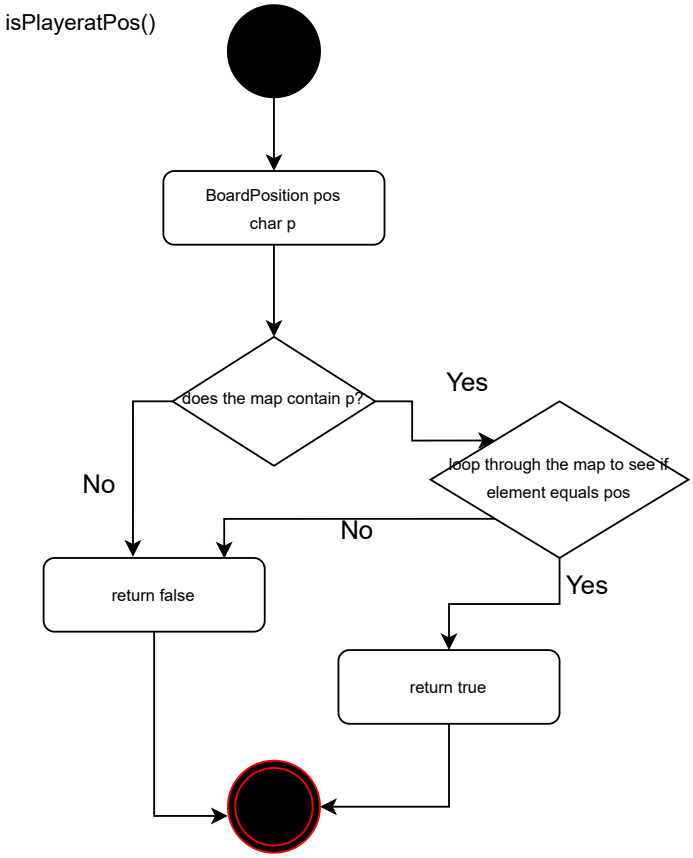
GameBoardMem
getNumRows()
getNumColumns
getNumToWin()

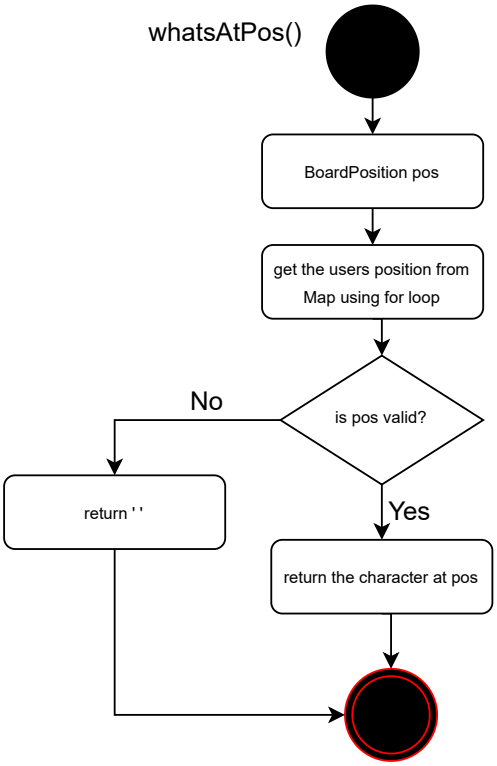


GameBoardMem
placeToken()



