CPSC 2150 Project 4 Report

Grant Benson, Daniel Chaput, Brianna Cherry

Requirements Analysis

Functional Requirements

1. As a player, I want the game to prompt me with how many players I would like to have at least 2 up to 10 so that I can play with more than two players.
2. As a player, I want to be able to insert a custom character to use as my player token so that I can play the game with more customizability.
3. As a player, I want to be able to choose the amount of rows and columns that the board contains at least 3 and up to 100 rows/columns so that I can resize the board to my liking.
4. As a player, I want to be able to choose the number of tokens that are required to win the game by lining up with at least 3 tokens and up to 25 tokens so that I can make the game harder and last longer.
5. As a player, I want to be able to choose between playing a game that makes the calculations required quickly and a game that is efficient with memory so that I can optimize my experience.
6. As a player, I want to see the current game board displayed on the screen at the start of my turn, so that I know where I should put my next token.
7. As a player, I want to choose a column to drop my token into during my turn so that I can control where my token goes.
8. As a player, I want the game to check if my move results in four tokens in a row either horizontally, vertically, or diagonally, so that I can win the game.
9. As a player, I want the game to let me know and try again if I choose a column that is already full so that I can choose a valid place for my token.
10. As a player, I want the game to let me know if I choose an invalid column outside the grid so that I can choose a different column and place my token.
11. As a player, I want the game to notify me if my move results in a tie game when all columns are filled without a win, so that the game does not continue forever.
12. As a player, I want the game to declare me as the winner and end the game if I get the number of tokens that I chose in a row, so that it displays the winning board and ends the game.
13. As a player, I want the game to give me the option to play again after a game has ended so that I can choose to start a new game or exit.
14. As a player, I want the game to alternate turns between each and every player so that we each get turns placing tokens.
15. As a player, I want the game to follow the same rules and display the same game board as all other players so that we can play a fair game.
16. As a player, I should be able to use the makefile to compile the program, run the program, and clean the program files

Non-Functional Requirements

1. Program should be able to update and re-print the game board quickly with minimal lag time.
2. Program should be written in Java.
3. Program should be written in a way that makes it easy to change the game board dimensions from game to game.
4. The game should recognize errors in the players' input and be able to respond without crashing.
5. Program should run on Unix.
6. (0,0) will always be the position of the bottom left corner of the board
7. The game board should be of size rows x columns based on what the user chose
8. There should be two separate implementations of the game board, one that is quick and one that is memory efficient. The quick implementation will use a 2D array and the other will use a hashmap.
9. The first turn of a new game should always start with player 1

Makefile Instructions

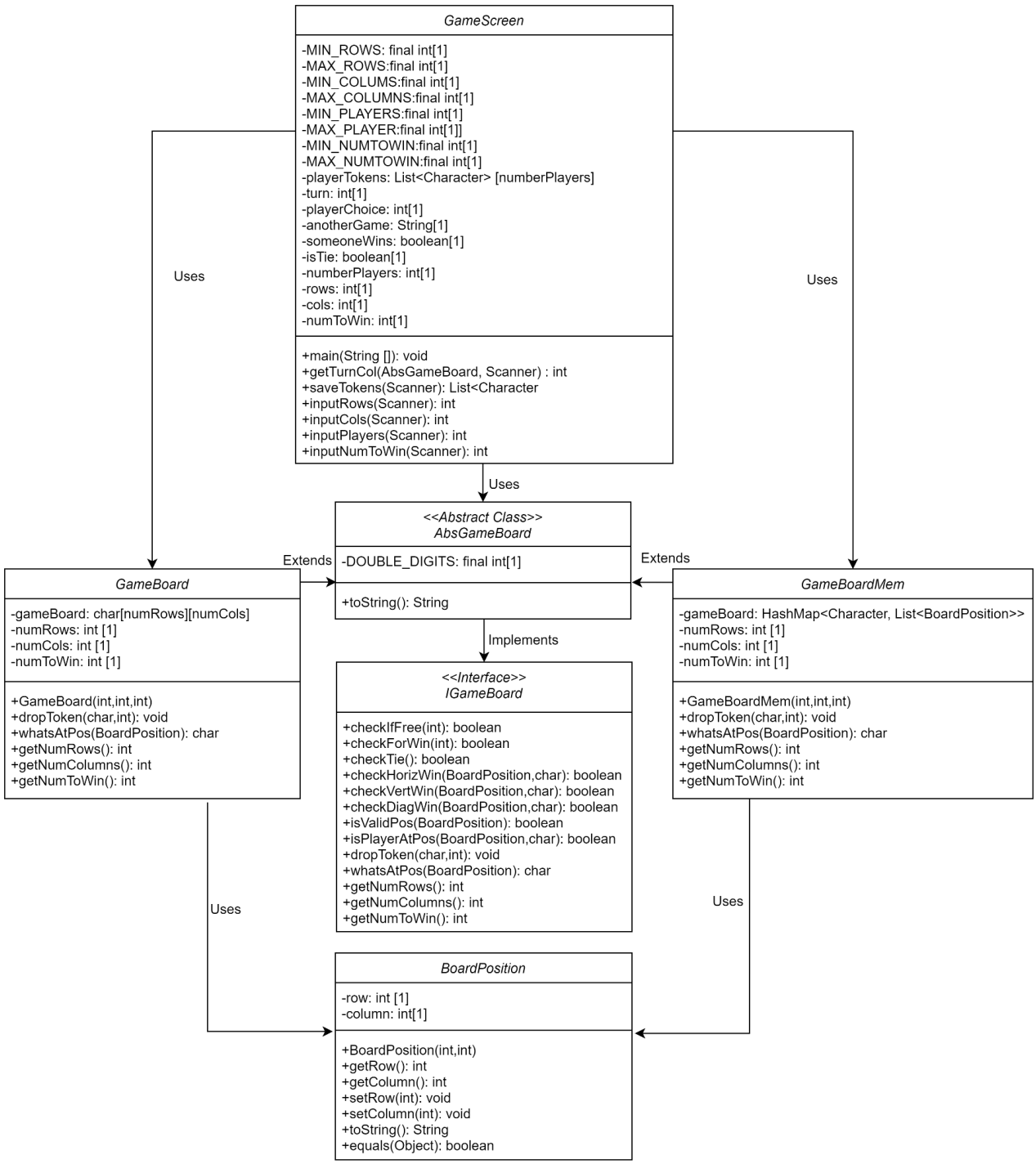
Running the Program

1. Navigate to the folder containing the makefile.
2. To compile the code, type “make” or “make default”, then [ENTER].
3. To run the now compiled program, type “make run”, then [ENTER].
4. To clean all of the compiled .class files, type “make clean”, then [ENTER].

Testing the Program

1. Navigate to the folder containing the makefile.
2. To compile the test cases, type “make test”, then [ENTER]
3. To run all the tests for the normal game board, type “make testGB”, then [ENTER]
4. To run all the tests for the memory efficient game board, type “make testGBmem”, then [ENTER]

System Design



Test Cases

boolean checkIfFree(int c)

Input: State: (number to win = 4) c = 0	**Output:** State: unchanged checkIfFree = true	**Reason:** This test case is unique because it is checking an empty first column, ensuring that the function works for the first column and ensuring that it works when it is almost full. **Function Name:** testCheckIfFree_empty_first_almost_full_true

o				
x				
o				
x				

boolean checkIfFree(int c)

Input: State: (number to win = 4) c = 4	**Output:** State: unchanged checkIfFree = true	**Reason:** This test case is unique because it is checking a completely empty last column, ensuring that the function works for the last column and ensuring that it works when it is completely empty. **Function Name:** testCheckIfFree_empty_last_true

boolean checkIfFree(int c)

Input: State: (number to win = 4)	**Output:** State: unchanged checkIfFree = false	**Reason:** This test case is unique because it is checking a full column, ensuring that it works when a column is full. **Function Name:** testCheckIfFree_full_middle_false

		x		
		o		
		x		
		o		

		x					

c = 2

boolean checkHorizWin(BoardPostion pos, char p)

Input: State: (number to win = 4)	**Output:** State: unchanged	**Reason:** This test case is unique because the last x character was placed in the top right of the board, ensuring the function works for the edge case of the max number of rows and columns.

