CPSC 2150 Project 4 Report

Owen Merwarth, Gavin Garland, Andrew Hoover, & Ben Simmons

Requirements Analysis

Functional Requirements:

1. As a player, I can provide the number of rows for the game board so that I can choose different-sized game boards.
2. As a player, I can provide the number of columns for the game board so that I can choose different-sized game boards.
3. As a player, I can choose the tokens needed in a row to win the game so that I can play with different game rules.
4. As a player, I can choose how many players will play the game so that I can play with multiple people.
5. As a player, I can choose my unique token so that I can choose a token that I like.
6. As a player, I can choose if I want to play the game with a fast game implementation or a memory-efficient game implementation so that I can play with the version that better suits my needs.
7. As a player, I can see who's turn it is so that I know who should be placing a token.
8. As a player, I can see the current game board during each turn so that I can decide where to place my next token.
9. As a player, I can input a column so that I can place my token in the desired position.
10. As a player, I can see when I have tried to place a token in a column that doesn't exist so that I can place my token in an existing column on the board.
11. As a player, I can see when I have tried to place a token in a column that is full so that I can place my token in an open column on the board.
12. As a player, I can place another token if the previous player has not won the game on their last turn so I can try to win the game.
13. As a player, I cannot place another token if the previous player has won the game on their last turn so that the game can be finished.
14. As a player, I cannot place another token if a tie occurs after the previous player placed a token so that the game can be finished.
15. As a player, I can win the game if I place the chosen number of tokens in a row to win vertically so that I can beat my opponent and win the game.
16. As a player, I can win the game if I place the chosen number of tokens in a row to win horizontally so that I can beat my opponent and win the game.
17. As a player, I can win the game if I place the chosen number of tokens in a row to win diagonally so that I can beat my opponent and win the game.
18. As a player, I can see if I have won the game so that I know when I have won the game.
19. As a player, I can see if another player has won the game so that I know when I have lost the game.

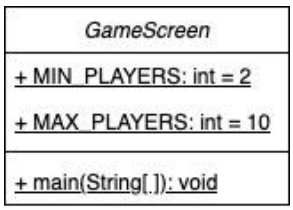
20. As a player, I can achieve a tie if all of the columns on the board are filled without a player reaching the number of tokens in a row to win so that I can avoid losing to my opponent(s).
21. As a player, I can see if there has been a tie so that I know when a tie has been reached.
22. As a player, I can see the winning game board if there is a winner so that I know how the game was won.
23. As a player, I can choose to play again so that I can play the game more than once.
24. As a player, I can change the number of players who are playing if I play again so that I can play with different players.
25. As a player, I can change the size of the game board if I choose to play again so that I can play with different-sized game boards in different games.
26. As a player, I can change the number of tokens in a row to win if I play again so that I can play with different rules.
27. As a player, I can change my choice to play the game with a fast game implementation or a memory-efficient game implementation so that I can play with whichever version I want each time.
28. As a player, I can choose not to play the game again so that I can stop playing the game when I am finished.

Non-Functional Requirements

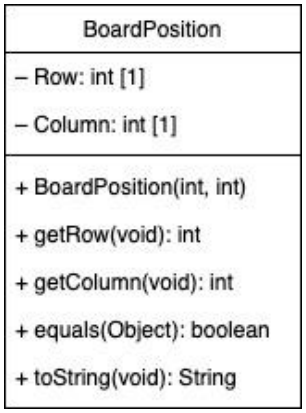
1. The program should process user actions quickly, having a response time of less than 1 second.
2. The game must run on a Unix system and be playable from the command line.
3. The program should be readable, clear, and easy to understand from the command line.
4. The program needs to accurately display information, such as who's turn it is or whether a player has won, to avoid confusion.
5. The program should be easy to update back-end functionality without redoing the front-end display, and vice-versa.
6. The program should be implemented in Java and print all game messages in English.
7. The program should be able to handle user input errors, such as inputting invalid or full columns, without affecting the game.
8. The game should support at least two, but no more than 10, players playing on a single computer.
9. The game should not allow more than one player to play with the same token.
10. The game board must allow at least 3, but no more than 100, rows on a single board.
11. The game board must allow at least 3, but no more than 100, columns on a single board.
12. The game board should be implemented as both a 2D array of characters and a Map, with either being fully supported.
13. The game board should have (0, 0) as the bottom left corner.
14. The game must require at least 3, but no more than 25 tokens in a row, horizontally, vertically, or diagonally, to win.
15. The game must not allow the number of tokens in a row required to win to be more than the number of rows or columns.