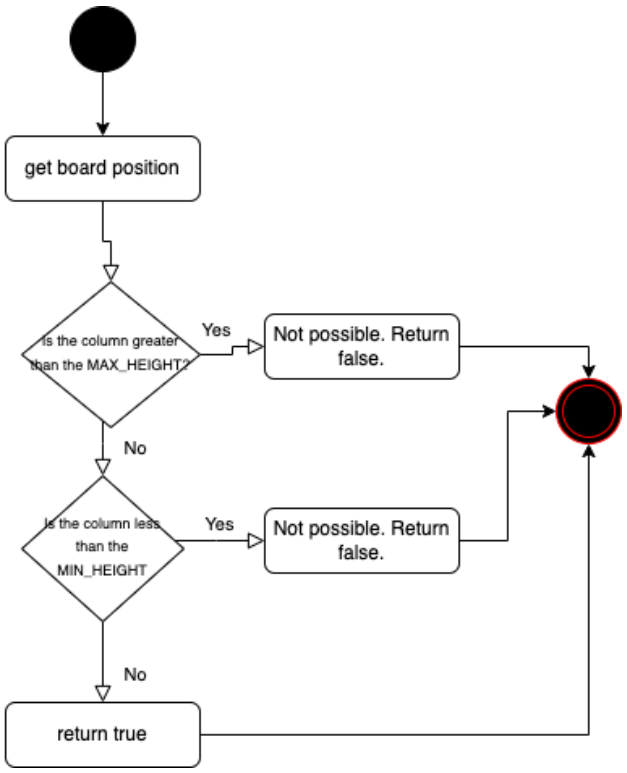
GameBoardMem



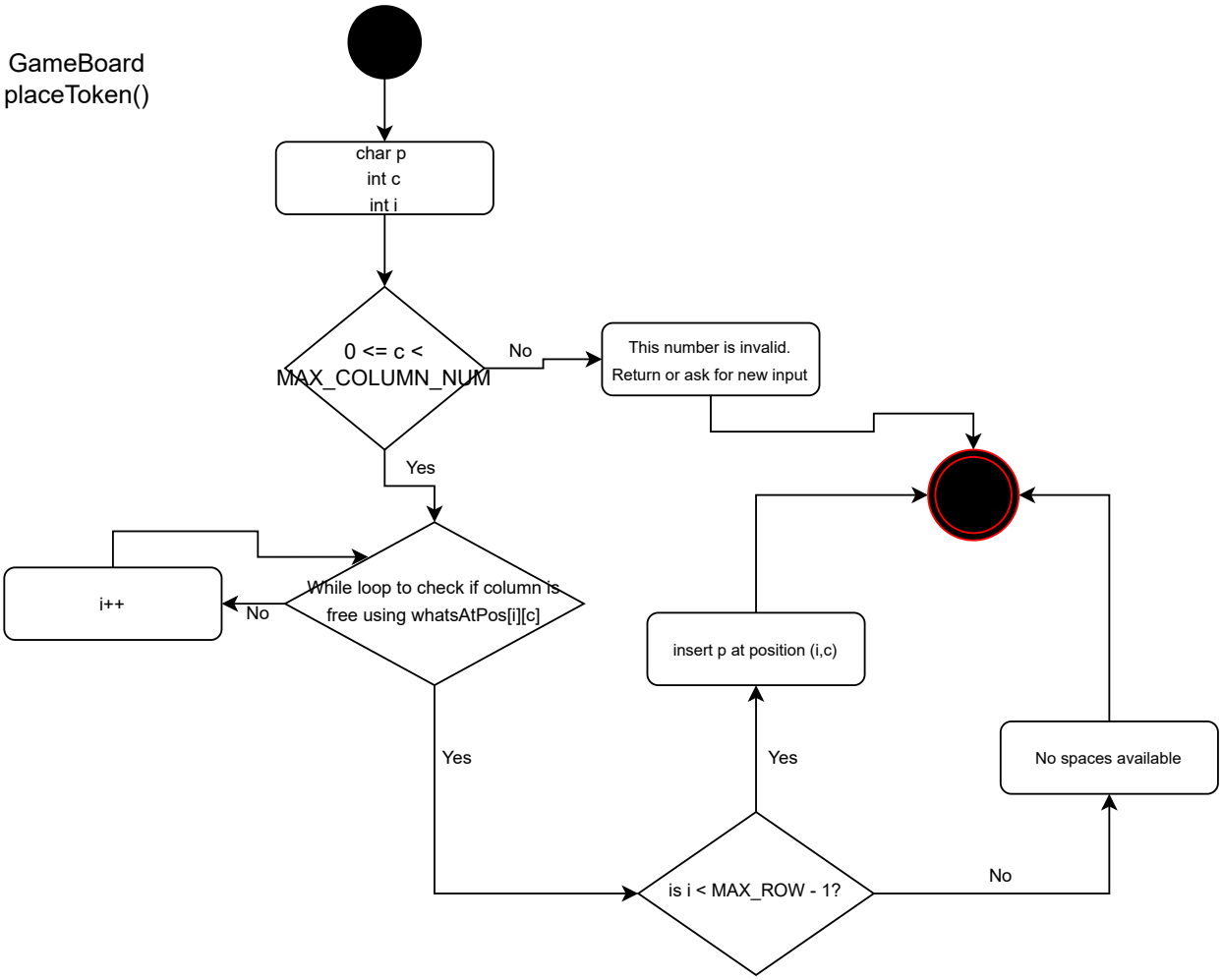


Default methods for IGameBoard:

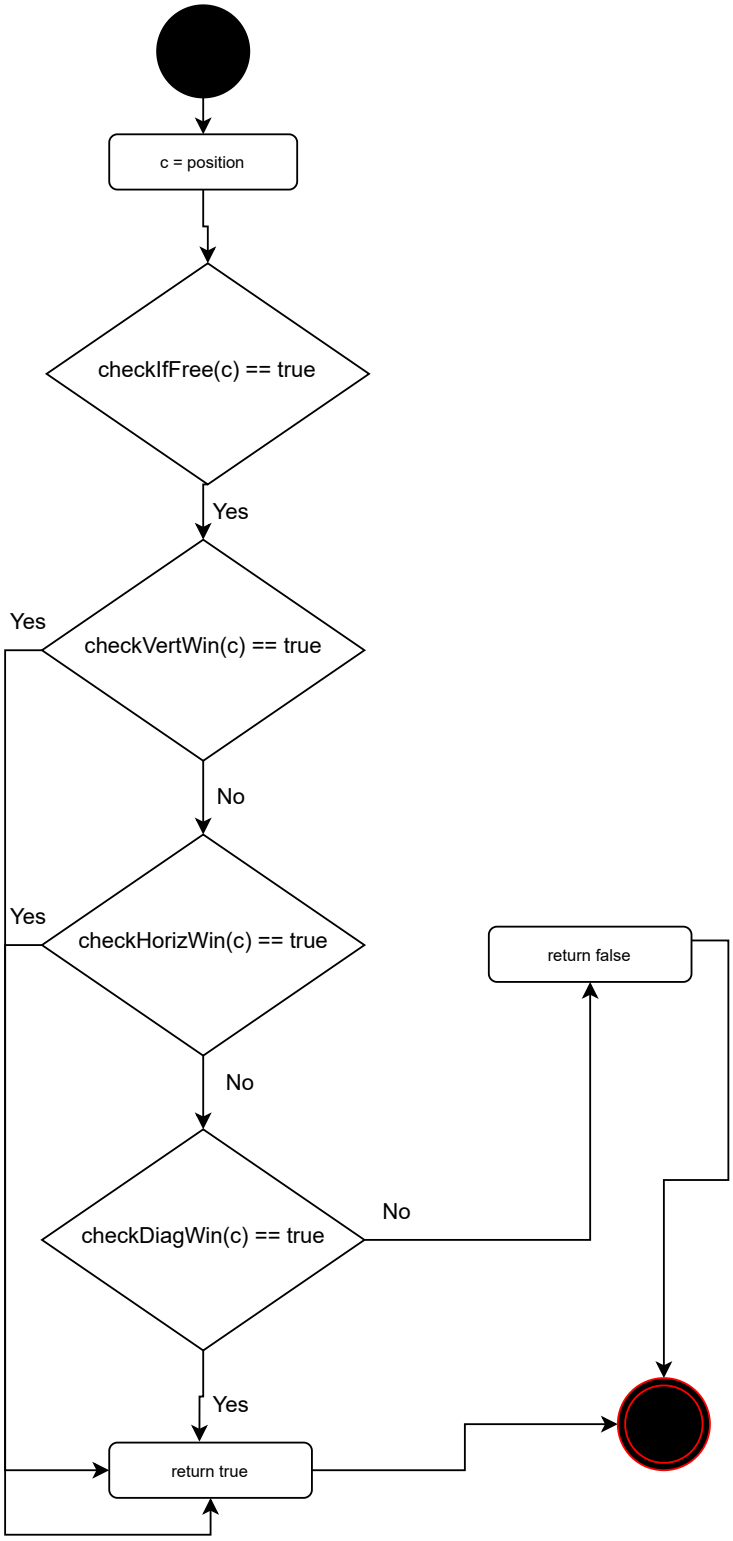
checkIfFree()

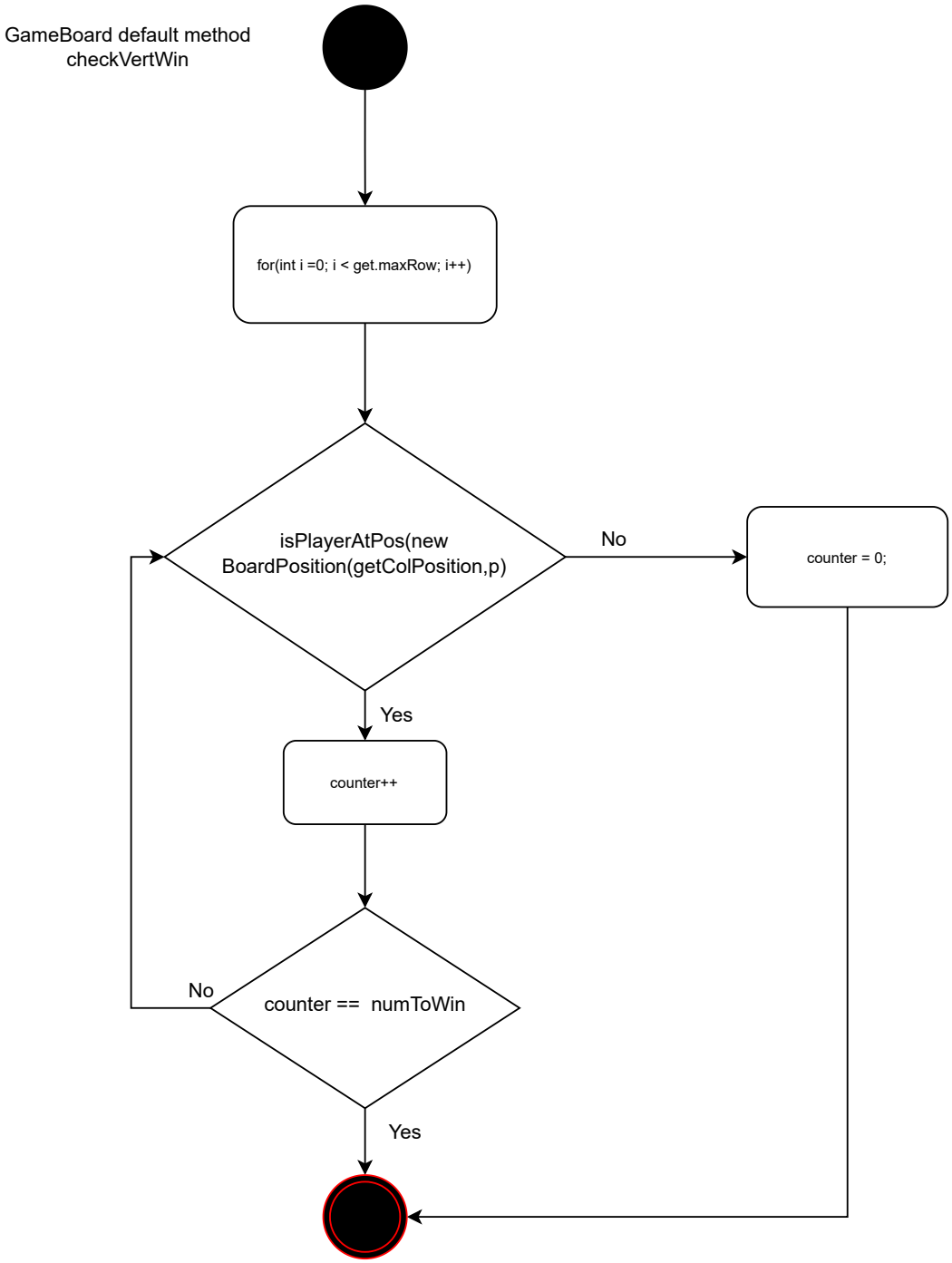


GameBoard
placeToken()

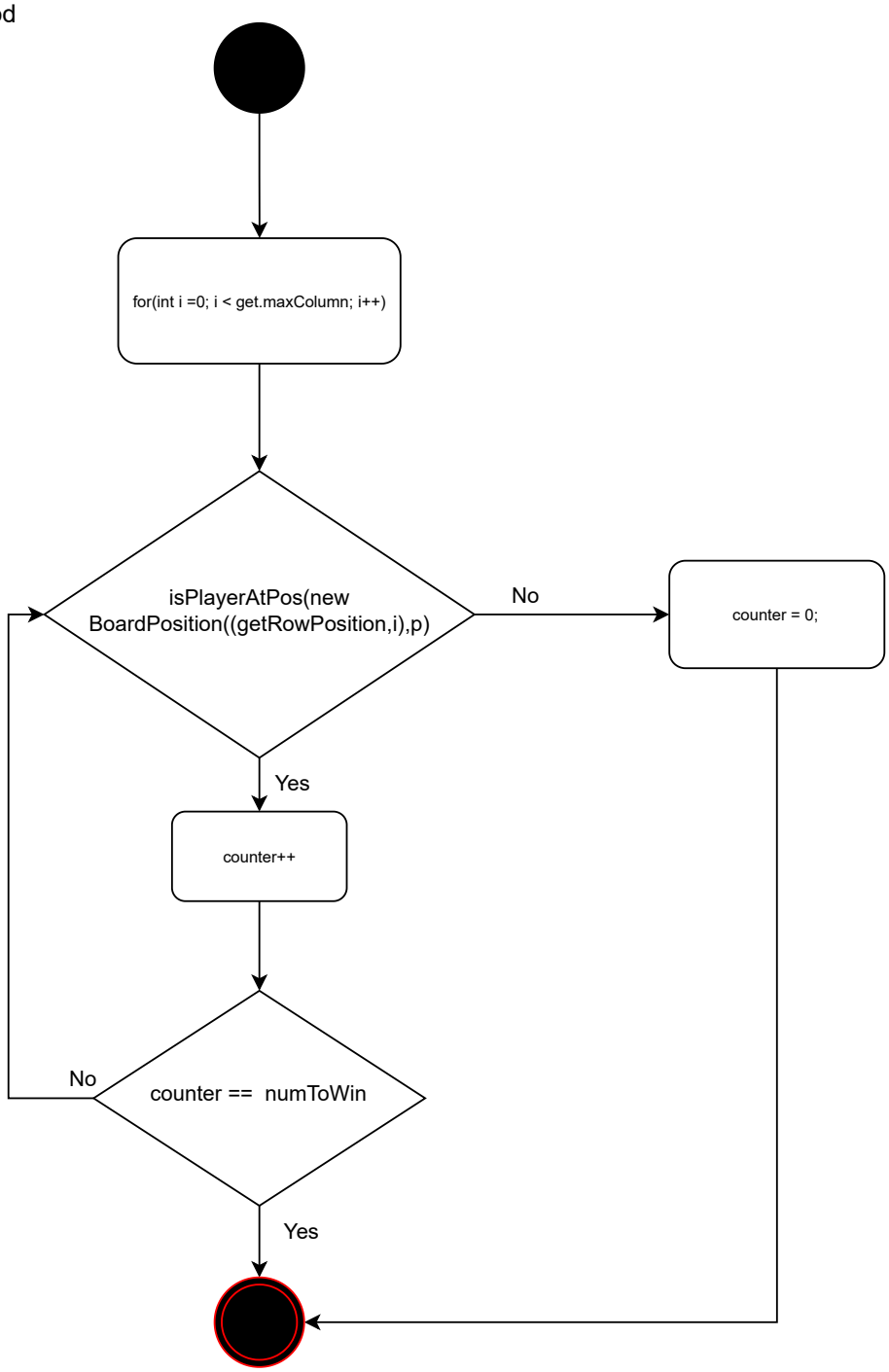


GameBoard default
checkForWin()

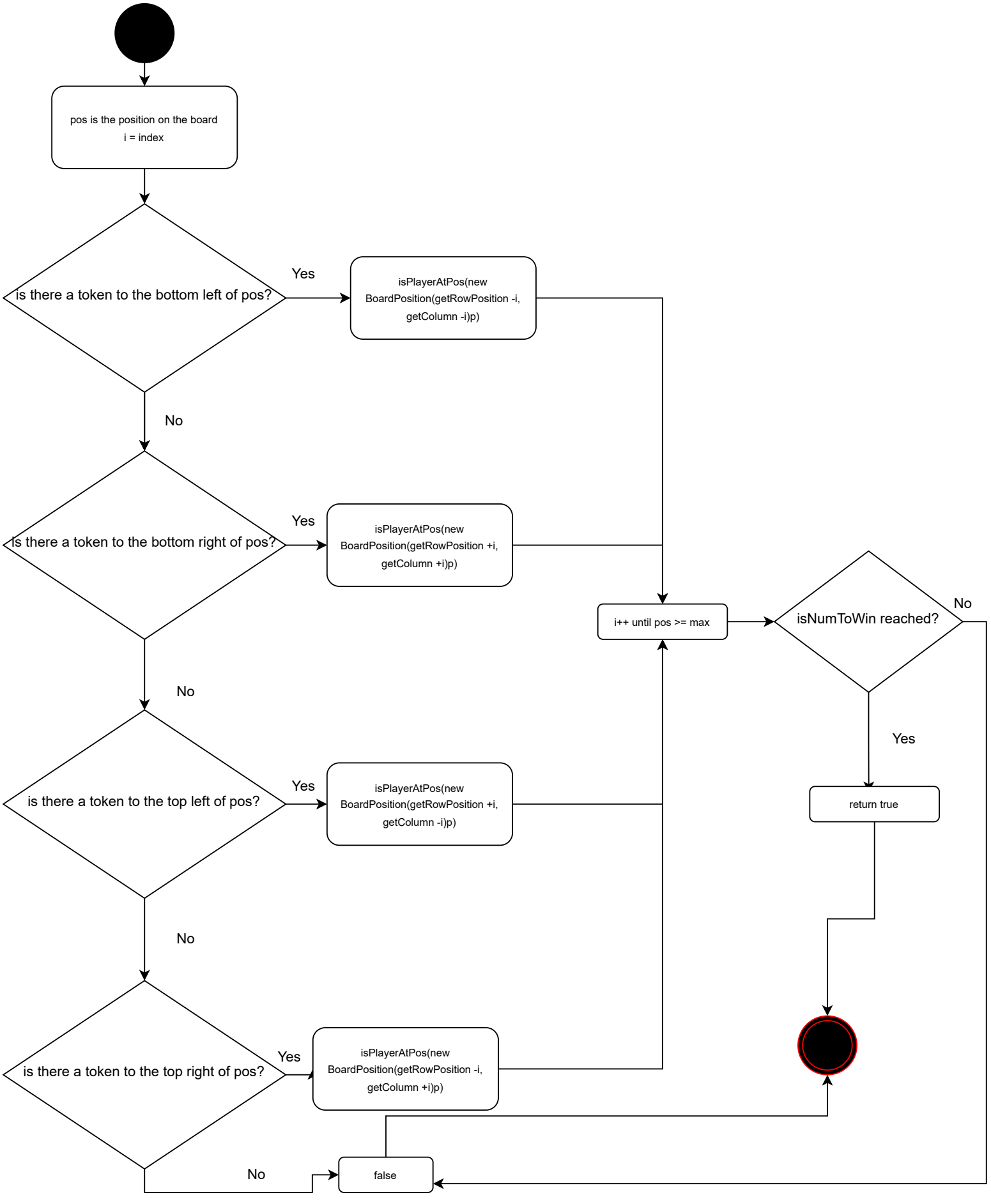



GameBoard default method
checkVertWin
for(int i =0; i < get.maxRow; i++)
isPlayerAtPos(new
BoardPosition(getColPosition,p)