	x	x	x	x
	o	x	x	x
	x	o	o	o
	o	x	x	o
o	x	o	o	o

pos.getRow() = 4
pos.getCol() = 4
p = ‘x’

checkHorizWin = true

Function Name:
testCheckHorizWin_top_right_true

boolean checkHorizWin(BoardPostion pos, char p)

Input: State: (number to win = 4)	**Output:** State: unchanged	**Reason:** This test case is unique because the last x character was placed in the bottom left position of the board. This ensures that the function works for the edge case of the minimum rows and columns.

	o	o	o	
x	x	x	x	

pos.getRow() = 0
pos.getCol() = 0
p = ‘x’

checkHorizWin = true

Function Name:
testCheckHorizWin_bottom_left_true

boolean checkHorizWin(BoardPostion pos, char p)

Input: State: (number to win = 4)	**Output:** State: unchanged	**Reason:** This test case is unique because the last x character placed did not result in a win, and instead resulted in the number of x’s in a row to be 3. This ensures that

checkHorizWin = false

	o	o		
	x	x	x	

pos.getRow() = 0
pos.getCol() = 3
p = ‘x’

	the function checks the correct number of characters in a row, not one less character. **Function Name:** testCheckHorizWin_middle_oneLessThenNumToWin_false

boolean checkHorizWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

o	o		o	
x	x	x	x	

pos.getRow() = 0
pos.getCol() = 2
p = ‘x’

Output:
State: unchanged

checkHorizWin = true

Reason:
This test case is unique because the last x character was placed in the middle of the board and it resulted in a win. This ensures that the function is able to check both sides of the last placed character for a win.

Function Name:
testCheckHorizWin_bottom_middle_true

boolean checkVertWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

				o
			x	o
			x	o
			x	o
		x	o	x

pos.getRow() = 4
pos.getCol() = 4
p = ‘o’

Output:
State: unchanged

checkVertWin = true

Reason:
This test case is unique because the last o character was placed in the very top right of the board and resulted in a win, ensuring that the function works for this edge case of the maximum number of columns and rows.

Function Name:
testCheckVertWin_top_right_true

boolean checkVertWin(BoardPostion pos, char p)

Input: State: (number to win = 4) o o x o x o x x o x pos.getRow() = 4 pos.getCol() = 0 p = ‘o’	**Output:** State: unchanged checkVertWin = true	**Reason:** This test case is unique because the last o character was placed in the very top left of the board, ensuring that the function works for this edge case of the minimum number of columns. **Function Name:** testCheckVertWin_top_left_true

boolean checkVertWin(BoardPostion pos, char p)

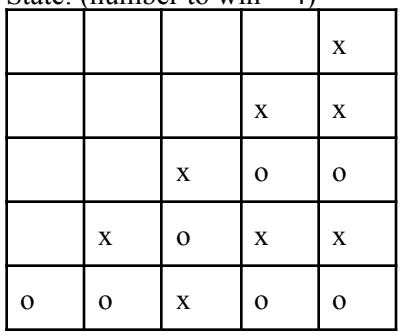
Input: State: (number to win = 4) x x o x o pos.getRow() = 2 pos.getCol() = 2 p = ‘x’	**Output:** State: unchanged checkVertWin = false	**Reason:** This test case is unique because the last x character placed did not result in a win, and instead resulted in the number of x’s in a row to be 3. This ensures that the function checks the correct number of characters in a row, not one less character. **Function Name:** testCheckVertWin_middle_oneLessThanNumToWin_false

boolean checkVertWin(BoardPostion pos, char p)

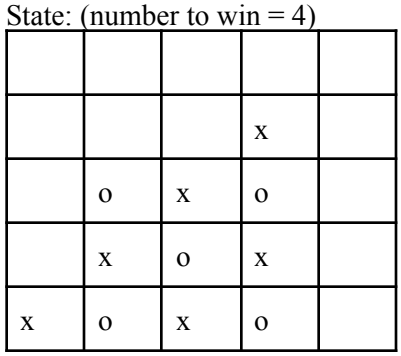
Input: State: (number to win = 4) x x x x x o x o o x o o x o o x o	**Output:** State: unchanged checkVertWin = false	**Reason:** This test case is unique because the last x character placed resulted in a horizontal win and a diagonal win, but not a vertical win. This ensures that the function is checking for a vertical win and not the other win cases. **Function Name:** testCheckVertWin_other_win_cases_false

pos.getRow() = 3 pos.getCol() = 3 p = ‘x’		

