Project 5: ConnectX GUI

Author: Kevin Mody

Class: CPSC 2150/2151

**User Story:**

**Functional Requirements:**

1. As a user, I select up to 10 players.
2. As a user, I can choose to have up to 20 rows and minimum of 3 rows
3. As a user, I can choose to have up to 20 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 these tokens: 'X', 'O', 'E', 'H', 'Y', 'A', 'P', 'U', 'J', 'M' .
7. As a user, I can choose to have up to 10 in a row to win but it cannot be more or less 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 If I win, the game will exit with the congratulatory message and I can press any button to restart the game or close the window if I chose not to play.
11. As a user if the game ties , the game will exit with the congratulatory message and I can press any button to restart the game or close the window if I chose not to play.
12. As a user, if I opted in to play again than I should start as any number of player and any player character.
13. 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.
14. 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.
15. As a user, if I place a token at a filled space, it will give me an error and prompt me to rechoose it.
16. 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.
17. 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 can run on Unix but prefer to run on MacOS or Windows.
3. The program runs on IntelliJ.
4. The number of tokens to win are between 3 and 20 inclusive.
5. The number of rows is between 3 and 20 inclusive.
6. The number of columns is between 3 and 20 inclusive.
7. The number of players is between 2 and 10 inclusive.
8. Player 1 always goes first as ‘X’.
9. 0,0 is the bottom left of the board.

**Deployment:**

1. To compile my program, you can simply “run” the program inside IntelliJ.

**Test Cases for GameBoard/GameBoardMem**

Gameboard(int r, int c, int w)

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| Input r = 3  c =3  w = 3 | Output State: | Reasoning:  This test case is distinct because it tests if the constructor can make the smallest possible board |
|  |  | Name: test\_Constructor\_smallest |
|  | board.getNumToWin() = 3 |  |

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)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output checkIfFree(2) = true  Board is unchanged | Reasoning:  This test case is distinct  because it represents a  standard test of a free column |
|  |  | Function name: test\_CheckIfFree\_empty |

boolean checkIfFree(int c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  | X |

|  |  |  |
| --- | --- | --- |
| 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)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  | X |
|  |  |  |  | X |
|  |  |  |  | X |
|  |  |  |  | X |
|  |  |  |  | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output checkIfFree(4) = false  Board is unchanged | Reasoning:  This test case is distinct because it represents a standard test of a column that isn’t free |
|  |  | Function name: test\_CheckIfFree\_full |

boolean checkHorizWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkHorizWin(pos, p) = false | Reasoning:  This test case is distinct because it represents a standard test of there being no horizontal win |
|  | Board is unchanged |  |
|  |  | Function name: test\_CheckHorizontalWin\_emp ty |
| pos = (2, 2) p = ‘X’  board.checkNumToWin() = 3 |  |  |

boolean checkHorizWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | X | X |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkHorizWin(pos, p) = true  Board is unchanged | 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 |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkHorizWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | X | X | X |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkHorizWin(pos, p) = true  Board is unchanged | 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\_mor e\_than\_enough |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkHorizWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | X | O | X |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output checkHorizWin(pos, p) = false | Reasoning:  This test case is distinct because it tests to make sure the function checks the tokens in a row are the same |
|  | Board is unchanged |  |
|  |  | Function name: test\_CheckHorizontalWin\_just  \_not\_enough |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkVertWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkVertWin(pos, p) = false  Board is unchanged | Reasoning:  This test case is distinct because it represents a standard case of there being no vertical win |
|  |  | Function name: test\_CheckVerticalWin\_empty |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkVertWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  | X |  |  |
|  |  | X |  |  |
|  |  | X |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkVertWin(pos, p) = true  Board is unchanged | Reasoning:  This test case is distinct because it represents a standard case of there being a vertical win |
|  |  | Function name: test\_CheckVerticalWin\_just\_e nough |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkVertWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  | X |  |  |
|  |  | X |  |  |
|  |  | X |  |  |
|  |  | X |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkVertWin(pos, p) = true  Board is unchanged | 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\_t han\_enough |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkVertWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  | X |  |  |
|  |  | O |  |  |
|  |  | X |  |  |
|  |  | X |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkVertWin(pos, p) = false  Board is unchanged | 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\_n ot\_enough |
| board.getNumToWin() = 3  Pos = (2,2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = false  Board is unchanged | Reasoning:  This test case is distinct because it represents a standard case of there being no diagonal win |
|  |  | Function name: test\_CheckDiagonalWin\_empt y |
| board.getNumToWin() = 3  Pos = (0,0) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
|  |  |  |  |
|  |  | X |  |
| O | X | X |  |
| X | O | O |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = true Board state is unchanged | 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\_j ust\_enough |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
|  |  |  | X |
|  |  | X | X |
| O | X | X | O |
| X | O | O | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = true Board state is unchanged | 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 |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
|  |  |  | O |
|  |  | O | X |
| O | X | X | O |
| X | O | O | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = false Board state is unchanged | 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\_j ust\_not\_enough |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
|  |  |  |  |
|  | X |  |  |
|  | O | X |  |
|  | O | O | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = true Board state is unchanged | 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 |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| X |  |  |  |
| X | X |  |  |
| O | O | X |  |
| X | O | O | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = true Board state is unchanged | 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 |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  |  |