No
counter = 0;
Yes
counter++
No
counter == numToWin
Yes

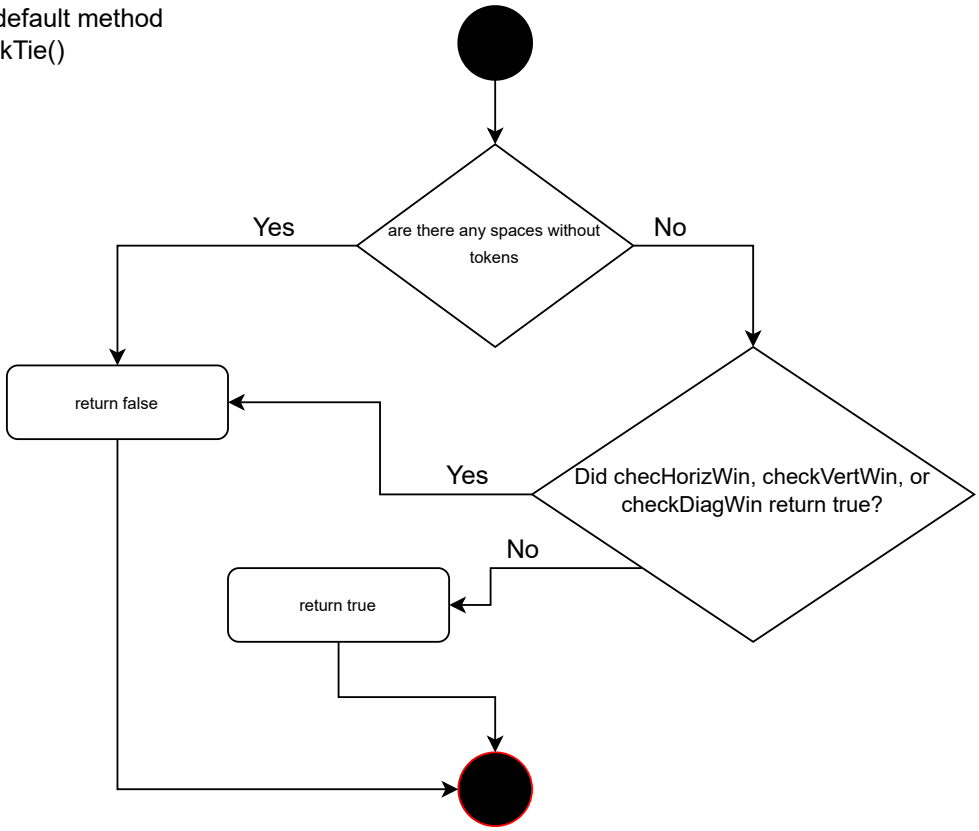
GameBoard default method
checkHorizWin()



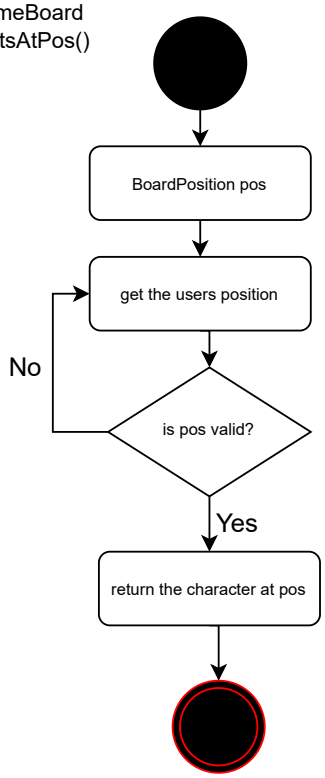
GameBoard default method
checkDiagWin



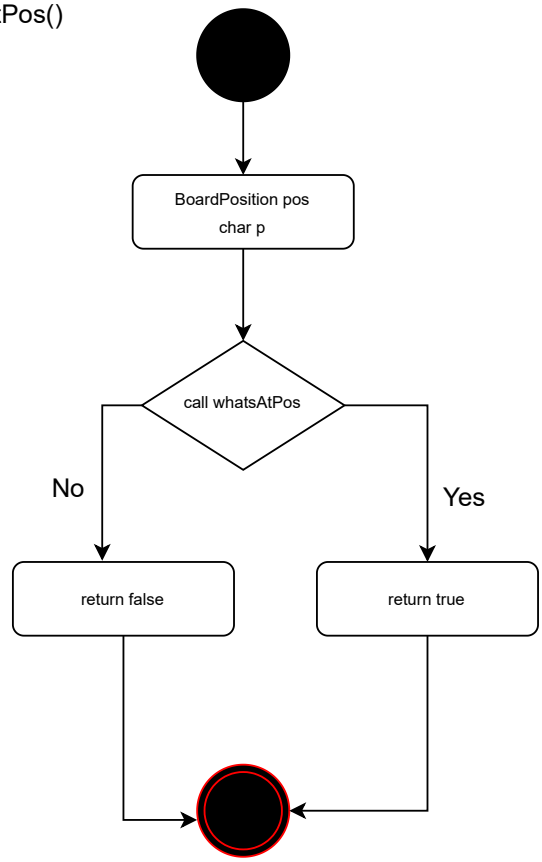
GameBoard default method
checkTie()



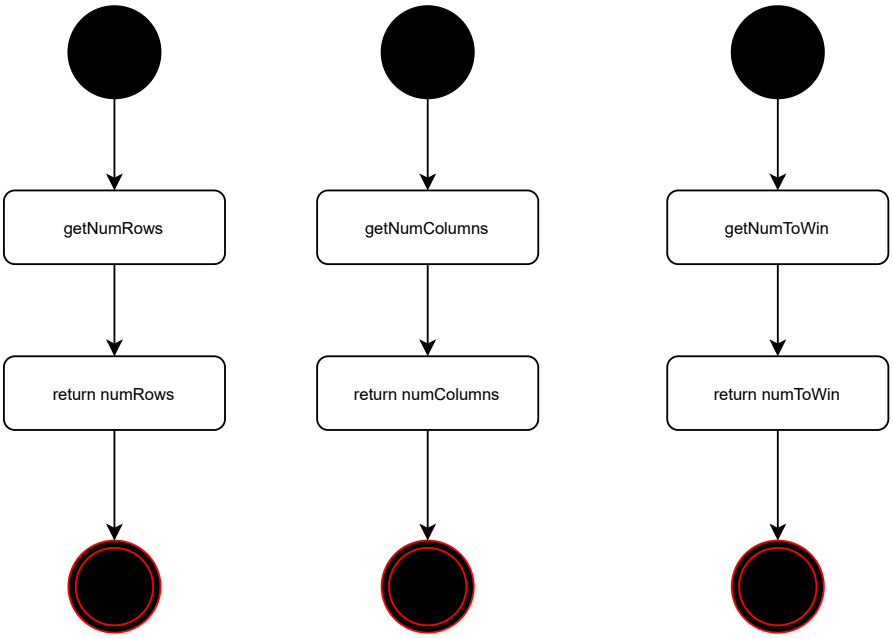
GameBoard
whatsAtPos()



GameBoard default method
isPlayerAtPos()



GameBoard
Getters



toString()

AbsGameBoard