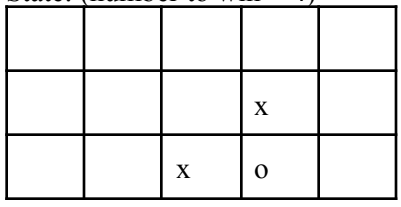
boolean checkDiagWin(BoardPostion pos, char p)

Input: State: (number to win = 4)  pos.getRow() = 4 pos.getCol() = 4 p = ‘x’	**Output:** State: unchanged checkDiagWin = true	**Reason:** This test case is unique because the last x character placed resulted in a win in the SW/NE diagonal direction and was also an edge case of making sure it checks the maximum rows and columns of the board. **Function Name:** testCheckDiagWin_SW_NE_top_right_true

boolean checkDiagWin(BoardPostion pos, char p)

Input: State: (number to win = 4)  pos.getRow() = 0 pos.getCol() = 0 p = ‘x’	**Output:** State: unchanged checkDiagWin = true	**Reason:** This test case is unique because the last x character placed resulted in a win in the SW/NE diagonal direction and was also an edge case of making sure it checks the minimum rows and columns of the board. **Function Name:** testCheckDiagWin_SW_NE_bottom_left_true

boolean checkDiagWin(BoardPostion pos, char p)

Input: State: (number to win = 4) 	**Output:** State: unchanged checkDiagWin = true	**Reason:** This test case is unique because the last x character was placed in the middle and resulted in a win in the SW/NE diagonal direction. This ensures that the function checks both to the SW and to the NE of the last placed token.

	x	o	x	
x	o	o	x	o

pos.getRow() = 2
pos.getCol() = 2
p = ‘x’

Function Name:
testCheckDiagWin_SW_NE_middle_true

boolean checkDiagWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

x				
o	x	o		
x	o	x		
o	x	o	x	
x	o	x	o	

pos.getRow() = 4
pos.getCol() = 0
p = ‘x’

Output:
State: unchanged

checkDiagWin = true

Reason:
This test case is unique because the last x character placed resulted in a win in the NW/SE diagonal direction and was also an edge case of making sure it checks the maximum rows and the minimum columns of the board.

Function Name:
testCheckDiagWin_NW_SE_top_left_true

boolean checkDiagWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

	x	o		
	o	x	o	
	o	x	x	
o	x	o	x	x

pos.getRow() = 0
pos.getCol() = 4
p = ‘x’

Output:
State: unchanged

checkDiagWin = true

Reason:
This test case is unique because the last x character placed resulted in a win in the NW/SE diagonal direction and was also an edge case of making sure it checks the minimum rows and the maximum columns of the board.

Function Name:
testCheckDiagWin_NW_SE_bottom_right_true

boolean checkDiagWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

Output:
State: unchanged

checkDiagWin = true

Reason:
This test case is unique because the last x character was placed in the middle and resulted in a win in the NW/SE diagonal

	x			
	o	x	o	
	x	o	x	
	o	x	o	x

pos.getRow() = 2
pos.getCol() = 2
p = ‘x’

	direction. This ensures that the function checks both to the NW and to the SE of the last placed token. **Function Name:** testCheckDiagWin_NW_SE_middle_true

boolean checkDiagWin(BoardPostion pos, char p)

Input:
State: (number to win = 4)

	x		x	
	o	x	o	
	x	o	x	
o	x	o	x	o

pos.getRow() = 2
pos.getCol() = 2
p = ‘x’

Output:
State: unchanged

checkDiagWin = false

Reason:
This test case is unique because the last x character was placed in the middle of the board and did not result in a win, but did get one less than numToWin in a row diagonally in both diagonal directions. This ensures that the function checks the correct number of characters in a row, not one less character.

Function Name:
testCheckDiagWin_oneLessThanNumToWin_false

boolean checkTie()

Input:
State: (number to win = 4)

x	x	o	o	x
o	o	x	x	o
x	x	o	o	x
o	o	x	x	o
x	x	o	o	x

Output:
State: unchanged

checkTie = true

Reason:
This test case is unique because the board is entirely full with no win cases.

Function Name:
testCheckTie_full_noWin_true

boolean checkTie()

Input: State: (number to win = 4)	**Output:** State: unchanged	**Reason:** This test case is unique because

x	x	x	x	o
o	o	x	x	o
x	o	o	o	x
o	x	x	x	o
x	o	o	x	o

checkTie = false

the board is entirely full with there being a win case generated as the last token was placed.

Function Name:
testCheckTie_full_win_false

boolean checkTie()

Input:
State: (number to win = 4)

x	o	x	o	
o	o	x	o	o
x	x	o	x	x
o	o	x	o	o
x	x	o	x	x

Output:
State: unchanged

checkTie = false

Reason:
This test case is unique because the board is almost entirely full with no win cases, leaving only the top right spot empty. This ensures that the function checks the top right corner of the board when checking for a tie.

Function Name:
testCheckTie_notFull_noWin_topRightEmpty_false

boolean checkTie()

Input:
State: (number to win = 4)

Output:
State: unchanged

checkTie = false

Reason:
This test case is unique because the board is entirely empty, ensuring that the test case works for the edge case of an empty board.

Function Name:
testCheckTie_empty_false

boolean isPlayerAtPos(BoardPostion pos, char p)

Input:
State: (number to win = 4)

Output:
State: unchanged

Reason:
This test case is unique because the position that we're checking

				x
				o
				x
				o
				x

pos.getRow() = 4
pos.getCol() = 4
p = ‘o’

isPlayerAtPos = true

for is the very top right corner of the board and the player token we’re looking for is at that position, so it should return true.

Function Name:
testIsPlayerAtPos_top_right_true

boolean isPlayerAtPos(BoardPostion pos, char p)

Input:
State: (number to win = 4)

x				

pos.getRow() = 0
pos.getCol() = 0
p = ‘x’

Output:
State: unchanged

isPlayerAtPos = true

Reason:
This test case is unique because the position that we’re checking for is the very bottom left corner of the board and the player token we’re looking for is at that position, so it should return true.

Function Name:
testIsPlayerAtPos_bottom_left_true

boolean isPlayerAtPos(BoardPostion pos, char p)

Input:
State: (number to win = 4)

		x		

pos.getRow() = 0
pos.getCol() = 2
p = ‘o’

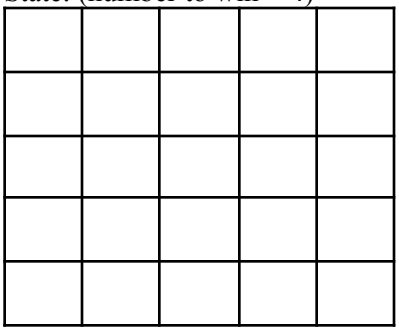
Output:
State: unchanged

isPlayerAtPos = false

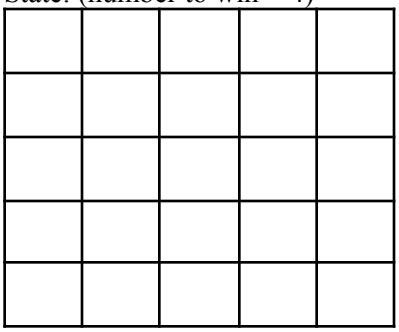
Reason:
This test case is unique because the position that we’re checking for is the middle of the board and there is a player token present in the position we’re examining, however it is not the correct player token, so it should return false.

