CPSC 2150 Project 4 Report

Rowan Froeschner, Graham Frazier, Terance Harrison, Kalyaan Narnamalpuram

Requirements Analysis

Functional Requirements:

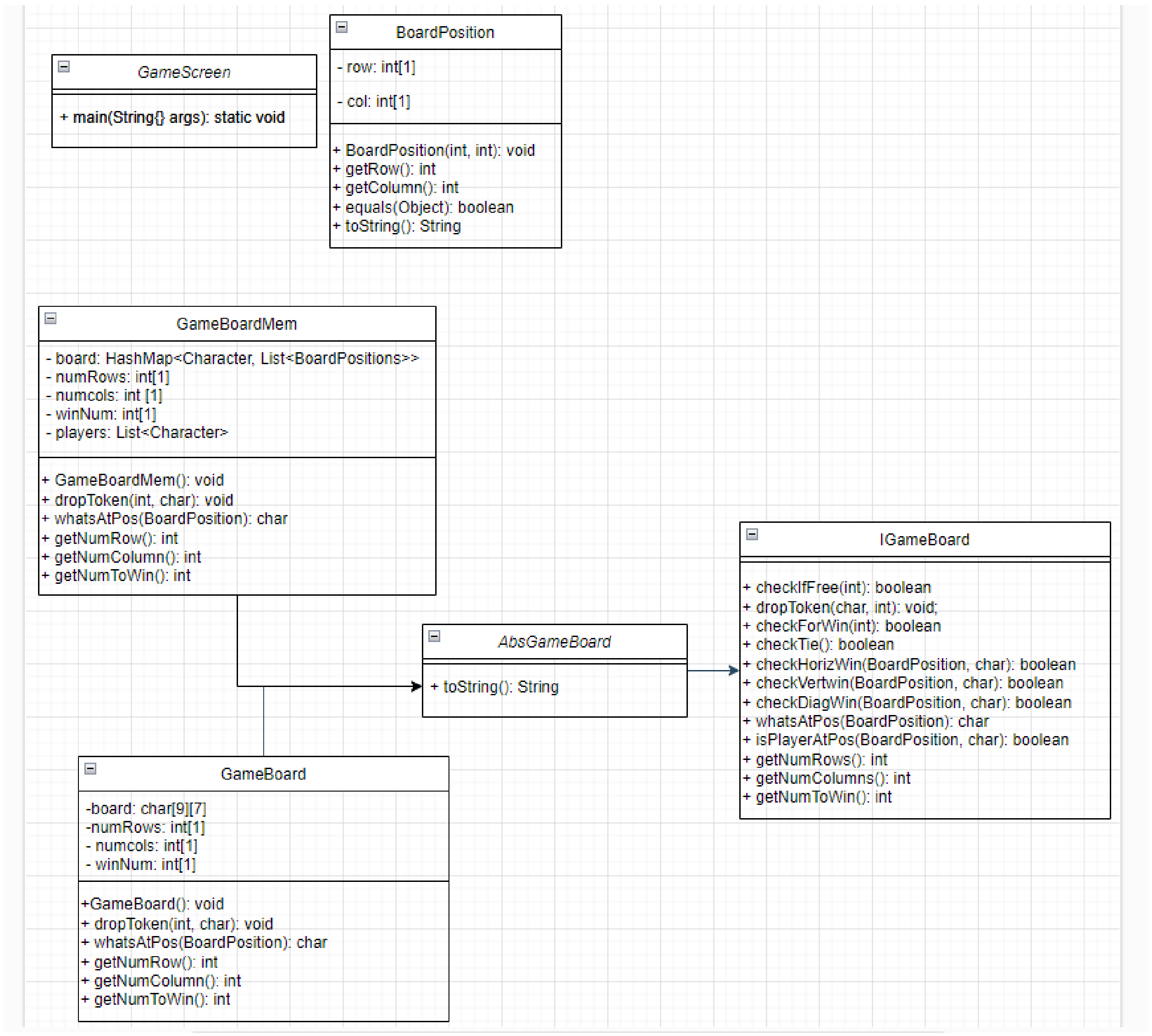
1. As a player, I need the game to let me know if I picked a nonexistent or full column and allow me to try again.
2. As a player, I need the tokens to drop to the most bottom available row after they select their column to imitate how gravity affects the game in the real world.
3. As a player, I need to alternate turns on the board between the players because the point of the game is to alternate turns and see who can win.
4. As a player, I need each slot to have its own “space” so that tokens can interact with each other
5. As a player, I need blank characters to represent each empty slot in the board so the space can be recognized as unoccupied.
6. As a player, I need a function that runs after every turn to see if a winner can be declared due to however many vertical tokens in a row I chose
7. As a player, I need a function that runs after every turn to see if a winner can be declared due to however many horizontal tokens in a row I chose
8. As a player, I need a function that runs after every turn to see if a winner can be declared due to however many diagonal tokens in a row I chose
9. As a player, I need each column to be checked so I can’t place a token in a full column.
10. As a player, I need a message that the game is a draw if the board completely fills up and there are no more available spaces.
11. As a player, I need the columns labeled so that I know which column is which while I am playing the game.
12. As a player, I need a way to input or select the column I want to put the marker/token on the board so that I can place my token where i want it.
13. As a player, I need an option after the game ends to play again, so I do not need to restart the program every time I want to play again.
14. As a player, I should be able to clearly see the board so i can track the place/location of game pieces within the game.
15. As a player, it needs to be a simple board, so the game is accessible to all ages.
16. As a player, I need to know when the column is full, in case I accidentally try to place a marker in a full column.