boolean checkDiagWin(BoardPosition pos, char p)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| O |  |  |  |
| X | X |  |  |
| O | O | X |  |
| X | O | O | X |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  checkDiagWin(pos, p) = false Board state is unchanged | 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 |
| board.getNumToWin() = 3  Pos = (2, 2) P = ‘X’ |  | Function name: test\_CheckDiagonalWin\_right\_ just\_not\_enough |

boolean CheckTie()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output CheckTie() = false  Board is unchanged | Reasoning:  This test case is distinct because it represents the standard case of there being no tie |
|  |  | Function name: test\_CheckTie\_empty |

boolean CheckTie()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

|  |  |  |
| --- | --- | --- |
| Input State: | Output CheckTie() = true  Board is unchanged | Reasoning:  This test case is distinct because it represents the standard case of there being a tie |
|  |  | Function name: test\_CheckTie\_full |

boolean CheckTie()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output CheckTie() = false  Board is unchanged | Reasoning:  This test case is distinct because it ensures that the function checks every column  Function name: test\_CheckTie\_almost\_full |

boolean CheckTie()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

|  |  |  |
| --- | --- | --- |
| Input State: | Output CheckTie() = false  Board is unchanged | 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 |

char whatsAtPos(BoardPosition pos)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output whatsAtPos(pos) = ‘ ‘  Board is unchanged | Reasoning:  This test case is distinct because it represents the standard case of no character being on the position |
|  |  | Function name: test\_WhatsAtPos\_empty |
| Pos = (0,0) |  |  |

char whatsAtPos(BoardPosition pos)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output whatsAtPos(pos) = ‘X ‘  Board is unchanged | 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 |
| Pos = (0,0) |  |  |

char whatsAtPos(BoardPosition pos)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| K |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output whatsAtPos(pos) = ‘E‘  Board is unchanged | 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 |
| Pos = (0,0) |  |  |

char whatsAtPos(BoardPosition pos)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output whatsAtPos(pos) = ‘ ‘  Board is unchanged | Reasoning:  This test case is distinct because it ensures the function is checking the correct position |
|  |  | Function name: test\_WhatsAtPos\_player\_near by |
| Pos = (0,1) |  |  |

char whatsAtPos(BoardPosition pos)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | O |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output whatsAtPos(pos) = ‘O‘  Board is unchanged | Reasoning:  This test case is distinct because it ensures the function will return the correct character |
|  |  | Function name: test\_WhatsAtPos\_two\_players |
| Pos = (0,1) |  |  |

boolean isPlayerAtPos(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output isPlayerAtPos = false  Board is unchanged | Reasoning:  This test case is distinct because it represents the standard case of no character being on the position |
|  |  | Function name: test\_IsPlayerAtPos\_empty |
| Pos = (0,0) P = ‘X’ |  |  |

boolean isPlayerAtPos(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  isPlayerAtPos(pos, p) = true  Board is unchanged | 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 |
| Pos = (0,0) P = ‘X’ |  |  |

boolean isPlayerAtPos(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| K |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  isPlayerAtPos(pos, p) = true  Board is unchanged | 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 |
| Pos = (0,0) P = ‘E’ |  |  |

boolean isPlayerAtPos(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  isPlayerAtPos(pos, p) = false  Board is unchanged | Reasoning:  This test case is distinct because it ensures that the function checks the correct position |
|  |  | Function name: test\_IsPlayerAtPos\_empty\_sp ace\_nearby |
| Pos = (0,1) P = ‘X’ |  |  |

boolean isPlayerAtPos(BoardPosition pos, char p)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | O |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input State: | Output  isPlayerAtPos(pos, p) = true  Board is unchanged | Reasoning:  This test case is distinct because it ensures the function is looking for the correct character |
|  |  | Function name: test\_IsPlayerAtPos\_two\_playe rs |
| Pos = (0,1) P = ‘O’ |  |  |

Void placeToken(char p, int c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input | Output | 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)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  | X |

|  |  |  |
| --- | --- | --- |
| Input | Output | 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)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

|  |  |  |
| --- | --- | --- |
| Input | Output | 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)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| X | O |  |  |  |

|  |  |  |
| --- | --- | --- |
| Input | Output | Reasoning:  This test case is distinct because it is testing if the function can add different tokens  Function name: test\_PlaceToken\_different\_cha racters |

Void placeToken(char p, int c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input | Output   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | S | H | R | E | K | | S | H | R | E | K | | S | H | R | E | K | | S | H | R | E | K | | S | H | R | E | K | | 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 |

UML Class Diagrams

Diagram

Description automatically generated

UML Activity Diagrams

Diagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated