CPSC 2150 Project 4 Report

Skyler Wolf (sWolf-7)
Samuel Jordan (srjorda)
Robby Infinger (RobbyInfinger)
Nicholas Jordan (BallyhooAndBigTop)

Requirements Analysis

Functional Requirements:

1. As a player, I want to define how many tokens needed in a row to win, so I can change the difficulty level.
2. As a player, I want to be able to choose between a fast game implementation, and a memory efficient game, so my computer will be able to handle running the game.
3. As a player, I want to be able to choose the number of rows, so I can change the difficulty level.
4. As a player, I want to be able to choose the number of columns, so I can change the difficulty level.
5. As a player, I want to be able to choose the number of players, so I can play with more friends simultaneously.
6. As a player, I want to be able to specify my own marker so that I can play with whichever character I want.
7. As a player, I want to be prompted again after choosing a taken marker so that all of the markers will be unique and differentiable.
8. As a player, I need to be able to win anytime I connect enough tokens to win in a horizontal direction so I can have a goal to work towards.
9. As a player, I need to be able to win anytime I connect enough tokens to win in a vertical direction so I can have a goal to work towards.
10. As a player, I need to be able to win anytime I connect enough tokens to win in a diagonal direction so I can have a goal to work towards.
11. As a player, I need the game to switch between all players' turns so that I can play with all of my friends.
12. As a player, I need the game to switch between all players' turns so that the game can progress.
13. As a player, I want to be told whose turn it is so that I don't forget.
14. As a player, I need to see the board so that I can understand where I can play.
15. As a player, I want to see an updated board every turn so I can know the current game state.
16. As a player, I need to be able to tell the game where to place my marker so I can try to win.
17. As a player, I want to see the column numbers, so I can place the markers in the correct spot.
18. As a player, I want to be prompted again after making a marker placement on a full column, so I can change my placement.
19. As a player, I want to be prompted again after making a marker placement on a non-existent column, so I can change my placement.
20. As a player, I need to know which player won, so I decide if I want to restart the game or end the program.
21. As a player, I want to see the board after someone wins so I can see the winning move.
22. As a player, I want to see the board if we tie so I can see there are no moves available.
23. As a player, I want to be able to restart the game after the game ends so I can play multiple games in one session.
24. As a player, I can change the game settings after each reset so that I can have each round be different.
25. As a player, I want to be able to end the program after the game ends so I can move on to other tasks.

Non-Functional Requirements

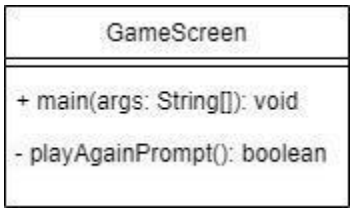
1. The system must be programmed in Java.
2. The program must properly compile and run on Unix.
3. The system must run in the terminal with a command-line interface.
4. The system must be able to execute and run in a timely manner.
5. The system must be encapsulated and not stop running unless told to.
6. The system must differentiate between players via their tokens.
7. The system must start with the first token provided.

8. The system must generate a gameBoard with user defined rows and columns.
9. The system must recognize the origin (0,0) to be the bottom left of the gameBoard.
10. The system must allow the user to choose between two different memory implementations of the game, fast or memory efficient.
11. The system must be deployable by the instructed use of a Makefile
12. The system must be tested for 100% code coverage.

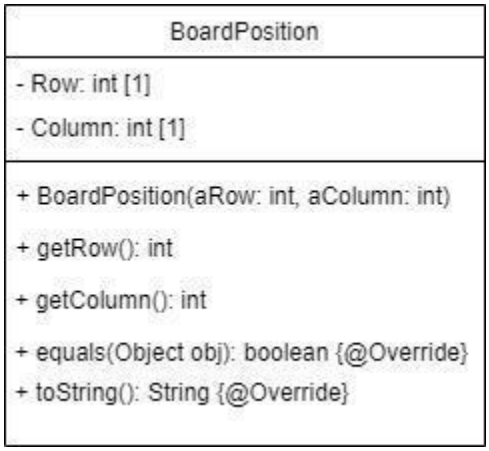
System Design

files documented: GameScreen.java , BoardPosition.java , GameBoard.java , AbsGameBoard.java , and IGameBoard.java

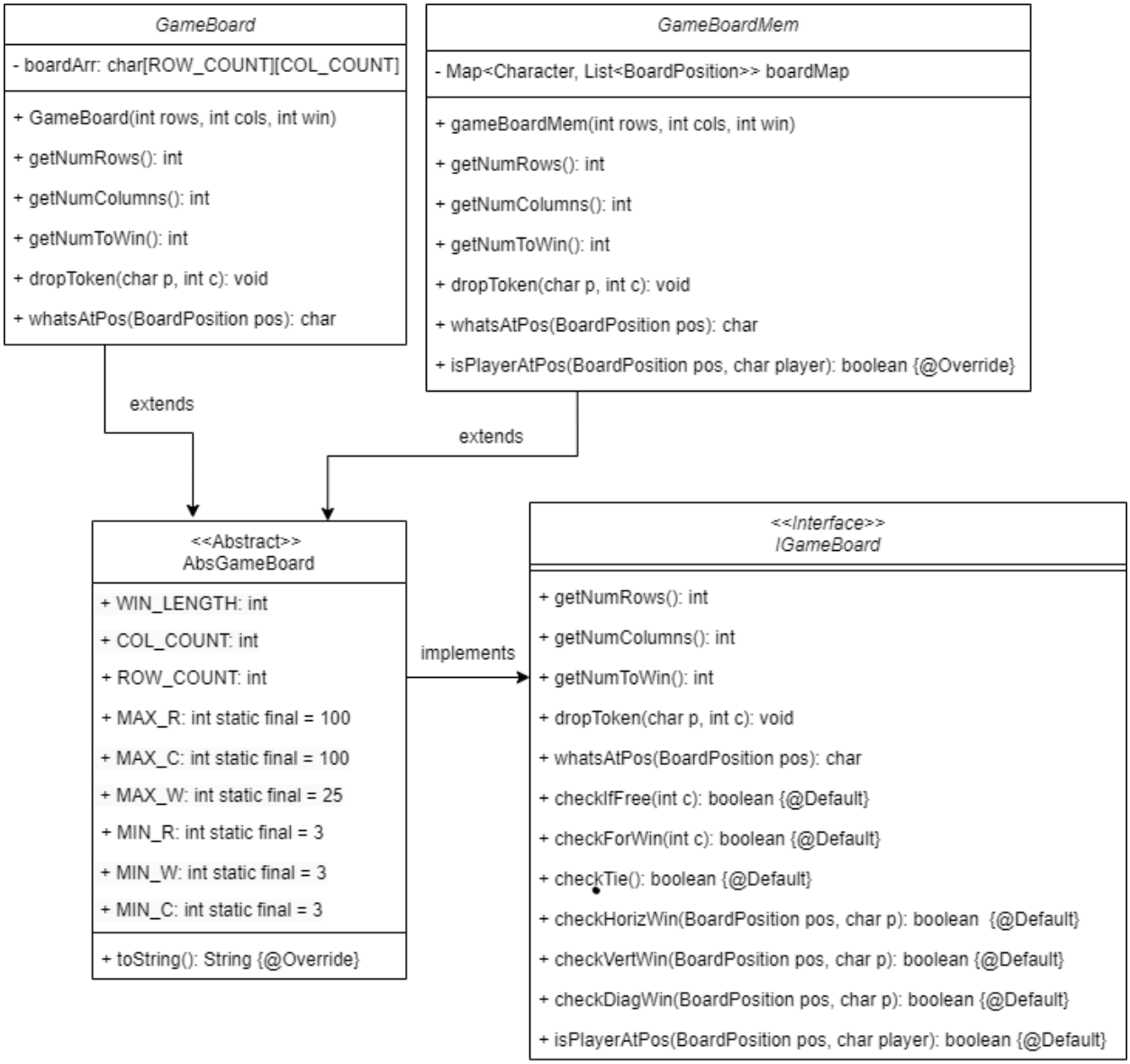
GameScreen.java:



BoardPosition.java:



GameBoard.java {extends} AbsGameBoard.java
GameBoardMem.java {extends} AbsGameBoard.java
AbsGameBoard.java {implements} IGameBoard.java



Testing

Constructor
GameBoard(int rows, int cols, int win) AND GameBoardMem(int rows, int cols, int win)

Input: rows = 4 cols = 5 win = 4 State: not created	**Output:** getNumToWin = 4 State: empty 4rx5c (only ' ') | | | | | | | | | | | | | | | | | | | | | | | |	**Reason:** This test case is unique and distinct because it tests the routine case of having a different height and width and win length each between their min and max **Function Name:** testConstructor_empty4x5_win length4

Input: rows = 3 cols = 3 win = 3 State: not created	**Output:** getNumToWin = 3 State: empty 3rx3c (only ' ') | | | | | | | | |	**Reason:** This test case is unique and distinct because it tests the boundary case of having the minimum height and width and win length **Function Name:** testConstructor_minArguments

Input: rows = 100 cols = 100 win = 25 State: not created	**Output:** getNumToWin = 25 State: empty 100rx100c (only ' ')	**Reason:** This test case is unique and distinct because it tests the boundary case of having the maximum height and width and win length **Function Name:** testConstructor_maxArguments

boolean checkIfFree(int c)

Input: c = 4 State: (WIN_LENGTH = 3) | | | | | | | | | | | | | | | | | |	**Output:** checkIfFree = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests the boundary case of checking an empty column **Function Name:** testCheckIfFree_emptyColumn_True

Input: c = 4 State: (WIN_LENGTH = 3) | | | | | | | | | | O | | | | | X |	**Output:** checkIfFree = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests the routine case of checking a partially full column

		Function Name: testCheckIfFree_partiallyFullColumn_True

Input: c = 4 State: (WIN_LENGTH = 3) [ ][ ][ ][ ][X] [ ][ ][ ][ ][O] [ ][ ][ ][ ][X]	**Output:** checkIfFree = false State: board is unchanged	**Reason:** This test case is unique and distinct because it tests the boundary case of checking an entirely full column **Function Name:** testCheckIfFree_completelyFullColumn_False

boolean checkHorizWin(boardPosition pos, char p)

Input: pos.getRow = 2 pos.getColumn = 4 p = 'X' State: (WIN_LENGTH = 3) [ ][ ][X][X][X] [ ][ ][O][O][X] [ ][ ][X][O][O]	**Output:** checkHorizWin = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests placing the final marker on the far right end of the streak and upper limits of the board **Function Name:** testCheckHorizWin_lastMarkerTopRight_Row2Col4

Input: pos.getRow = 0 pos.getColumn = 0 p = 'X' State: (WIN_LENGTH = 3) [ ][ ][ ][ ][ ] [ ][ ][ ][ ][ ] [X][X][X][ ][ ]	**Output:** checkHorizWin = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests placing the final marker on the far left end of the streak and the lower limits of the board **Function Name:** testCheckHorizWin_lastMarkerBottomLeft_Row0Col0

Input: pos.getRow = 1 pos.getColumn = 2 p = 'X' State: (WIN_LENGTH = 3) [ ][ ][ ][ ][ ] [ ][X][X][X][ ] [ ][X][O][O][ ]	**Output:** checkHorizWin = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests placing the final marker in the middle of the streak and board **Function Name:** testCheckHorizWin_lastMarkerMiddle_Row1Col2

<table>
<tr>
<td>Input:
pos.getRow = 0
pos.getColumn = 3
p = ‘X’
State: (WIN_LENGTH = 3)
<table>
<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>
</td>
<td>Output:

checkHorizWin = false

State: board is unchanged</td>
<td>Reason:
This test case is unique and distinct because it tests placing a marker not resulting in a long enough sequence despite there being enough in the same row

Function Name:
testCheckHorizWin_enoughMarkersNoWin_Row0Col3</td>
</tr>
</table>

boolean checkVertWin(boardPosition pos, char p)

<table>
<tr>
<td>Input:
pos.getRow = 2
pos.getColumn = 0
p = ‘X’
State: (WIN_LENGTH = 3)
<table>
<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>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td>Output:

checkVertWin = true

State: board is unchanged</td>
<td>Reason:
This test case is unique and distinct because it tests the final marker being placed in a stack at the board’s lower limits

Function Name:
testCheckVertWin_stackBottomLeft_Row2Col0</td>
</tr>
</table>

<table>
<tr>
<td>Input:
pos.getRow = 4
pos.getColumn = 4
p = ‘X’
State: (WIN_LENGTH = 3)
<table>
<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>O</td></tr>
<tr><td></td><td></td><td></td><td></td><td>O</td></tr>
</table>
</td>
<td>Output:

checkVertWin = true

State: board is unchanged</td>
<td>Reason:
This test case is unique and distinct because it tests the final marker in the stack being placed at the board’s upper limits

Function Name:
testCheckVertWin_stackTopRight_Row4Col4</td>
</tr>
</table>

<table>
<tr>
<td>Input:
pos.getRow = 3
pos.getColumn = 2
p = ‘X’
State: (WIN_LENGTH = 3)
<table>
<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>O</td><td></td><td></td></tr>
</table>
</td>
<td>Output:

checkVertWin = true

State: board is unchanged</td>
<td>Reason:
This test case is unique and distinct because it tests placing a marker in neither the highest or lowest possible positions for row and column

Function Name:
testCheckVertWin_stackInMiddle_Row3Col2</td>
</tr>
</table>

Input: pos.getRow = 4 pos.getColumn = 0 p = 'X' State: (WIN_LENGTH = 3) X O X X O	**Output:** checkVertWin = false State: board is unchanged	**Reason:** This test case is unique and distinct because it tests placing a marker not resulting in a long enough sequence despite there being enough in the same column **Function Name:** testCheckVertWin_enoughMarkersNoWin_Row4Col0

boolean checkDiagWin(boardPosition pos, char p)

Input: pos.getRow = 0 pos.getColumn = 0 p = 'X' State: (WIN_LENGTH = 3) X X O X O O	**Output:** checkDiagWin = true State: board is unchanged	**Reason:**This test case is unique and distinct because it checks if a win is recognized on a southwest to northeast diagonal when the most southwest token is placed last **Function Name:** testCheckDiagWin_SWtoNE_SWlast

Input: pos.getRow = 1 pos.getColumn = 1 p = 'X' State: (WIN_LENGTH = 3) X X O X O O	**Output:** checkDiagWin = true State: board is unchanged	**Reason:** This test case is unique and distinct because it checks if a win is recognized on a southwest to northeast diagonal when a middle token is placed last **Function Name:** testCheckDiagWin_SWtoNE_middleLast

Input: pos.getRow = 2 pos.getColumn = 2 p = 'X' State: (WIN_LENGTH = 3) X X O X O O	**Output:** checkDiagWin = true State: board is unchanged	**Reason:** This test case is unique and distinct because it checks if a win is recognized on a southwest to northeast diagonal when the most northeast token is placed last **Function Name:** testCheckDiagWin_SWtoNE_NElast

Input:
pos.getRow = 2
pos.getColumn = 0
p = 'X'
State: (WIN_LENGTH = 3)

X		
O	X	
O	O	X

Output:

checkDiagWin = true

State: board is unchanged

Reason: This test case is unique and distinct because it checks if a win is recognized on a northwest to southeast diagonal when the most northwest token is placed last

Function Name: testCheckDiagWin_NWtoSE_NWlast

Input:
pos.getRow = 1
pos.getColumn = 1
p = 'X'
State: (WIN_LENGTH = 3)

X		
O	X	
O	O	X

Output:

checkDiagWin = true

State: Board is unchanged

Reason: This test case is unique and distinct as it checks if a win is recognized on a northwest to southeast diagonal when a middle token is placed last

Function Name: testCheckDiagWin_NWtoSE_middleLast

Input:
pos.getRow = 0
pos.getColumn = 2
p = 'X'
State: (WIN_LENGTH = 3)

X		
O	X	
O	O	X

Output:

checkDiagWin = true

State: Board is unchanged

Reason: This test case is unique and distinct as it checks if a win is recognized on a northwest to southeast diagonal when the most southeast token is placed last

Function Name: testCheckDiagWin_NWtoSE_SElast

Input:
pos.getRow = 2
pos.getColumn = 2
p = 'X'
State: (WIN_LENGTH = 3)