17. As a player, I need the game to say who won before a new round starts so we can confirm the score.
18. As a player i need to be able to choose how many rows on the board so the board can be built for me to play on
19. As a player i need to be able to choose how many columns on the board so the board can be built for me to play on
20. As a player i need to be able to choose how many tokens in a row on the board win so the board can be built for me to play on
21. As a player i need to be able to choose how many people are playing so that I can include everyone.

Non-Functional Requirements

1. There needs to be a board that is resizable from 100 x 100 to 3 x 3 using lines to give the player a visual of the game board.
2. The numToWin must be resizable from 3 to 25
3. The game must always begin with player 1 going first.
4. There should be messages that pop up indicating which player's turn it is or if they need to redo their turn for playing an illegal move to help guide the game more directly.
5. The code must be in java.
6. The program must run on unix.
7. The boards bottom left corner should always begin at 0,0 to not create problems and keep uniformity.

System Design


GameScreen
+ main(String{} args): static void
BoardPosition
- row: int[1]
- col: int[1]
+ BoardPosition(int, int): void
+ getRow(): int
+ getColumn(): int
+ equals(Object): boolean
+ toString(): String
GameBoardMem
- board: HashMap<Character, List<BoardPositions>>
- numRows: int[1]
- numcols: int [1]
- winNum: int[1]
- players: List<Character>
+ GameBoardMem(): void
+ dropToken(int, char): void
+ whatsAtPos(BoardPosition): char
+ getNumRow(): int
+ getNumColumn(): int
+ getNumToWin(): int
IGameBoard
+ checkIfFree(int): boolean
+ dropToken(char, int): void;
+ checkForWin(int): boolean
+ checkTie(): boolean
+ checkHorizWin(BoardPosition, char): boolean
+ checkVertwin(BoardPosition, char): boolean
+ checkDiagWin(BoardPosition, char): boolean
+ whatsAtPos(BoardPosition): char
+ isPlayerAtPos(BoardPosition, char): boolean
+ getNumRows(): int
+ getNumColumns(): int
+ getNumToWin(): int
AbsGameBoard
+ toString(): String
GameBoard
-board: char[9][7]
-numRows: int[1]
- numcols: int[1]
- winNum: int[1]
+GameBoard(): void
+ dropToken(int, char): void
+ whatsAtPos(BoardPosition): char
+ getNumRow(): int
+ getNumColumn(): int
+ getNumToWin(): int

Testing

Input:

State:

```
| 0| 1| 2|
|  |  |  |
|  |  |  |
|  |  |  |
```

small = 3

Output:

gb.getNumRows() = 3
gb.getNumColumns() = 3
gb.getNumToWin() = 3

State of the object stays the same.

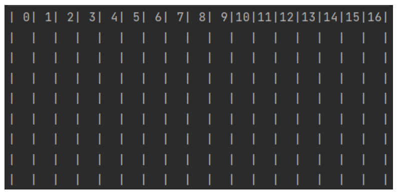
Reason:

This is the most simple constructor test with minimum values, just to make sure the process of creating the object works.

