Project 4: ConnectX

Author: Kevin Mody

Class: CPSC 2150/2151

User Story:

Functional Requirements:

1. As a user, I can have up to 10 players.
2. As a user, I can choose to have up to 100 rows and minimum of 3 rows
3. As a user, I can choose to have up to 100 columns and minimum of 3 columns.
4. As a user, I can start from any number of given columns.
5. As a user, my player token can only start from the bottom row.
6. As a user, my player token can be any character from the keyboard.
7. As a user, I can choose to have up to 25 in a row to win but it cannot be more than columns or rows.
8. As a user, I can try to stop other player winning by placing the token in the columns.
9. As a user, I can win if I can get the number of given place tokens in a row.
10. As a user, I can choose to have either fast game or memory efficient game.
11. As a user If I win, I get to decide to play again or exit the game.
12. As a user if the game ties, I have an option to play it again or close it.
13. As a user, if I opted in to play again than I should start as any number of player and any player character.
14. As a user, if I place a place token outside the bounds it will give me an error and it will ask me to choose it again.
15. As a user, if I place a place token other than mine it will give me an error and it will ask me to choose it again.
16. As a user, if I place a token at a filled space, it will give me an error and prompt me to rechoose it.
17. As a user, if I put a token in a column(s) that is/are already full, it will inform me an error and ask me to choose again.
18. As a user if I don’t have any free space than game will be tie.

Non-Functional Requirements:

1. The program is written in Java.
2. The program runs on Unix.
3. The program runs on IntelliJ.

4. The number of tokens to win are between 3 and 25 inclusive.
5. The number of rows is between 3 and 100 inclusive.
6. The number of columns is between 3 and 100 inclusive.
7. The number of players is between 2 and 10 inclusive.
8. Player 1 always goes first.
9. 0,0 is the bottom left of the board.

Deployment:

1. To compile my program, you can simply type make in the command line inside src.
2. Typing “make run” after running the make command will run my program
3. Typing “make test” will compile all of the test cases
4. Typing “make testGB” will run the 40 test cases for the GameBoard implementation
5. Typing “make testGBMem” will run the 40 test cases for the GameBoardMem implementation
6. After running my program, typing “make clean” will remove all of the .class files in the package

Test Cases for GameBoard/GameBoardMem

Gameboard(int r, int c, int w)

<table>
<tr>
<td>Input r = 3
c =3
w = 3</td>
<td>Output State:
<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>
<tr><td></td><td></td><td></td></tr>
</table>
board.getNumToWin() = 3</td>
<td>Reasoning:
This test case is distinct because it tests if the constructor can make the smallest possible board
Name: test_Constructor_smallest</td>
</tr>
</table>

Gameboard(int r, int c, int w)

Input
r = 100
c = 100
w = 25
	Output State:
	Board = 100 X 100
	board.getNumToWin() = 25
		Reasoning:
		This test case is distinct because it tests if the constructor can make the biggest possible board
		Name: test_Constructor_biggest

Gameboard(int r, int c, int w)

Input r = 30
c = 20
w = 3
	Output State:
	Board = 30 X 20
	board.getNumToWin() = 3
		Reasoning:
		This test case is distinct because it tests if the constructor can make a board with unequal rows and columns
		Name: test_Constructor_different

boolean checkIfFree(int c)

<table>
<tr>
<td>Input 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>
</table>
</td>
<td>Output checkIfFree(2) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct
because it represents a
standard test of a free column
Function name: test_CheckIfFree_empty</td>
</tr>
</table>

boolean checkIfFree(int c)

Input State:	Output checkIfFree(4) = true Board is unchanged	Reasoning: This test case is distinct because it represents a standard test of a column that has tokens in it Function name: test_CheckIfFree_one_space

boolean checkIfFree(int c)

<table>
<tr>
<td>Input 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>X</td></tr><tr><td></td><td></td><td></td><td></td><td>X</td></tr><tr><td></td><td></td><td></td><td></td><td>X</td></tr><tr><td></td><td></td><td></td><td></td><td>X</td></tr><tr><td></td><td></td><td></td><td></td><td>X</td></tr></table></td>
<td>Output checkIfFree(4) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a standard test of a column that isn’t free
Function name:
test_CheckIfFree_full</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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></table>
pos = (2, 2) p = ‘X’
board.checkNumToWin() = 3</td>
<td>Output
checkHorizWin(pos, p) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a standard test of there being no horizontal win
Function name:
test_CheckHorizontalWin_emp ty</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td>X</td><td>X</td><td></td><td></td></tr></table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td>
<td>Output
checkHorizWin(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a standard test of conditions being fulfilled for a horizontal win
Function name:
test_CheckHorizontalWin_just
_enough</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
</table>
board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output
checkHorizWin(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it tests to make sure more than the specified amount will result in a win
Function name: test_CheckHorizontalWin_more_than_enough</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td>X</td><td>O</td><td>X</td><td></td></tr>
</table>
board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output checkHorizWin(pos, p) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it tests to make sure the function checks the tokens in a row are the same
Function name: test_CheckHorizontalWin_just
_not_enough</td>
</tr>
</table>

boolean checkVertWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>
</table>
board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output
checkVertWin(pos, p) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a standard case of there being no vertical win
Function name:
test_CheckVerticalWin_empty</td>
</tr>
</table>

boolean checkVertWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</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>
board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output
checkVertWin(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a standard case of there being a vertical win
Function name:
test_CheckVerticalWin_just_enough</td>
</tr>
</table>

boolean checkVertWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</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>

board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output
checkVertWin(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a case where there is more than enough for a vertical win
Function name: test_CheckVerticalWin_more_than_enough</td>
</tr>
</table>

boolean checkVertWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>

board.getNumToWin() = 3
Pos = (2,2) P = ‘X’</td>
<td>Output
checkVertWin(pos, p) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents a case where the function checks to make sure the tokens in a row are the same
Function name: test_CheckVerticalWin_just_not_enough</td>
</tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><td>Input 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></table>
board.getNumToWin() = 3
Pos = (0,0) P = ‘X’</td><td>Output
checkDiagWin(pos, p) = false
Board is unchanged</td><td>Reasoning:
This test case is distinct because it represents a standard case of there being no diagonal win
Function name: test_CheckDiagonalWin_empt y</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><td>Input State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>X</td><td></td></tr><tr><td>O</td><td>X</td><td>X</td><td></td></tr><tr><td>X</td><td>O</td><td>O</td><td></td></tr></table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td><td>Output
checkDiagWin(pos, p) = true
Board state is unchanged</td><td>Reasoning:
This test case is distinct because it represent the standard case of there being a diagonal win that starts from the left
Function name: test_CheckDiagonalWin_left_just_enough</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><td>Input State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td></td><td></td><td></td><td>X</td></tr><tr><td></td><td></td><td>X</td><td>X</td></tr><tr><td>O</td><td>X</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>O</td><td>X</td></tr></table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td><td>Output
checkDiagWin(pos, p) = true
Board state is unchanged</td><td>Reasoning:
This test case is distinct because it represents the case where there is more than enough for a diagonal win that starts to the left
Function name:
test_CheckDiagonalWin_left_more_than_enough</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><td>Input State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td></td><td></td><td></td><td>O</td></tr><tr><td></td><td></td><td>O</td><td>X</td></tr><tr><td>O</td><td>X</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>O</td><td>X</td></tr></table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td><td>Output
checkDiagWin(pos, p) = false
Board state is unchanged</td><td>Reasoning:
This test case is distinct because it represents the case where the function checks that the tokens in a row are the same, for a diagonal that starts at the left
Function name:
test_CheckDiagonalWin_left_just_not_enough</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input State:
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td>X</td><td></td><td></td></tr><tr><td></td><td>O</td><td>X</td><td></td></tr><tr><td></td><td>O</td><td>O</td><td>X</td></tr></table>board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td>
<td>Output
checkDiagWin(pos, p) = true
Board state is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the basic case where there is a diagonal win that starts to the right
Function name:
test_CheckDiagonalWin_right_just_enough</td>
</tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input State:
<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>X</td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td></td><td></td></tr>
<tr><td>O</td><td>O</td><td>X</td><td></td></tr>
<tr><td>X</td><td>O</td><td>O</td><td>X</td></tr>
</table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td>
<td>Output

checkDiagWin(pos, p) = true
Board state is unchanged</td>
<td>Reasoning:

This test case is distinct because it represents the case where there is a diagonal win that starts to the right when there is more than enough tokens in a row

Function name:
test_CheckDiagonalWin_right_more_than_enough</td>
</tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr>
<td>Input State:
<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>O</td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td></td><td></td></tr>
<tr><td>O</td><td>O</td><td>X</td><td></td></tr>
<tr><td>X</td><td>O</td><td>O</td><td>X</td></tr>
</table>
board.getNumToWin() = 3
Pos = (2, 2) P = ‘X’</td>
<td>Output

checkDiagWin(pos, p) = false
Board state is unchanged</td>
<td>Reasoning:

This test case is distinct because it represents the case where the function checks to make sure a diagonal starting from the right has equivalent tokens in a row

Function name:
test_CheckDiagonalWin_right_just_not_enough</td>
</tr>
</table>

boolean CheckTie()

<table>
<tr>
<td>Input State:
<table><tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></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></td>
<td>Output CheckTie() = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the standard case of there being no tie
Function name: test_CheckTie_empty</td>
</tr>
</table>

boolean CheckTie()

<table>
<tr>
<td>Input State:
<table><tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><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></tr><tr><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></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr></table></td>
<td>Output CheckTie() = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the standard case of there being a tie
Function name: test_CheckTie_full</td>
</tr>
</table>

boolean CheckTie()

<table>
<tr>
<td>Input State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><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></td></tr>
<tr><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></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
</table>
</td>
<td>Output CheckTie() = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it ensures that the function checks every column
Function name: test_CheckTie_almost_full</td>
</tr>
</table>

boolean CheckTie()

<table>
<tr>
<td>Input State:
<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><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></tr>
<tr><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></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
</table>
</td>
<td>Output CheckTie() = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it ensures that the function checks every space on the board
Function name: test_CheckTie_all_but_one</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td>Input 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>
</table>
Pos = (0,0)</td>
<td>Output whatsAtPos(pos) = ‘ ‘
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the standard case of no character being on the position
Function name:
test_WhatsAtPos_empty</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td>Input 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>X</td><td></td><td></td><td></td><td></td></tr>
</table>
Pos = (0,0)</td>
<td>Output whatsAtPos(pos) = ‘X ‘
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the standard case of a character being on the position
Function name:
test_WhatsAtPos_player_x</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td>Input 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>K</td><td></td><td></td><td></td><td></td></tr>
</table>
Pos = (0,0)</td>
<td>Output whatsAtPos(pos) = ‘E‘

Board is unchanged</td>
<td>Reasoning:

This test case is distinct because it ensures the function recognizes characters that aren’t X or O

Function name:
test_WhatsAtPos_player_K</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td>Input 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>X</td><td></td><td></td><td></td><td></td></tr>
</table>
Pos = (0,1)</td>
<td>Output whatsAtPos(pos) = ‘ ‘

Board is unchanged</td>
<td>Reasoning:

This test case is distinct because it ensures the function is checking the correct position

Function name:
test_WhatsAtPos_player_near by</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr>
<td>Input 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>X</td><td>O</td><td></td><td></td><td></td></tr>
</table>

Pos = (0,1)</td>
<td>Output whatsAtPos(pos) = ‘O‘

Board is unchanged</td>
<td>Reasoning:

This test case is distinct because it ensures the function will return the correct character

Function name:
test_WhatsAtPos_two_players</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>
</table>

Pos = (0,0) P = ‘X’</td>
<td>Output isPlayerAtPos = false

Board is unchanged</td>
<td>Reasoning:

This test case is distinct because it represents the standard case of no character being on the position

Function name:
test_IsPlayerAtPos_empty</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td></td><td></td><td></td><td></td></tr></table>
Pos = (0,0) P = ‘X’</td>
<td>Output
isPlayerAtPos(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it represents the standard case of a character being on the position
Function name: test_IsPlayerAtPos_player_x</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>K</td><td></td><td></td><td></td><td></td></tr></table>
Pos = (0,0) P = ‘E’</td>
<td>Output
isPlayerAtPos(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it ensures the function recognizes characters that aren’t X and O
Function name: test_IsPlayerAtPos_player_K</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td></td><td></td><td></td><td></td></tr>
</table>

Pos = (0,1) P = ‘X’</td>
<td>Output
isPlayerAtPos(pos, p) = false
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it ensures that the function checks the correct position
Function name:
test_IsPlayerAtPos_empty_space_nearby</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr>
<td>Input 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>X</td><td>O</td><td></td><td></td><td></td></tr>
</table>

Pos = (0,1) P = ‘O’</td>
<td>Output
isPlayerAtPos(pos, p) = true
Board is unchanged</td>
<td>Reasoning:
This test case is distinct because it ensures the function is looking for the correct character
Function name:
test_IsPlayerAtPos_two_playe rs</td>
</tr>
</table>

Void placeToken(char p, int c)

Input

0	1	2	3	4

Output

0	1	2	3	4
X				

Reasoning:

This test case is distinct because it is testing if the function can add a token to the first available column

Function name:
test_PlaceToken_bottom_left

Void placeToken(char p, int c)

Input

0	1	2	3	4

Output

0	1	2	3	4
				X

Reasoning:

This test case is distinct because it is testing if the function can add a token to the last available column

Function name:
test_PlaceToken_bottom_right

Void placeToken(char p, int c)

Input

0	1	2	3	4

Output

0	1	2	3	4
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X

Reasoning:

This test case is distinct because it is testing if the function can add many tokens over and over again

Function name:
test_PlaceToken_fill_board

Void placeToken(char p, int c)

Input

0	1	2	3	4

Output

0	1	2	3	4
X	O			

Reasoning:

This test case is distinct because it is testing if the function can add different tokens

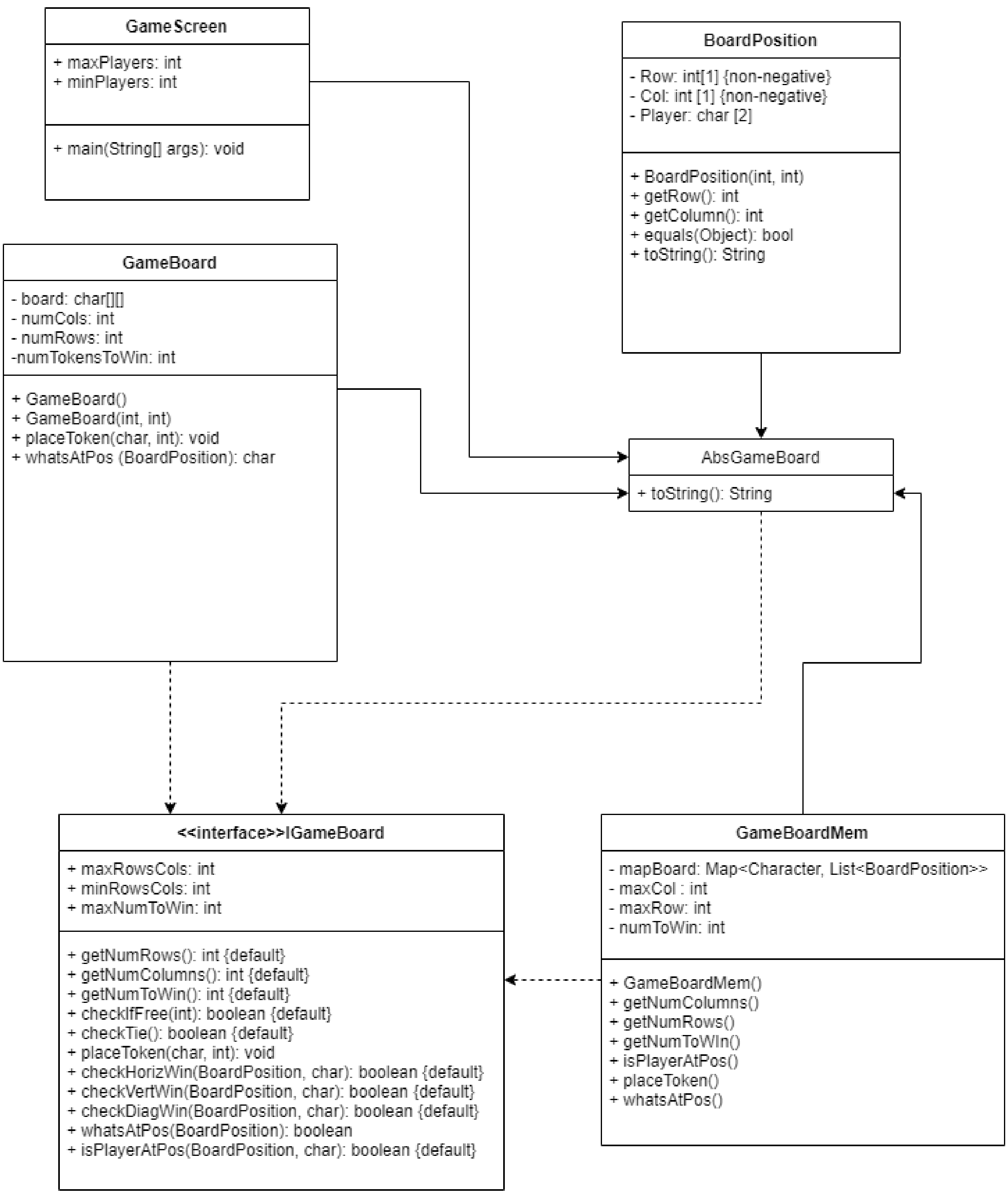
Function name:
test_PlaceToken_different_characters

Void placeToken(char p, int c)

<table>
<tr>
<td>Input
<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></table></td>
<td>Output
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>S</td><td>H</td><td>R</td><td>E</td><td>K</td></tr><tr><td>S</td><td>H</td><td>R</td><td>E</td><td>K</td></tr><tr><td>S</td><td>H</td><td>R</td><td>E</td><td>K</td></tr><tr><td>S</td><td>H</td><td>R</td><td>E</td><td>K</td></tr><tr><td>S</td><td>H</td><td>R</td><td>E</td><td>K</td></tr></table></td>
<td>Reasoning:
This test case is distinct because it is testing if the function can add characters in a pattern
Function name: test_PlaceToken_fill_different_characters</td>
</tr>
</table>

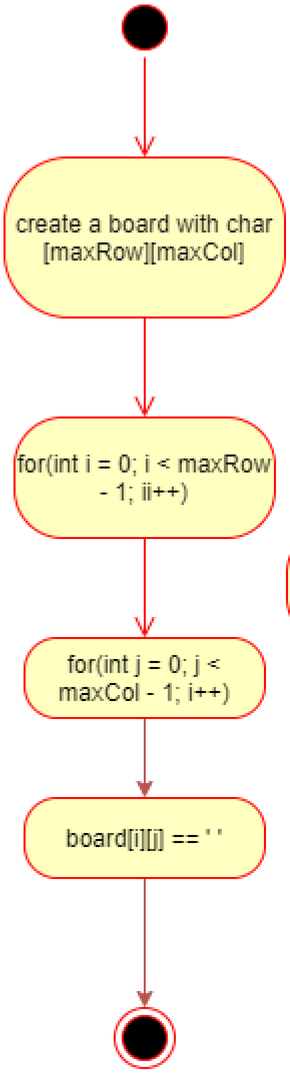
UML Class Diagrams

ConnectX - UML Class Diagrams

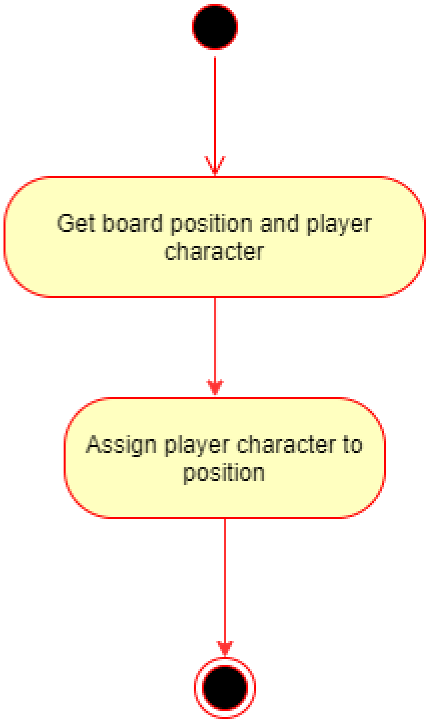

GameScreen
+ maxPlayers: int
+ minPlayers: int
+ main(String[] args): void
BoardPosition
- Row: int[1] {non-negative}
- Col: int [1] {non-negative}
- Player: char [2]
+ BoardPosition(int, int)
+ getRow(): int
+ getColumn(): int
+ equals(Object): bool
+ toString(): String
GameBoard
- board: char[][]
- numCols: int
- numRows: int
-numTokensToWin: int
+ GameBoard()
+ GameBoard(int, int)
+ placeToken(char, int): void
+ whatsAtPos (BoardPosition): char
AbsGameBoard
+ toString(): String
<<interface>>IGameBoard
+ maxRowsCols: int
+ minRowsCols: int
+ maxNumToWin: int
+ getNumRows(): int {default}
+ getNumColumns(): int {default}
+ getNumToWin(): int {default}
+ checkIfFree(int): boolean {default}
+ checkTie(): boolean {default}
+ placeToken(char, int): void
+ checkHorizWin(BoardPosition, char): boolean {default}
+ checkVertWin(BoardPosition, char): boolean {default}
+ checkDiagWin(BoardPosition, char): boolean {default}
+ whatsAtPos(BoardPosition): boolean
+ isPlayerAtPos(BoardPosition, char): boolean {default}
GameBoardMem
- mapBoard: Map<Character, List<BoardPosition>>
- maxCol : int
- maxRow: int
- numToWin: int
+ GameBoardMem()
+ getNumColumns()
+ getNumRows()
+ getNumToWIn()
+ isPlayerAtPos()
+ placeToken()
+ whatsAtPos()

UML Activity Diagrams

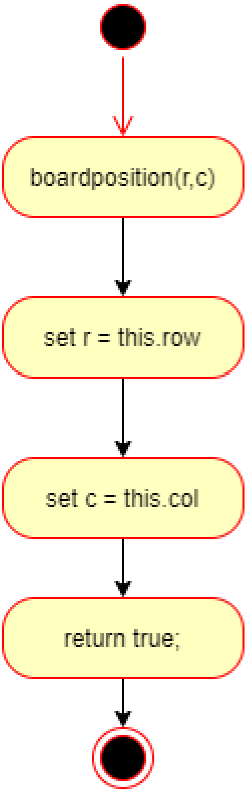
GameBoard constructor


create a board with char [maxRow][maxCol]
for(int i = 0; i < maxRow - 1; ii++)
for(int j = 0; j < maxCol - 1; i++)
board[i][j] == ' '

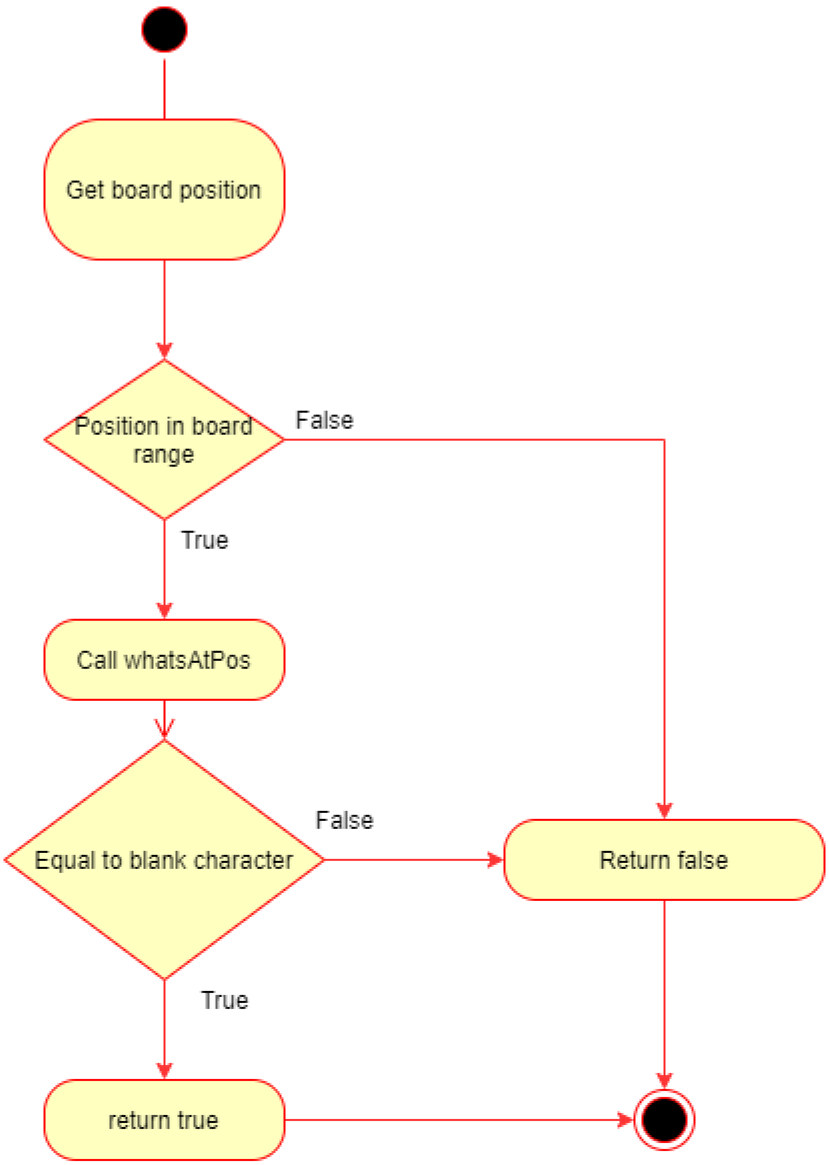
placeToken


Get board position and player character
Assign player character to position

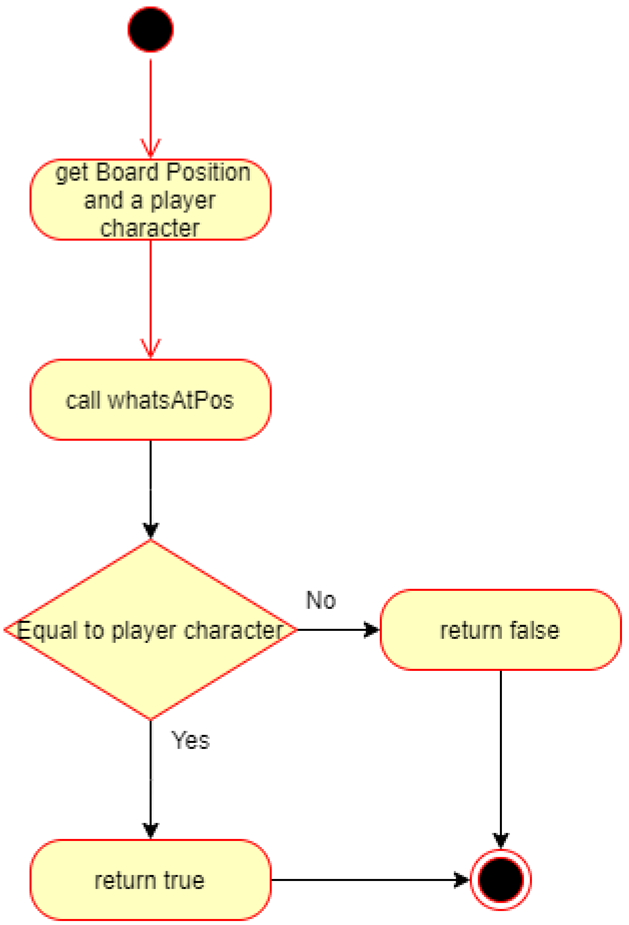
BoardPosition

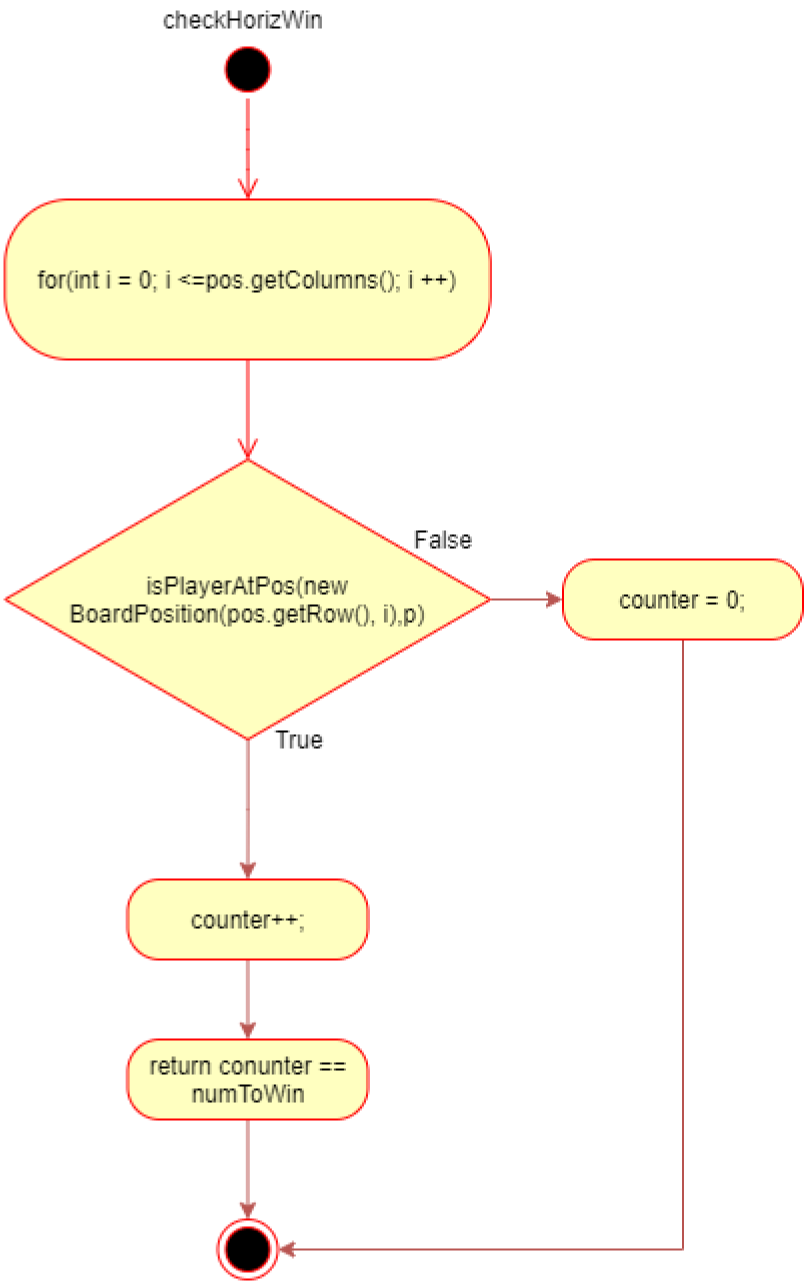


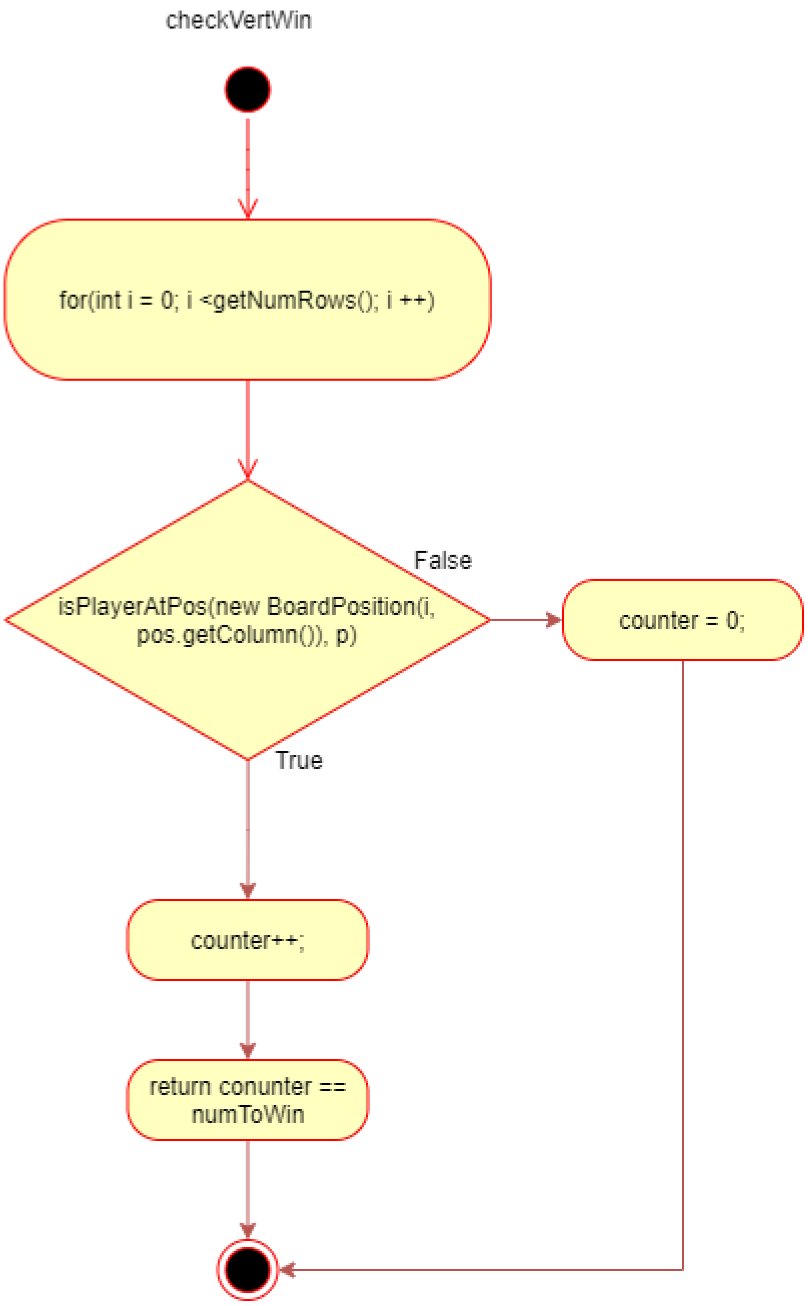
checkIfFree

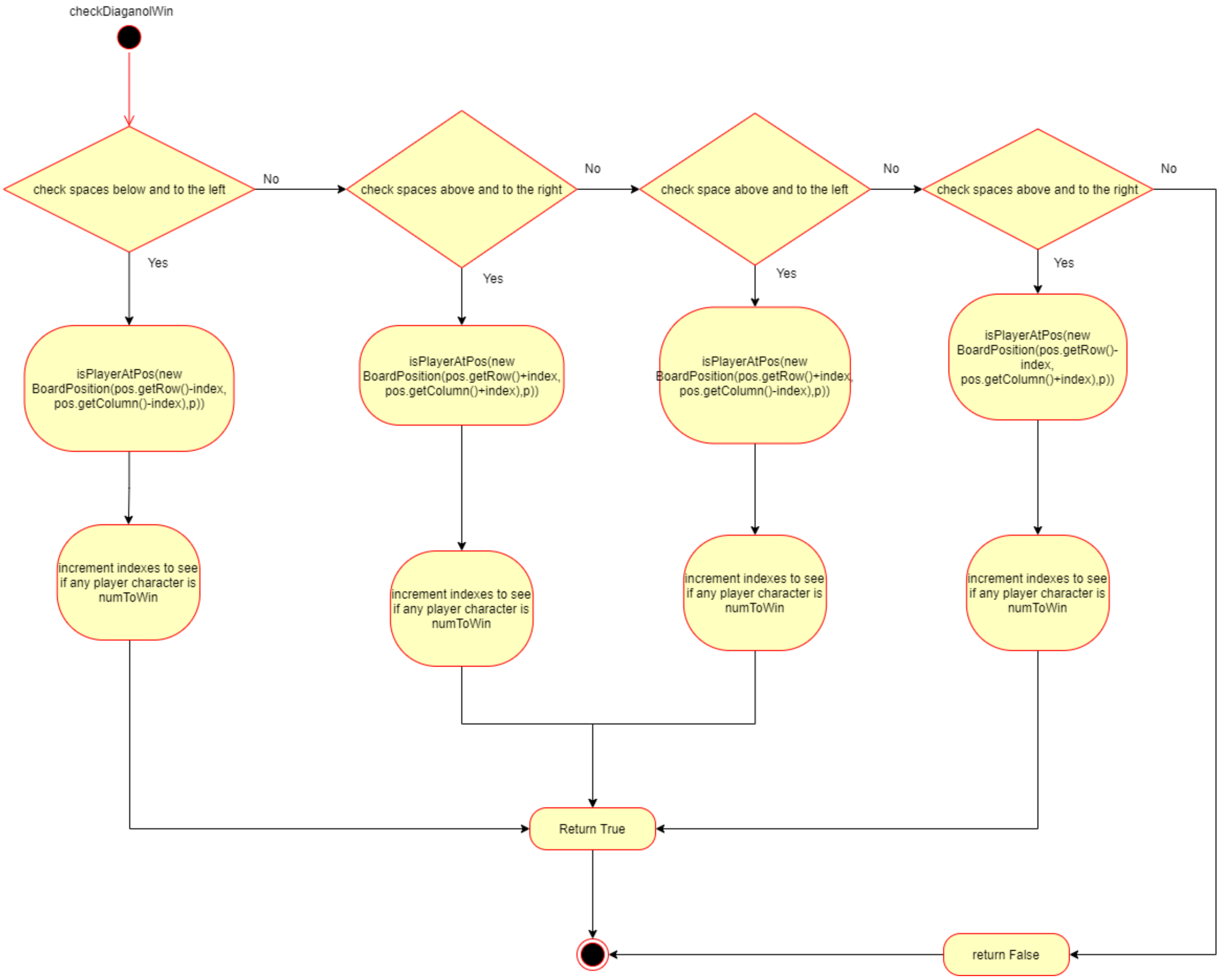


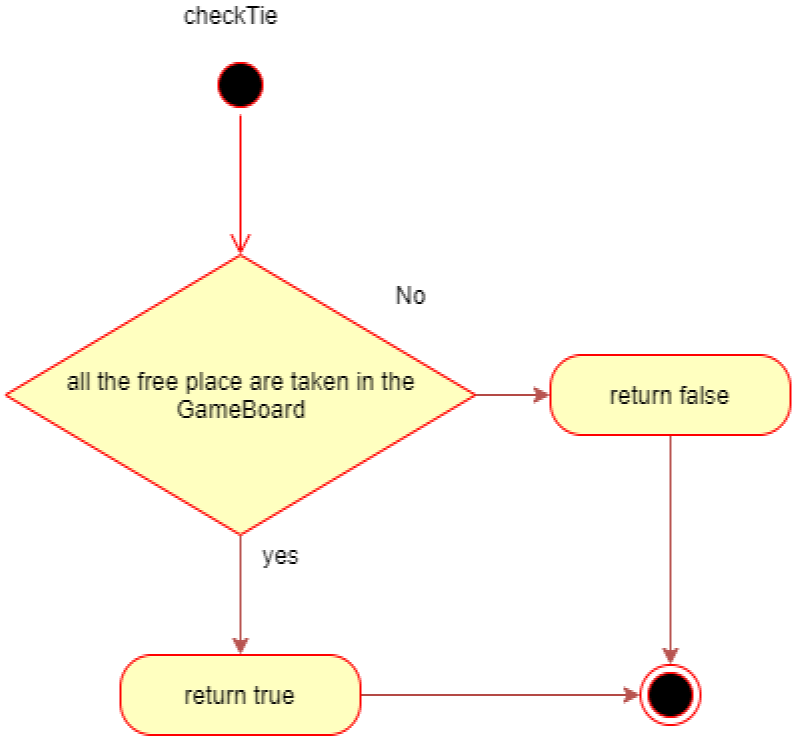
isPlayerAtPos

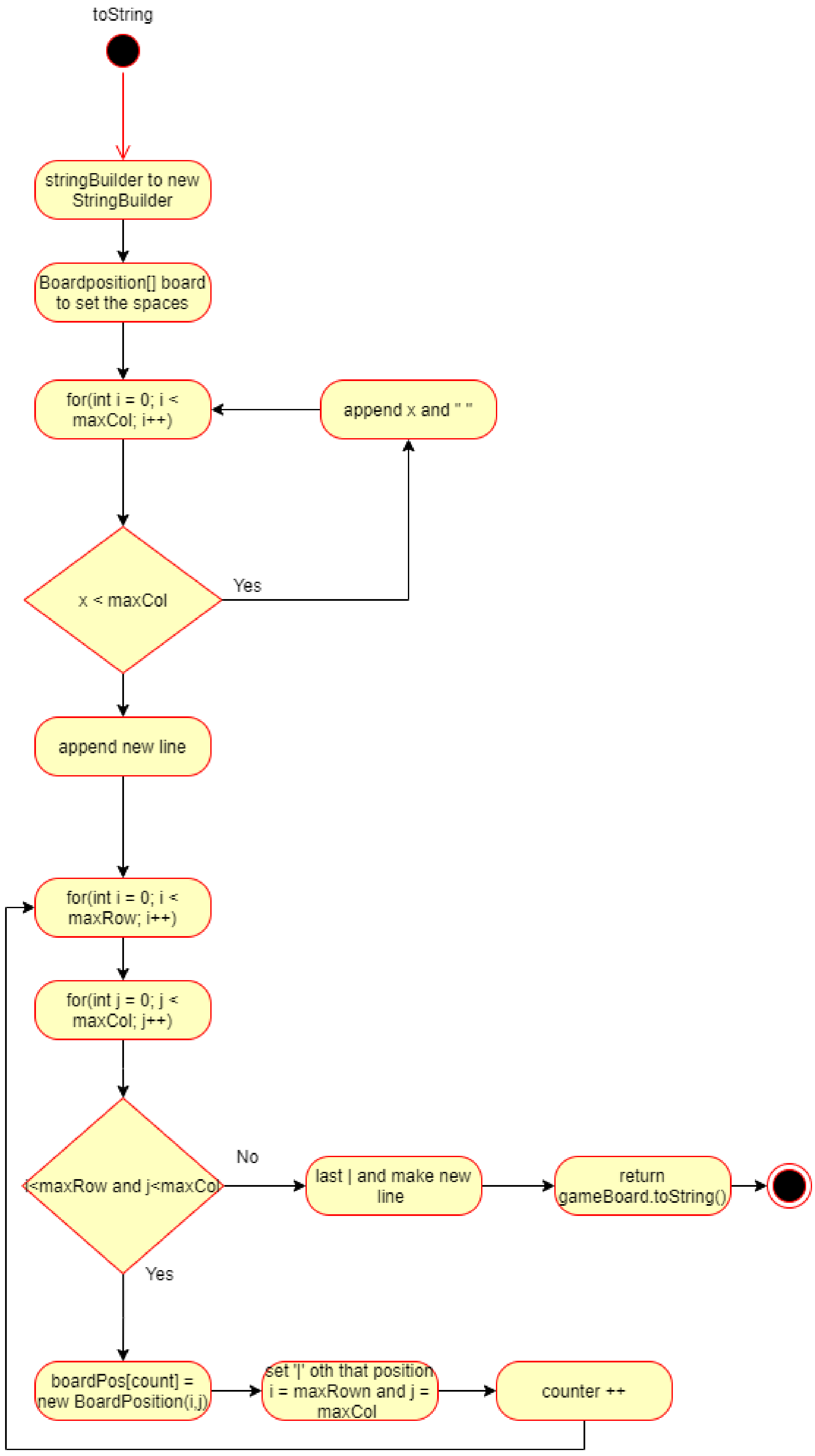

get Board Position and a player character
call whatsAtPos
Equal to player character
No
return false
Yes
return true


checkHorizWin
for(int i = 0; i <=pos.getColumns(); i ++)
isPlayerAtPos(new BoardPosition(pos.getRow(), i),p)
False
counter = 0;
True
counter++;
return conunter == numToWin