X	O	X
O	O	X
X	X	O

Output:

checkDiagWin = false

State: Board is unchanged

Reason: This test case is unique and distinct as it checks if the function returns false when the last token dropped doesn't cause a diagonal win

Function Name: testCheckDiagWin_noWin

boolean checkTie()

Input:
 	
State: (WIN_LENGTH = 3)
`  ` | `  ` | `  `
`  ` | `  ` | `  `
`X` | `O` | `X`
	Output:
	
	checkTie = false
	
	State: board is unchanged
		Reason:
		This test case is unique and distinct because it tests player 1 placing a marker and not resulting in tied game state
		
		Function Name:
		testCheckTie_player1_false

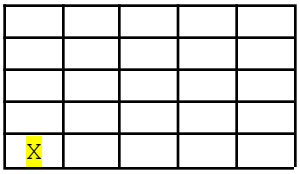
Input:
 	
State: (WIN_LENGTH = 3)
`X` | `O` | `X`
`O` | `O` | `X`
`X` | `X` | `O`
	Output:
	
	checkTie = true
	
	State: board is unchanged
		Reason:
		This test case is unique and distinct because it tests player 1 placing a marker and resulting in tied game state
		
		Function Name:
		testCheckTie_player1_true

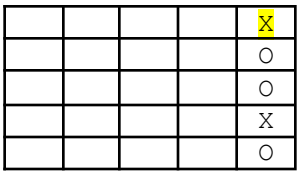
Input:
 	
State: (WIN_LENGTH = 3)
`  ` | `  ` | `  `
`O` | `  ` | `  `
`O` | `X` | `X`
	Output:
	
	checkTie = false
	
	State: board is unchanged
		Reason:
		This test case is unique and distinct because it tests player 2 placing a marker and not resulting in tied game state
		
		Function Name:
		testCheckTie_player2_false

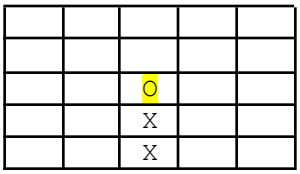
Input:
 	
State: (WIN_LENGTH = 3)
`X` | `O` | `X` | `O`
`O` | `X` | `O` | `X`
`O` | `X` | `O` | `X`
	Output:
	
	checkTie = true
	
	State: board is unchanged
		Reason:
		This test case is unique and distinct because it tests player 2 placing a marker and resulting in tied game state
		
		Function Name:
		testCheckTie_player2_true

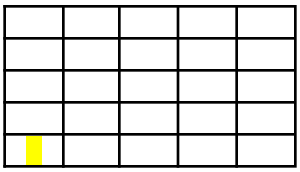
char whatsAtPos(BoardPosition pos)

Input:
pos.getRow = 1
pos.getColumn = 2
State: (WIN_LENGTH = 4)
`  ` | `  ` | `  ` | `  ` | `  `
`  ` | `  ` | `  ` | `  ` | `  `
`  ` | `  ` | `  ` | `  ` | `  `
`  ` | `  ` | `X` | `  ` | `  `
`  ` | `  ` | `O` | `  ` | `  `
	Output:
	
	whatsAtPos = 'X'
	
	State: board is unchanged
		Reason:
		This test case is unique and distinct because it tests the routine case between the min and max row/column value
		
		Function Name:
		testWhatsAtPos_routine_Row1Col2

Input: pos.getRow = 0 pos.getColumn = 0 State: (WIN_LENGTH = 4) 	**Output:** whatsAtPos = 'X' State: board is unchanged	**Reason:** This test case is unique and distinct because it checks the marker at the lower limits of the board **Function Name:** testWhatsAtPos_bottomLeft_Row0Col0

Input: pos.getRow = 4 pos.getColumn = 4 State: (WIN_LENGTH = 4) 	**Output:** whatsAtPos = 'X' State: board is unchanged	**Reason:** This test case is unique and distinct because it checks the marker at the upper limits of the board **Function Name:** testWhatsAtPos_topRight_Row4Col4

Input: pos.getRow = 2 pos.getColumn = 2 State: (WIN_LENGTH = 4) 	**Output:** whatsAtPos = 'O' State: board is unchanged	**Reason:** This test case is unique and distinct because it checks a non-X marker at the center of the board **Function Name:** testWhatsAtPos_centerO_Row2Col2

Input: pos.getRow = 0 pos.getColumn = 0 State: (WIN_LENGTH = 4) 	**Output:** whatsAtPos = ' ' State: board is unchanged	**Reason:** This test case is unique and distinct because it checks a position on an empty board with no markers **Function Name:** testWhatsAtPos_emptyBoard_Row0Col0

boolean isPlayerAtPos(BoardPosition pos, char player)

Input: pos.getRow = 0 pos.getColumn = 0 player = 'X' State: (WIN_LENGTH = 3) |		
X				**Output:** isPlayerAtPos = true State: board is unchanged	**Reason:** This test case is unique and distinct because it tests to make sure true is returned when the marker is actually at the position specified **Function Name:** testIsPlayerAtPos_true

Input: pos.getRow = 0 pos.getColumn = 2 player = 'X' State: (WIN_LENGTH = 3) |		
X	O		**Output:** isPlayerAtPos = false State: board is unchanged	**Reason:** This test case is unique and distinct because it tests to make sure isPlayerAtPos can identify when the player is not at the position, even when the marker does exist elsewhere on the board and the current spot is occupied by some token **Function Name:** isPlayerAtPos_false_existsElsewhere

Input: pos.getRow = 0 pos.getColumn = 0 player = 'X' State: (WIN_LENGTH = 3) |		
			Output: isPlayerAtPos = false State: board is unchanged	**Reason:** This test case is unique and distinct because it ensures the function returns false when the board is completely empty, meaning no markers exist on the board **Function Name:** isPlayerAtPos_false_emptyBoard

Input: pos.getRow = 2 pos.getColumn = 2 player = 'Z' State: (WIN_LENGTH = 3) |		X
		O
		X	**Output:** isPlayerAtPos = false State: board is unchanged	**Reason:** This test case is unique and distinct because it ensures the function returns false when the given marker does not exist on the board at all, but others do **Function Name:** isPlayerAtPos_false_tokenNotOnBoard

Input: pos.getRow = 1 pos.getColumn = 1 player = 'X' State: (WIN_LENGTH = 3) | X		
O			**Output:** isPlayerAtPos = false State: board is unchanged	**Reason:** This test case is unique and distinct because it ensures the function returns false when the given position is empty but the marker does exist on the board elsewhere

X		

Function Name:
isPlayerAtPos_false_emptySpace

void dropToken(char p, int c)

Input:

State: (WIN_LENGTH = 3)

p = 'X'
C = 0

Output:

State: (WIN_LENGTH = 3)

X		

Reason:
A standard case ensuring that you can place a token down onto an empty board, testing the lower limits of the rows and columns

Function Name:
testDropToken_emptyBoard

Input:

State: (WIN_LENGTH = 3)

X		

p = 'O'
C = 0

Output:

State: (WIN_LENGTH = 3)

O		
X		

Reason:
Ensuring that you can place a token down in non-empty column and have it land above the previously placed token; new token doesn't fill the column

Function Name:
testDropToken_notFullRow

Input:

State: (WIN_LENGTH = 3)

X		

p = 'O'
C = 1

Output:

State: (WIN_LENGTH = 3)

X	O	

Reason:
Ensures that you can place a token in an empty row of a non-empty board, having it fall to the bottom despite there being a token somewhere in the bottom row

Function Name:
testDropToken_newRow_true

Input:

State: (WIN_LENGTH = 3)

X		
O		

p = 'O'
C = 0

Output:

State: (WIN_LENGTH = 3)

O		
X		
O		

Reason:
Ensuring that the token will land at the top of a nearly full column rather than always falling to a lower level

Function Name:
testDropToken_FillColumn

Input:

State: (WIN_LENGTH = 3)

O	X	
X	Z	Y
O	Y	X

Output:

State: (WIN_LENGTH = 3)

O	X	Z
X	Z	Y
O	Y	X

Reason:
Ensuring that the token will land at the top of a nearly full column of a nearly full board rather than always falling to a lower level,

p = 'Z' C = 2		testing the upper limits of the rows and columns **Function Name:** testDropToken_FullBoard