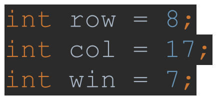
Function Name:
testConstructorMinimum()

Input:

State:



```
| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```



```
int row = 8;
int col = 17;
int win = 7;
```

Output:

gb.getNumRows() = row
gb.getNumColumns() = col
gb.getNumToWin() = win

State of the object stays the same.

Reason:

This is a middle case that tests deeper boundaries of the object construction.

Function Name:
testConstructorMedium()

Input:

State:

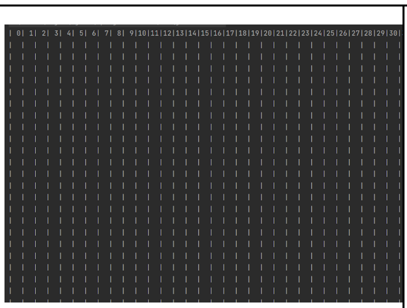
Output:

gb.getNumRows() = row
gb.getNumColumns() = col
gb.getNumToWin() = win

State of the object stays the same.

Reason:

This is a boundary case, because it tests the creation of the biggest possible board allowed by the program, making sure it is capable of reaching its limit.



```
| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

(not fully visible because its so huge)

```
int row = 100;
int col = 100;
int win = 25;
```

Function Name:
testConstructorMaximum()

Input:

State:

```
| 0| 1| 2|
|  |  |  |
|  |  |  |
|  |  |  |
```

```
char p1 = 'X';
char p2 = 'O';
char empty = ' ';
int col1 = 0;
int col2 = 1;
int col3 = 2;
int row = 0;
int row2 = 1;
```

Output:

```
p1, gb.whatsAtPos(new
BoardPosition(row,
col1))
p2 = gb.whatsAtPos(new
BoardPosition(row,
col2))
```

```
| 0| 1| 2|
|  |  |  |
|  |  |  |
|X| O|   |
```

Reason:

This is a lower case that should be most likely to pass because there are only two drops and just tests that basic inputs work.

Function Name:
`testDropToken2Horizontal()`

Input:

State:

Output:

```
p1 = gb.whatsAtPos(new
BoardPosition(row,
```

Reason:
This is a slightly harder case involving stacking this time, so making sure it does this

| 0| 1| 2|
| | | |
| | | |
| | | |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
`int row2 = 1;`
	`col1))`
	
	`p2 = gb.whatsAtPos(new BoardPosition(row, col2))`
	
	`p1 = gb.whatsAtPos(new BoardPosition(row2, col1))`
	
	| 0| 1| 2|
	| | | |
	|X| | |
	|X| O| |
		correctly.
		
		Function Name:
		`testDropToken2InAColumn()`

Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
| | | |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
`int row2 = 1;`
	Output:
	`p1 = gb.whatsAtPos(new BoardPosition(row, col1)`
	`p1 = gb.whatsAtPos(new BoardPosition(row, col2))`
	`p1 = gb.whatsAtPos(new BoardPosition(row2, col3))`
	
	| 0| 1| 2|
	| | | |
	| | | |
	|X| X| X|
		Reason:
		This tests to make sure a whole row can get filled with no conflict producing from running out of columns unoccupied.
		
		Function Name:
		`testDropTokenFillRow()`

Input:
 	
State:
 	
| 0| 1| 2|
	Output:
	`p1 = gb.whatsAtPos(new BoardPosition(row1, col1))`
	`p1 = gb.whatsAtPos(new BoardPosition(row2,`
		Reason:
		This tests full verticals reaching the top of a column and not having any issues being able to reach the top.

| | | |
| | | |
| | | |
 	
`char p1 = 'X';`
`int col1 = 0;`
`int row1 = 0;`
`int row2 = 1;`
`int row3 = 2;`
	`col1))`
	`p1 = gb.whatsAtPos(new BoardPosition(row3, col1))`
	
	| 0| 1| 2|
	|X| | |
	|X| | |
	|X| | |
		Function Name:
		`testDropTokenFillColumn()`