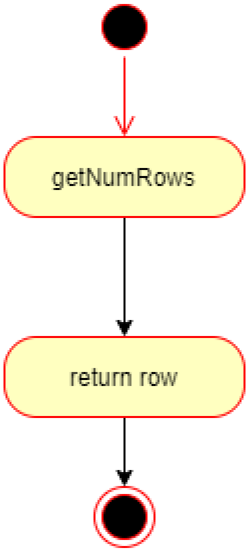

checkVertWin
for(int i = 0; i <getNumRows(); i ++)
isPlayerAtPos(new BoardPosition(i, pos.getColumn()), p)
False
counter = 0;
True
counter++;
return conunter == numToWin


checkDiaganolWin
check spaces below and to the left
No
check spaces above and to the right
No
check space above and to the left
No
check spaces above and to the right
No
Yes
Yes
Yes
Yes
isPlayerAtPos(new BoardPosition(pos.getRow()-index, pos.getColumn()-index),p))
isPlayerAtPos(new BoardPosition(pos.getRow()+index, pos.getColumn()+index),p))
isPlayerAtPos(new BoardPosition(pos.getRow()+index, pos.getColumn()-index),p))
isPlayerAtPos(new BoardPosition(pos.getRow()-index, pos.getColumn()+index),p))
increment indexes to see if any player character is numToWin
increment indexes to see if any player character is numToWin
increment indexes to see if any player character is numToWin
increment indexes to see if any player character is numToWin
Return True
return False

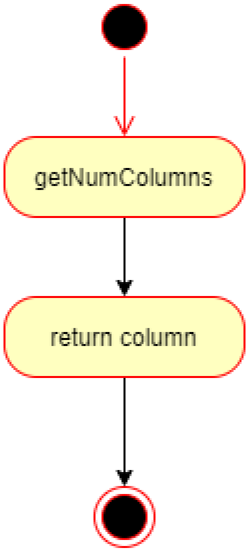

checkTie
No
all the free place are taken in the GameBoard
return false
yes
return true


toString
stringBuilder to new StringBuilder
Boardposition[] board to set the spaces
for(int i = 0; i < maxCol; i++)
append x and " "
x < maxCol
Yes
append new line
for(int i = 0; i < maxRow; i++)
for(int j = 0; j < maxCol; j++)
i<maxRow and j<maxCol
No
last | and make new line
return gameBoard.toString()
Yes
boardPos[count] = new BoardPosition(i,j)
set '|' oth that position i = maxRown and j = maxCol
counter ++

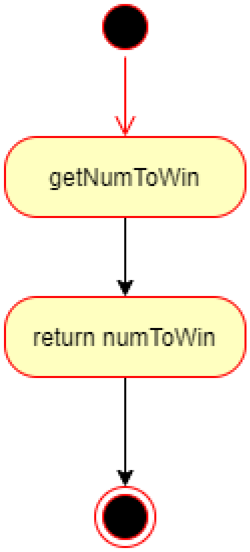
getNumRows



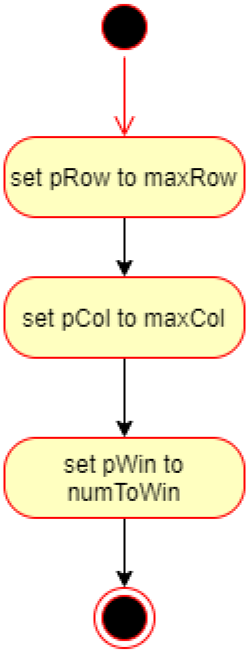
getNumColumns



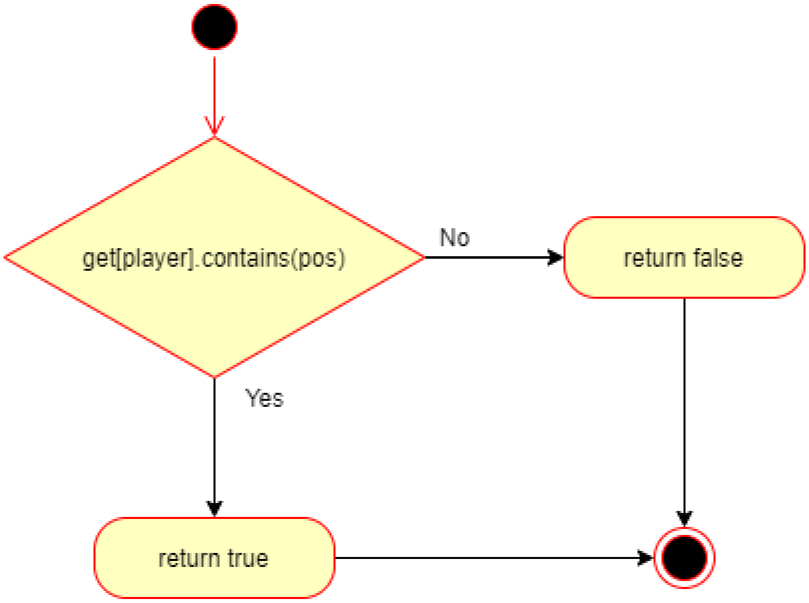
getNumToWin



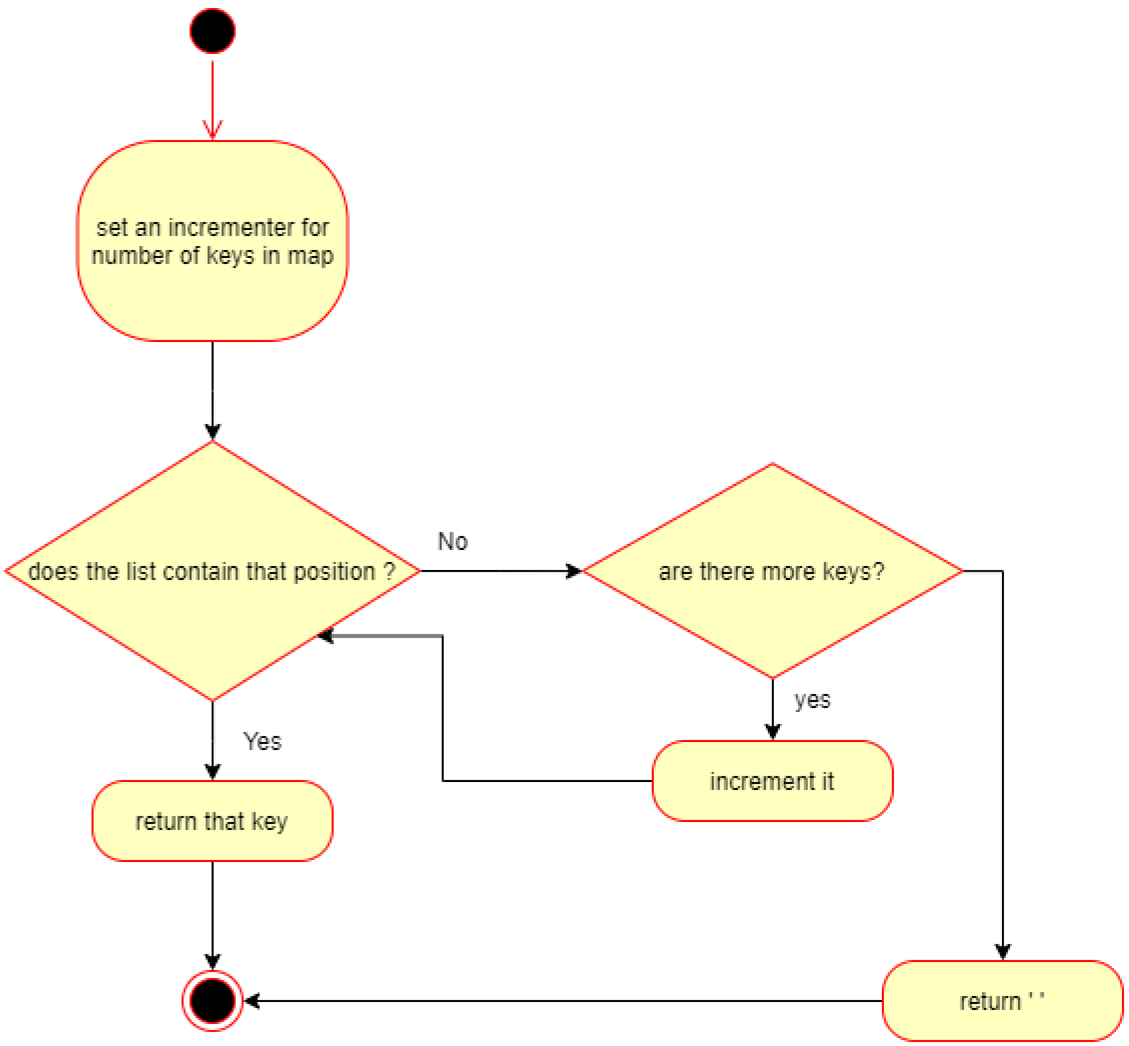
GameBoardMem Constructor



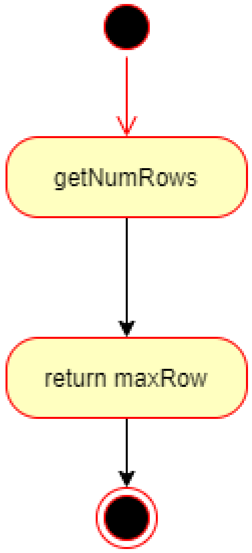
gameboardmem isPlayerAtPos



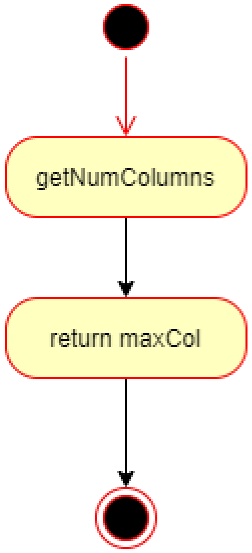
Gameboardmem whatsAtPos


set an incrementer for number of keys in map
does the list contain that position ?
No
are there more keys?
yes
increment it
Yes
return that key
return ' '

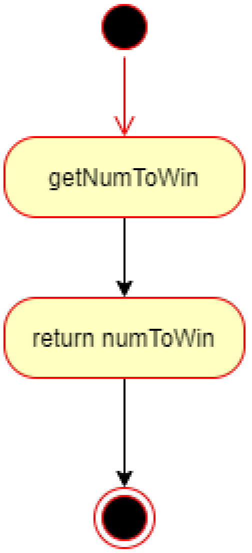
gameboardMem getNumRows

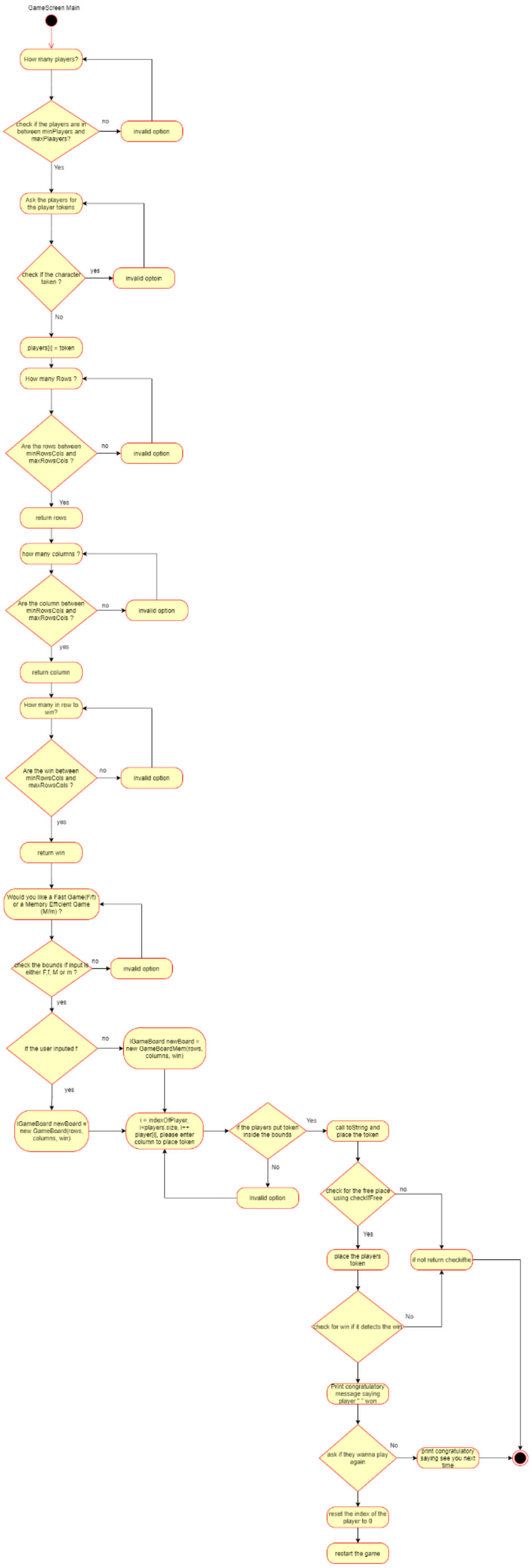


gameboardMem getNumColumns



gameboardMem getNumToWin




GameScreen Main
How many players?
check if the players are in between minPlayers and maxPlaayers?
no
invalid option
Yes
Ask the players for the player tokens
check if the character taken ?
yes
invalid optoin
No
players[i] = token
How many Rows ?
Are the rows between minRowsCols and maxRowsCols ?
no
invalid option
Yes
return rows
how many columns ?
Are the column between minRowsCols and maxRowsCols ?
no
invalid option
yes
return column
How many in row to win?
Are the win between minRowsCols and maxRowsCols ?
no
invalid option
yes
return win
Would you like a Fast Game(F/f) or a Memory Efficient Game (M/m) ?
check the bounds if input is either F,f, M or m ?
no
invalid option
yes
if the user inputed f
no
IGameBoard newBoard = new GameBoardMem(rows, columns, win)
yes
IGameBoard newBoard = new GameBoard(rows, columns, win)
i = indexOfPlayer, i<players.size, i++ player[i], please enter column to place token
if the players put token inside the bounds
Yes
call toString and place the token
No
invalid option
check for the free place using checkIfFree
no
Yes
place the players token
if not return checkiftie
check for win if it detects the win
No
Print congratulatory message saying player " " won
ask if they wanna play again
No
print congratulatory saying see you next time
reset the index of the player to 0
restart the game