System Design

GameScreen:

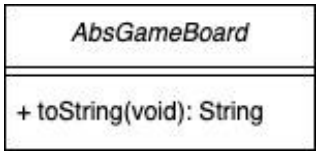


BoardPosition:



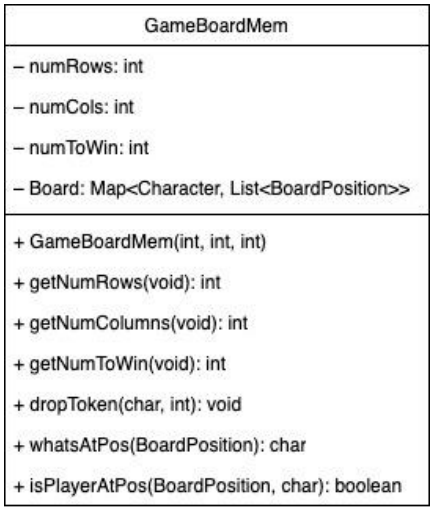
IGameBoard:

AbsGameBoard:



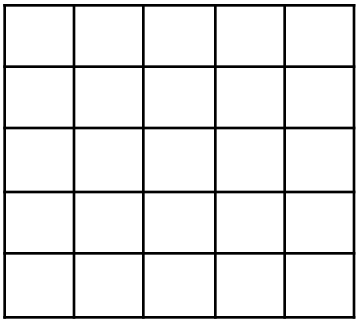
GameBoard:

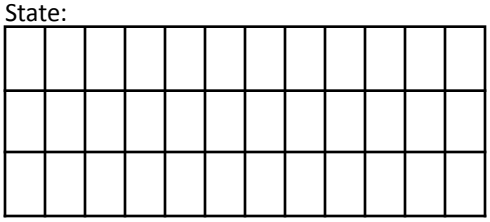
GameBoardMem:

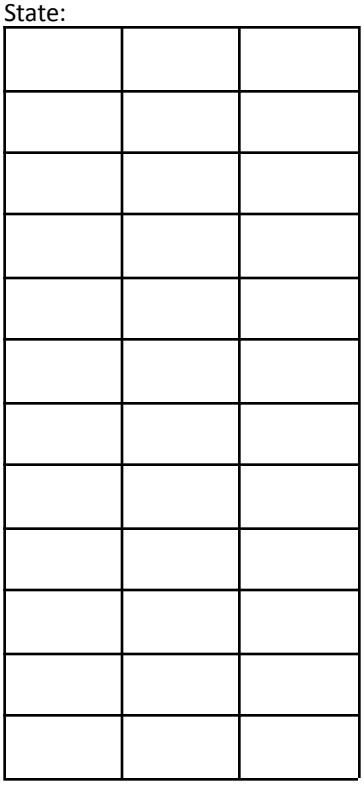


Testing

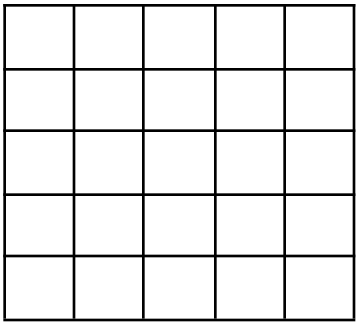
GameBoard(int rows, int cols, int winSize) OR GameBoardMem(int rows, int cols, int winSize)

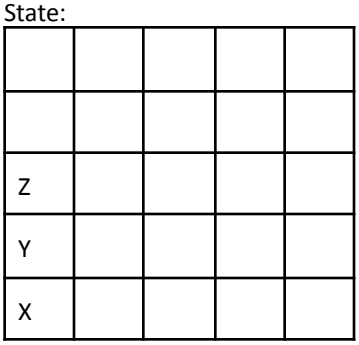
Input:	**Output:**	**Reason:**
rows = 5 cols = 5 winSize = 3	State: 	This test case is unique because it creates a square game board between the limits of the game board size. **Function Name:** testGameBoard_Constructor_5Rows5Columns

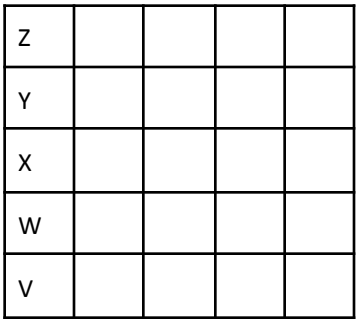
Input:	**Output:**	**Reason:**
rows = 3 cols = 12 winSize = 3	State: 	This test case is unique because it creates a game board with the smallest allowed number of rows and a double digit number of columns **Function Name:** testGameBoard_Constructor_3Rows12Columns

Input:	**Output:**	**Reason:**
rows = 12 cols = 3 winSize = 3	State: 	This test case is unique because it creates a game board with the smallest allowed number of columns and a double digit number of rows **Function Name:** testGameBoard_Constructor_12Rows3Columns

boolean checkIfFree(int c)

Input:	**Output:**	**Reason:**
State:  c = 0	checkIfFree = true The state of the board is unchanged.	This test case is unique because the column checked is completely empty. The column has no tokens in it, so it should accept tokens. **Function Name:** testGameBoard_checkIfFree_0_EmptyColumn_true

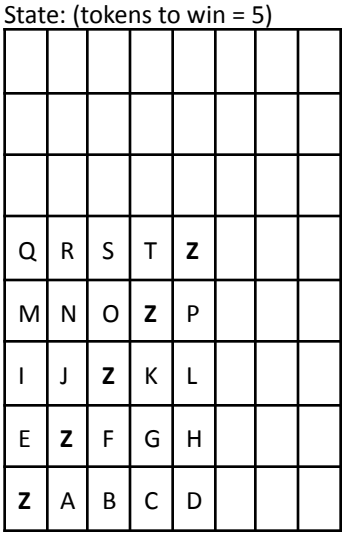
Input: State:  c = 0	**Output:** checkIfFree = true The state of the board is unchanged.	**Reason:** This test case is unique because the column checked is not fully empty nor completely full. The column already has some tokens in it, but it can still accept more. **Function Name:** testGameBoard_checkIfFree_0_PartiallyFullColumn_true

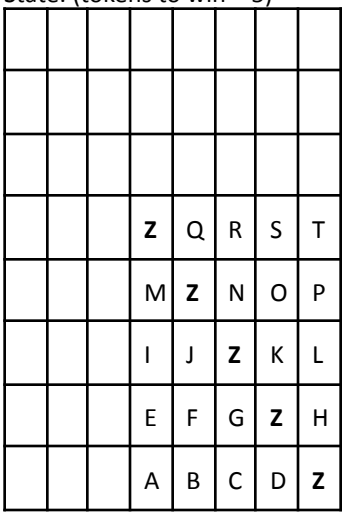
Input: State:  c = 0	**Output:** checkIfFree = false The state of the board is unchanged.	**Reason:** This test case is unique because the column checked is completely full. The column is full of tokens, so it should not accept any more. **Function Name:** testGameBoard_checkIfFree_0_FullColumn_false

boolean checkDiagWin(BoardPosition pos, char p)

Input:	**Output:**	**Reason:**
State: (tokens to win = 5) Q R S T **Z** M N O **Z** P I J **Z** K L E **Z** F G H **Z** A B C D pos.getRow = 0 pos.getCol = 0 p = 'Z'	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NESW diagonal that touches the bottom left corner of the game board. The last token placed is on the end of the diagonal in the bottom left corner, so the test will ensure the function does not check out of bounds while also still checking to the right for the win. The test should return true because there are 5 'Z' tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NESW_LastPlacedOnLeft_BottomLeft_true

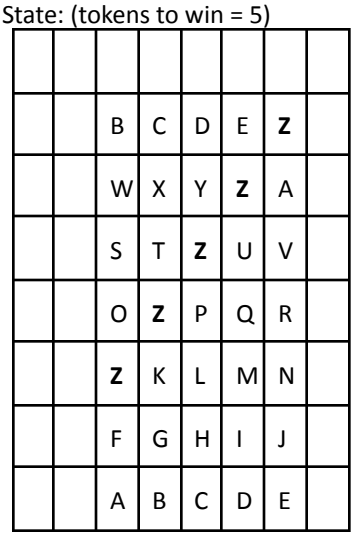
Input:	**Output:**	**Reason:**
State: (tokens to win = 5) G H I J **Z** C D E **Z** F X Y **Z** A B T **Z** U V W **Z** P Q R S K L M N O F G H I J A B C D E pos.getRow = 7 pos.getCol = 7 p = 'Z'	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NESW diagonal that touches the top right corner of the game board. The last token placed is on the end of the diagonal in the top right corner, so the test will ensure the function does not check out of bounds while also still checking to the left for the win. The test should return true because there are 5 'Z' tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NESW_LastPlacedOnRight_TopRight_true

Input:	**Output:**	**Reason:**
State: (tokens to win = 5)  pos.getRow = 2 pos.getCol = 2 p = ‘Z’	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NESW diagonal that touches the bottom left corner of the game board. The last token placed is in the middle so the test will ensure the function does not check out of bounds, but it also has to check to the left and right for the win since the last token is not on the end. The test should return true because there are 5 ‘Z’ tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NESW_LastPlacedInMiddle_BottomLeft_true

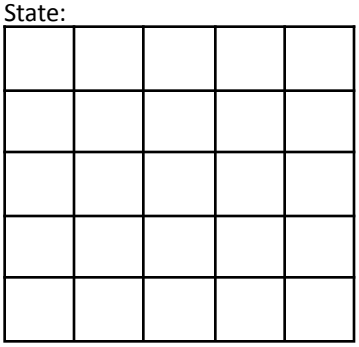
Input:	**Output:**	**Reason:**
State: (tokens to win = 5)  pos.getRow = 0 pos.getCol = 7 p = ‘Z’	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NWSE diagonal that touches the bottom right corner of the game board. The last token placed is on the end of the diagonal in the bottom right corner, so the test will ensure the function does not check out of bounds while also still checking to the right for the win. The test should return true because there are 5 ‘Z’ tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NWSE_LastPlacedOnRight_BottomRight_true

Input:	**Output:**	**Reason:**
State: (tokens to win = 5) **Z** G H I J C **Z** D E F X Y **Z** A B T U V **Z** W P Q R S **Z** K L M N O F G H I J A B C D E pos.getRow = 7 pos.getCol = 0 p = 'Z'	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NWSE diagonal that touches the top left corner of the game board. The last token placed is on the end of the diagonal in the top left corner, so the test will ensure the function does not check out of bounds while also still checking to the right for the win. The test should return true because there are 5 'Z' tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NWSE_LastPlacedOnLeft_TopLeft_true

Input:	**Output:**	**Reason:**
State: (tokens to win = 5) **Z** Q R S T M **Z** N O P I J **Z** K L E F G **Z** H A B C D **Z** pos.getRow = 2 pos.getCol = 5 p = 'Z'	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NWSE diagonal that touches the bottom right corner of the game board. The last token placed is in the middle of the diagonal, so the test will ensure the function does not check out of bounds, but it also has to check to the left and right for the win since the last token is not on the end. The test should return true because there are 5 'Z' tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NWSE_LastPlacedInMiddle_BottomRight_true

Input:	**Output:**	**Reason:**
State: (tokens to win = 5)  pos.getRow = 2 pos.getCol = 2 p = 'Z'	checkDiagWin = true The state of the board is unchanged.	This test case is unique because the diagonal is a NESW diagonal that does not touch any corner or edge of the game board. The last token placed is on the end of the diagonal on the far left. The test will ensure the function checks to the right of the passed-in position for the win and successfully finds a win for a diagonal occurring in the middle of the board. The test should return true because there are 5 'Z' tokens in a row. **Function Name:** testGameBoard_checkDiagWin_NESW_LastPlacedInRow2Col2_true

boolean checkTie()

Input:	**Output:**	**Reason:**
State: 	checkTie = false The state of the board is unchanged.	This test case is unique because the board that is checked is completely empty. The board has no tokens in it, so there is not a tie. **Function Name:** testGameBoard_checkTie_EmptyBoard_false

Input:

State:

A				
B	C			

Output:

checkTie = false

The state of the board is unchanged.

Reason:

This test case is unique because the board that is checked is not completely empty and the leftmost and lower bounds are checked to ensure the function does not go past them. The board has some tokens in it, but is not full so there is not a tie

Function Name:
testGameBoard_checkTie_PartiallyFullBoard_false

Input:

State:

T	U	V	W	
O	P	Q	R	D
J	K	L	M	C
E	F	G	H	B
A	B	C	D	A

Output:

checkTie = false

The state of the board is unchanged.

Reason:

This test case is unique because the board that is checked is almost completely full. Every coordinate has a token in it except for one, therefore there is not a tie since there is one more space for a token

Function Name:
testGameBoard_checkTie_AlmostFullBoard_false

Input:

State:

U	V	W	X	Y
P	Q	R	S	T
K	L	M	N	O
F	G	H	I	J
A	B	C	D	E

Output:

checkTie = true

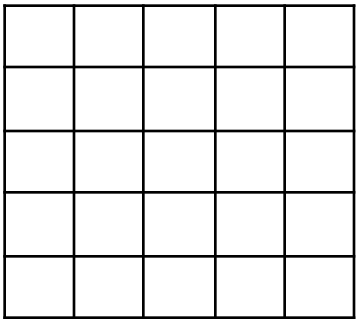
The state of the board is unchanged.

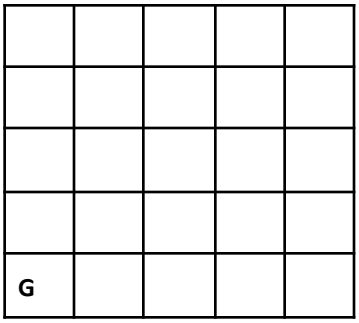
Reason:

This test case is unique because the board that is checked is completely full. Every coordinate has a token in it, therefore there is a tie since there is no more spaces for a token

Function Name:
testGameBoard_checkTie_FullBoard_true

char whatsAtPos(BoardPosition pos)

Input:	**Output:**	**Reason:**
State:  pos.getRow = 1 pos.getCol = 1	whatsAtPos = ' ' The state of the board is unchanged.	This test case is unique because the board is completely empty. Therefore, it will ensure that a random position returns a blank space character. **Function Name:** testGameBoard_whatsAtPos_emptyBoard

Input:	**Output:**	**Reason:**
State:  pos.getRow = 0 pos.getCol = 0	whatsAtPos = 'G' The state of the board is unchanged.	This test case is unique because the position that is checked has a token, located in the bottom left corner. **Function Name:** testGameBoard_whatsAtPos_singleToken_tokenInBottomLeft

Input:

State:

T	U	V	W	X
O	P	Q	R	S
K	L	**G**	M	N
F	G	H	I	J
A	B	C	D	E

pos.getRow = 2
pos.getCol = 2

Output:

whatsAtPos = 'G'

The state of the board is unchanged.

Reason:

This test case is unique because the position that is checked has a token in it and the board is full of tokens

Function Name:
testGameBoard_whatsAtPos_FullBoard_tokenInMiddle

Input:

State:

U	V	W	X	**G**
P	Q	R	S	T
K	L	M	N	O
F	G	H	I	J
A	B	C	D	E

pos.getRow = 4
pos.getCol = 4

Output:

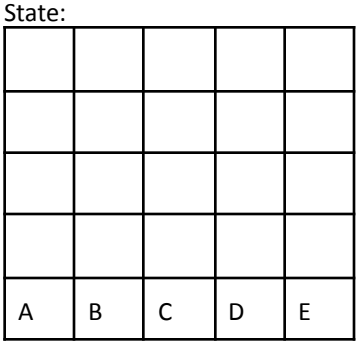
whatsAtPos = 'G'

The state of the board is unchanged.

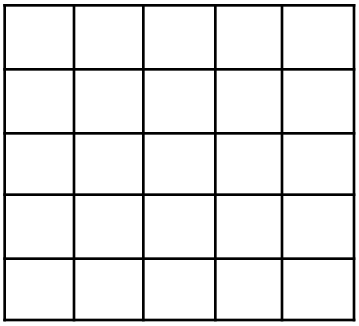
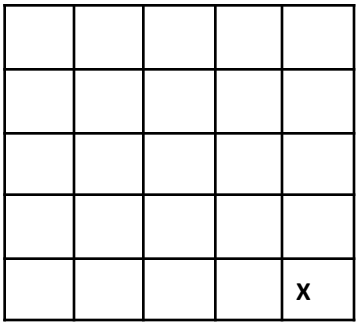
Reason:

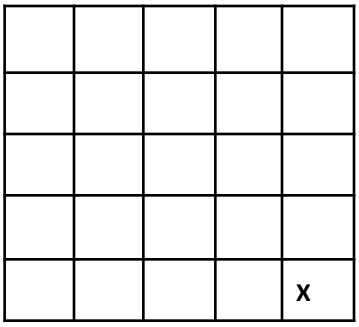
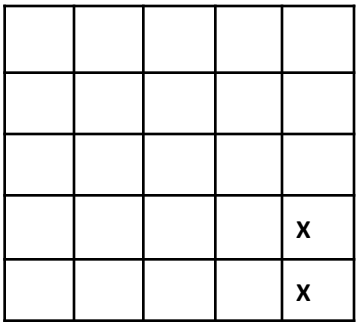
This test case is unique because the position that is checked has a token in it and the board is full of tokens. This token is on the edge of the board, however, rather than in the middle of the board.

Function Name:
testGameBoard_whatsAtPos_FullBoard_topRight

Input: State:  pos.getRow = 4 pos.getCol = 4	**Output:** whatsAtPos = ' ' The state of the board is unchanged.	**Reason:** This test case is unique because the position that is checked is empty, but there are tokens placed on the board. **Function Name:** testGameBoard_whatsAtPos_tokensPlaced_topRight_emptyPos

void dropToken(char p, int c)

Input: **gb.dropToken('X', 4);** State: 	**Output:** State:  	**Reason:** This test case is unique because it tests to see if the dropToken method dropped a token in the bottom right hand corner of an empty game board. **Function Name:** **testGameBoard_dropToken_BottomRight**

Input: **gb.dropToken('X', 4);** State: 	**Output:** State:  	**Reason:** This test case is unique because it tests to see if the dropToken method dropped a token in the correct position on top of another token. **Function Name:** **testGameBoard_dropToken_RightmostColumn_PartiallyFullColumn**

Input:
gb.dropToken('X', 4);

State:

				X
				X
				X
				X

Output:

State:

				X
				X
				X
				X
				X

Reason:

This test case is unique because it tests to see if the dropToken method will drop another token to fill the column.

Function Name:
testGameBoard_dropToken_RightmostColumn_FullColumn

Input:
gb.dropToken('X', 4);
gb.dropToken('O', 4);
gb.dropToken('X', 4);
gb.dropToken('O', 4);
gb.dropToken('X', 4);

State:

Output:

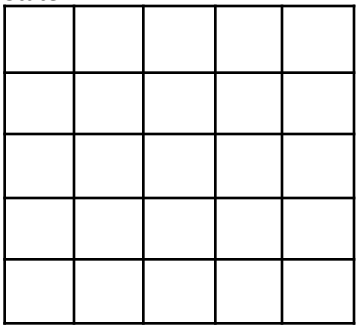
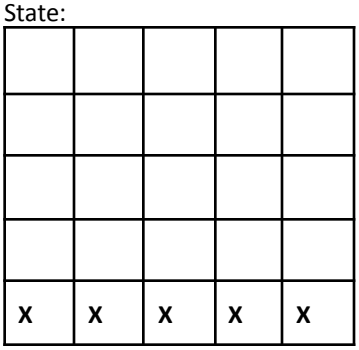
State:

		X		
		O		
		X		
		O		
		X		

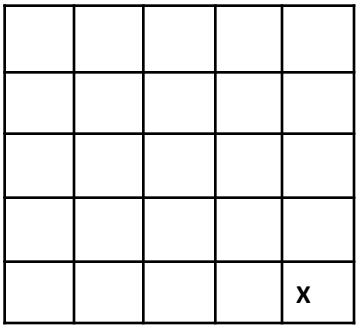
Reason:

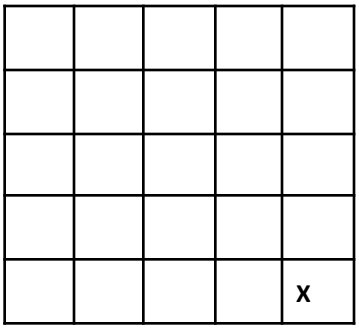
This test case is unique because it tests to see whether each token will drop correctly on another player's token.

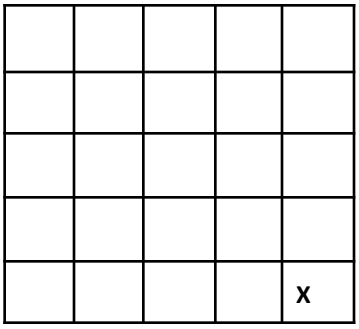
Function Name:
testGameBoard_dropToken_MiddleColumn_AlternatingTokensStacked

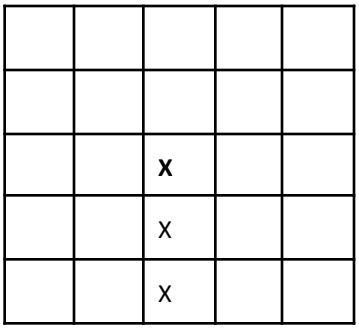
Input:	Output:	Reason:
gb.dropToken('X', 0); **gb.dropToken('X', 1);** **gb.dropToken('X', 2);** **gb.dropToken('X', 3);** **gb.dropToken('X', 4);** State: 	State:  **X X X X X**	This test case is unique because it tests to see if the dropToken method will fill an entire row correctly. **Function Name:** **testGameBoard_dropToken_FillFirstRow**

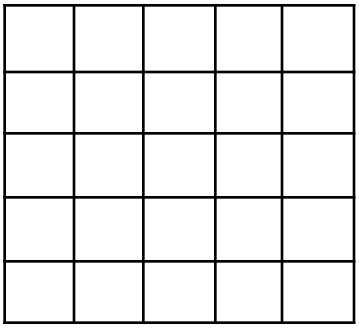
boolean isPlayerAtPos(BoardPosition pos, char player)

Input:	Output:	Reason:
pos.getRow = 0; **pos.getColumn = 4;** **gb.isPlayerAtPos(pos, 'X');** State:  **X**	isPlayerAtPos = true The state of the board is unchanged.	This test case is unique because it tests to see if isPlayerAtPos returns true for a token in the bottom right corner of the gameboard. **Function Name:** **testGameBoard_isPlayerAtPos_BottomRight_True**

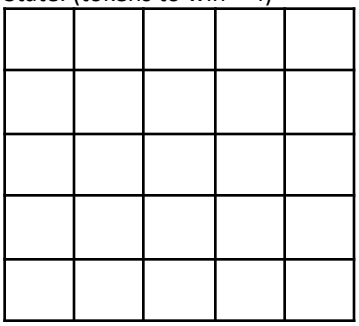
Input: **pos.getRow = 1;** **pos.getColumn = 4;** **gb.isPlayerAtPos(pos, 'X');** State:  	**Output:** isPlayerAtPos = false The state of the board is unchanged.	**Reason:** This test case is unique because it tests to see if isPlayerAtPos returns false for an empty position above a placed token. **Function Name:** **testGameBoard_isPlayerAtPos_AboveToken_BottomRight_False**

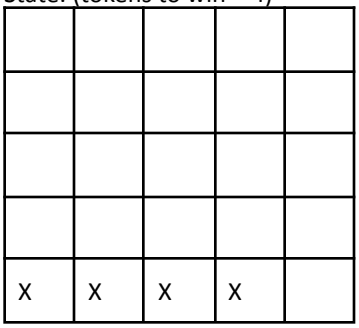
Input: **pos.getRow = 0;** **pos.getColumn = 4;** **gb.isPlayerAtPos(pos, '0');** State:  	**Output:** isPlayerAtPos = true The state of the board is unchanged.	**Reason:** This test case is unique because it tests to see if isPlayerAtPos returns false for a different player's token occupying checked the board position **Function Name:** **testGameBoard_isPlayerAtPos_BottomRight_OtherPlayer_False**

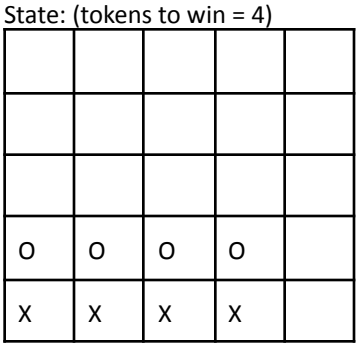
Input: **pos.getRow = 0;** **pos.getColumn = 4;** **gb.isPlayerAtPos(pos, 'X');** State: 	**Output:** isPlayerAtPos = true The state of the board is unchanged.	**Reason:** This test case is unique because it tests to see if isPlayerAtPos returns true for a token in the middle of the gameboard. **Function Name:** **testGameBoard_isPlayerAtPos_MiddleColumn_True**

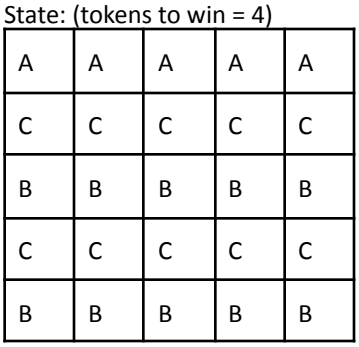
Input: **pos.getRow = 0;** **pos.getColumn = 0;** **gb.isPlayerAtPos(pos, 'X');** State: 	**Output:** isPlayerAtPos = false The state of the board is unchanged.	**Reason:** This test case is unique because it tests to see if isPlayerAtPos returns false for a position on an empty game board. **Function Name:** **testGameBoard_isPlayerAtPos_EmptyBoard_False**

boolean checkHorizWin(BoardPosition pos, char p)

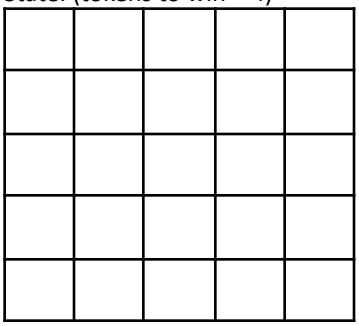
Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 0 pos.getCol = 4 p = 'X'	checkHorizWin = false The state of the board is unchanged.	This test is unique because it is testing on an empty game board and it should return false. **Function Name:** testGameBoard_checkHorizWin_EmptyBoard_BottomRight_False

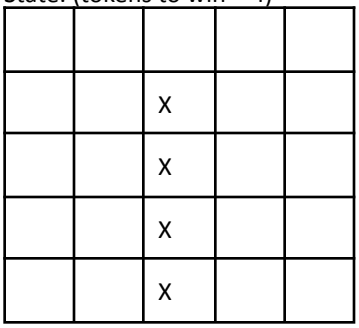
Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 0 pos.getCol = 0 p = 'X'	checkHorizWin = true The state of the board is unchanged.	This test is testing that checkHorizWin returns true for a horizontal win on the bottom row of the game board **Function Name:** testGameBoard_checkHorizWin_LastPlacedOnLeft_BottomLeft_True

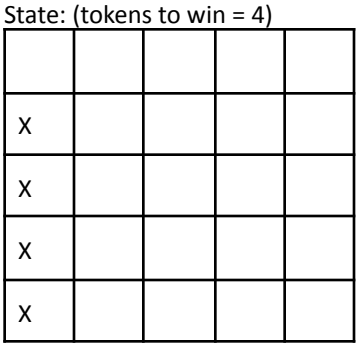
Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 0 pos.getCol = 4 p = '0'	checkHorizWin = true The state of the board is unchanged.	This test simulates a regular game and a player gets a win on the second row. checkHorizWin should return true **Function Name:** testGameBoard_checkHorizWin_LastPlacedTokenRow1Col3_True

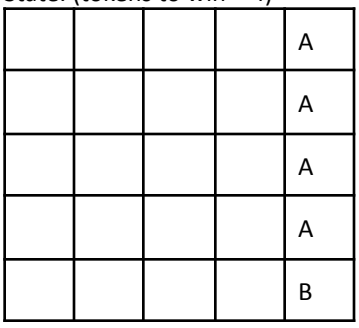
Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 4 pos.getCol = 0 p = 'A'	checkHorizWin = true The state of the board is unchanged.	This test fills the entire board with tokens, and a player gets a win in the top left corner of the board. The last placed token is A and checkHorizWin should return true. **Function Name:** testGameBoard_checkHorizWin_LastPlacedOnLeft_TopLeft_True

boolean checkVertWin(BoardPosition pos, char p)

Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 0 pos.getCol = 4 p = 'X'	checkVertWin = false The state of the board is unchanged.	This test is unique because it is testing on an empty game board. **Function Name:** testGameBoard_checkVertWin_EmptyBoard_BottomRight_False

Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 3 pos.getCol = 2 p = 'X'	checkVertWin = true The state of the board is unchanged.	This test fills the middle column with tokens, creating a win so checkVertWin should return true. **Function Name:** testGameBoard_checkVertWin_MiddleColumn_True

Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 3 pos.getCol = 0 p = 'X'	checkVertWin = true The state of the board is unchanged.	This test tests a vertical win on the far left corner of the game board, a boundary of the board. **Function Name:** testGameBoard_checkVertWin_FarLeftColumn_True

Input:	**Output:**	**Reason:**
State: (tokens to win = 4)  pos.getRow = 4 pos.getCol = 4 p = 'A'	checkVertWin = true The state of the board is unchanged.	This test tests a vertical win on the far right corner of the game board, a boundary of the board, with another player's token below the vertical win. **Function Name:** testGameBoard_checkVertWin_FarRightColumn_OpponentBelow_True

Deployment

This project uses a makefile to compile and run the program. To compile the program, type the command “make” into the command line within the game’s top directory and press enter. To run the program, type the command “make run” into the command line and press enter. The makefile also allows for easy testing of the program. To compile the test cases, type the command “make test” into the command line and press enter. To test, you must choose a specific implementation to test. To test the 2D array implementation of the program, type “testGB” into the command line after compiling the test cases. To test the hashmap implementation, type “testGBMem” into the command line after compiling the test cases. Typing the command “make clean” into the command line will clean out your directory of any compiled class files.