Function Name:
testIsPlayerAtPos_bottom_middle_wrongToken_false

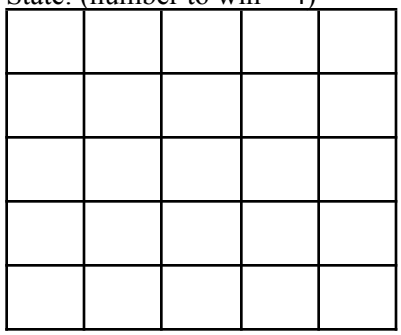
boolean isPlayerAtPos(BoardPostion pos, char p)

Input:	**Output:**	**Reason:**
State: (number to win = 4)  pos.getRow() = 2 pos.getCol() = 2 p = ‘x’	State: unchanged isPlayerAtPos = false	This test case is unique because the position that we’re checking for is empty, so it should return false since we’re checking for ‘x’. **Function Name:** testIsPlayerAtPos_middle_empty_false

boolean isPlayerAtPos(BoardPostion pos, char p)

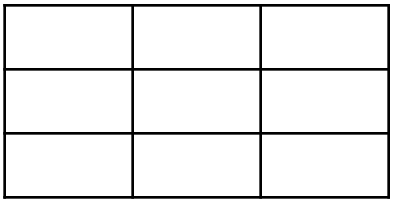
Input:	**Output:**	**Reason:**
State: (number to win = 4)  pos.getRow() = 4 pos.getCol() = 4 p = ‘ ’	State: unchanged isPlayerAtPos = true	This test case is unique because we are checking an empty position for the space ‘ ’ character, which is the placeholder for an empty position. This ensures that it was initialized correctly / that spaces occupy the empty spots. **Function Name:** testIsPlayerAtPos_top_right_empty_true

GameBoard(int r, int c, int n) w

Input:	**Output:**	**Reason:**
State: game board does not exist r = 5 c = 5 n = 4	State: (number to win = 4)  this.getNumRows() = 5	This test case is unique because it ensures the game board correctly sets the parameters of the board during instantiation. **Function Name:** test_constructor_setParameters

	this.getNumCols() = 5 this.getNumToWin() = 4	

GameBoard(int r, int c, int n)

Input: State: game board does not exist r = 3 c = 3 n = 3	**Output:** State: (number to win = 4)  this.getNumRows() = 6 this.getNumCols() = 5 this.getNumToWin() = 4	**Reason:** This test case is unique because it ensures that the constructor correctly creates a board filled with whitespaces when the minimum dimensions are used. **Function Name:** test_constructor_smallestBoard

GameBoard(int r, int c, int n)

Input: State: game board does not exist r = 100 c = 100 n = 25	**Output:** State: [GameBoard is a 100x100 board of cells containing the whitespace ‘ ‘ character] this.getNumRows() = 100 this.getNumCols() = 100 this.getNumToWin() = 25	**Reason:** This test case is unique because it ensures that the constructor correctly creates a board of whitespaces when the maximum dimensions are used. This is especially relevant in the GameBoardMem class, for which the structure of the board might be more reactive to different dimensions than GameBoard. **Function Name:** test_constructor_biggestBoard

void dropToken(char p, int c)

Input: State: (numberToWin = 4, rows = 5, columns = 5) 	**Output:** State: (numberToWin = 4, rows = 5, columns = 5) 	**Reason:** This test case is unique because it ensures that dropToken correctly places a token in an empty column. **Function Name:** testDropToken_ValidEmptyColumn

p = ‘x’ c = 0	x	

void dropToken(char p, int c)

Input: State: (numberToWin = 4, rows = 5, columns = 5) p = ‘o’ c = 3	**Output:** State: (numberToWin = 4, rows = 5, columns =5) o x	**Reason:** This test case is unique because it ensures that dropToken correctly places a token in a column that already has tokens but is not full. **Function Name:** testDropToken_ValidNonEmpty Column

void dropToken(char p, int c)

Input: State: (numberToWin = 4, rows = 5, columns = 5) p = ‘o’ c = 4	**Output:** State: (numberToWin = 4, rows = 5, columns =5) o x x x x	**Reason:** This test case is unique because it ensures that dropToken correctly places a token in the top right position of the board, ensuring that it is able to do so correctly. **Function Name:** testDropToken_top_right

void dropToken(char p, int c)

Input:
State: (numberToWin = 4,
rows = 5,
columns = 5)

			x	
			o	
			x	
			o	
			x	

p = ‘o’
c = 2

Output:
State: UNCHANGED

Reason:
This test case is unique because it ensures that dropToken correctly does NOT place a token (or otherwise change the state of the board) for a column that is already full.

Function Name:
testDropToken_OnlyFullCol

void dropToken(char p, int c)

Input:
State: (numberToWin = 3,
rows = 4,
columns = 4)

b	a	b	a
a	b	a	b
b	a	b	a

p = ‘c’
c = 1

Output:
State:
(numberToWin = 3,
rows = 4,
columns =4)

	c		
b	a	b	a
a	b	a	b
b	a	b	a

Reason:
This test case is unique because it ensures that dropToken can place a token on a board that has many tokens placed in such a way that

A) The token is in the correct spot, and
B) The other tokens are not changed

Function Name:
testDropToken_validCol_nonEmptyBoard

char whatsAtPos(BoardPosition pos)

Input:
State: (numberToWin = 3,
rows = 4,
columns = 4)

d			
c			

Output:
State: UNCHANGED

whatsAtPos = ‘d’

Reason:
This test case is unique because it ensures that whatsAtPos() works for the upper left corner cell, showing that it works for a full lowest column.

Function Name:
testWhatsAtPos_checking_NW_

<table><tr><td>b</td><td></td><td></td><td></td></tr><tr><td>a</td><td></td><td></td><td></td></tr></table> pos.getRow() = 3 pos.getColumn() = 0		corner

char whatsAtPos(BoardPosition pos)

Input: State: (numberToWin = 3, rows = 4, columns = 4) <table><tr><td>d</td><td></td><td></td><td></td></tr><tr><td>c</td><td></td><td></td><td></td></tr><tr><td>b</td><td></td><td></td><td></td></tr><tr><td>a</td><td></td><td></td><td></td></tr></table> pos.getRow() = 0 pos.getColumn() = 0	**Output:** State: UNCHANGED whatsAtPos = ‘a’	**Reason:** This test case is unique because it ensures that whatsAtPos() works for the lower left corner cell, showing that it works for a empty lowest column. **Function Name:** testWhatsAtPos_checking_SW_corner

char whatsAtPos(BoardPosition pos)

Input: State: (numberToWin = 3, rows = 4, columns = 4) <table><tr><td></td><td></td><td></td><td></td><td>e</td></tr><tr><td></td><td></td><td></td><td></td><td>d</td></tr><tr><td></td><td></td><td></td><td></td><td>c</td></tr><tr><td></td><td></td><td></td><td></td><td>b</td></tr><tr><td></td><td></td><td></td><td></td><td>a</td></tr></table> pos.getRow() = 4 pos.getColumn() = 4	**Output:** State: UNCHANGED whatsAtPos = ‘e’	**Reason:** This test case is unique because it ensures that whatsAtPos() works for the lower right corner cell, showing that it works for a full highest column. **Function Name:** testWhatsAtPos_checking_NE_corner

char whatsAtPos(BoardPosition pos)

Input: State: (numberToWin = 3,	**Output:** State: UNCHANGED	**Reason:** This test case is unique because

rows = 4, columns = 4) <table><tr><td></td><td></td><td></td><td>d</td></tr><tr><td></td><td></td><td></td><td>c</td></tr><tr><td></td><td></td><td></td><td>b</td></tr><tr><td></td><td></td><td></td><td>a</td></tr></table> pos.getRow() = 0 pos.getColumn() = 3	whatsAtPos = ‘a’	it ensures that whatsAtPos() works for the lower right corner cell, showing that it works for an empty highest column. **Function Name:** testWhatsAtPos_checking_SE_corner

char whatsAtPos(BoardPosition pos)

Input: State: (numberToWin = 3, rows = 4, columns = 4) <table><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>o</td><td></td></tr><tr><td>o</td><td></td><td>x</td><td></td></tr><tr><td>x</td><td>x</td><td>o</td><td></td></tr></table> pos.getRow() = 2 pos.getColumn() = 2	**Output:** State: UNCHANGED whatsAtPos = ‘o’	**Reason:** This test case is unique because it ensures that whatsAtPos() works for a cell that is not on the edge or corner of the board. In combination with each corner test, shows that whatsAtPos can handle the border and inner cells of the game board. **Function Name:** testWhatsAtPos_center_cell_NonEmptyBoard