Input:	**Output:**	**Reason:**

State:
 	
| 0| 1| 2|
| | | |
| | | |
| | | |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char p3 = 'A';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row1 = 0;`
`int row2 = 1;`
`int row3 = 2;`
	`p1 = gb.whatsAtPos(new BoardPosition(row1, col1))`
	`p1 = gb.whatsAtPos(new BoardPosition(row2, col1))`
	`p1 = gb.whatsAtPos(new BoardPosition(row3, col1))`
	`p2 = gb.whatsAtPos(new BoardPosition(row1, col2))`
	`p2 = gb.whatsAtPos(new BoardPosition(row2, col2))`
	`p2 = gb.whatsAtPos(new BoardPosition(row3, col2))`
	`p3 = gb.whatsAtPos(new BoardPosition(row1, col3))`
	`p3 = gb.whatsAtPos(new BoardPosition(row2, col3))`
	`p3 = gb.whatsAtPos(new BoardPosition(row3, col3))`
	
	| 0| 1| 2|
	|X| O|A |
	|X|O | A|
	|X| O| A|
		This is the hardest test case as it tests the ability of the program place into every single position possible on the board with more than 2 characters also.
		
		Function Name:
		`testDropTokenFillBoard()`

Input:

State:

```
| 0| 1| 2|
|  |  |  |
|  |  |  |
| X|  |  |
```

```
char p1 = 'X';
int col = 0;
```

Output:

```
p1 = gb.whatsAtPos(new
BoardPosition(0, col))
```

State of object stays the same.

Reason:
This is the simplest case, all it has is 1 position where it drops a token and it makes sure the function is able to find that its there.

Function Name:
```
testWhatsAtPosSingleToke
n()
```

Input:

State:

```
| 0| 1| 2|
|  |  |  |
|  |  |  |
|X |O|   |
```

```
char p1 = 'X';
char p2 = 'O';
int col1 = 0;
int col2 = 1;
int row = 0;
```

Output:

```
p1 = gb.whatsAtPos(new
BoardPosition(row,
col1))
p2 = gb.whatsAtPos(new
BoardPosition(row,
col2))
```

State of object stays the same.

Reason:
Slightly harder case check two distinct characters at different locations making sure both were placed.

Function Name:
```
testWhatsAtPosMultipleTo
kens()
```

Input:

State:

```
| 0| 1| 2|
|  |  |  |
```

Output:

```
p1 = gb.whatsAtPos(new
BoardPosition(row,
col1))
p2 = gb.whatsAtPos(new
BoardPosition(row,
```

Reason:
Very similar to the last test, but tests boundary cases a bit better with far left and right column, also making sure the center is empty to test all cases.

| | | |
| X| |O|
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
	`col3))`
	`empty = gb.whatsAtPos(new BoardPosition(row, col2))`
	
	State of object stays the same.
		Function Name:
		`testWhatsAtPosEmptySpaceBetweenTokens()`

Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
|X |O| |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`int col1 = 0;`
`int col2 = 1;`
`int row = 0;`
	Output:
	
	`p1 = gb.whatsAtPos(new BoardPosition(row, col1))`
	`p2 = gb.whatsAtPos(new BoardPosition(row, col2))`
	`' ' = gb.whatsAtPos(new BoardPosition(row, col3))`
	
	State of object stays the same.
		Reason:
		This is basically the same as the second one fro whats at pos, but checks other spaces around the board better.
		
		Function Name:
		`testWhatsAtPosEmptySpaceAfterTokens()`

Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
| X| | |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
	Output:
	
	`gb.isPlayerAtPos(new BoardPosition(row, col1) = p1`
	`gb.isPlayerAtPos(new BoardPosition(row, col1) != p2`
	
	State of object stays the
		Reason:
		Simple check to make sure that it was an X and that the one placed in 0,0 and not 0, so acts as the light boundary case.
		
		Function Name:
		`testIsPlayerAtPosSingleToken()`

`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
	same.	

State:
 	
| 0| 1| 2|
| | | |
| | | |
| X| | |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
`int Lrow =`
`gb.getNumRows() - 1;`
	Output:
	
	`gb.isPlayerAtPos(new BoardPosition(row, col1) = p1`
	`gb.isPlayerAtPos(new BoardPosition(col3, Lrow) != p1`
	
	State of object stays the same.
		Reason:
		This one is making sure the player is not at an empty space this time instead of being another character.
		
		Function Name:
		`testIsPlayerAtPosEmptySpace()`

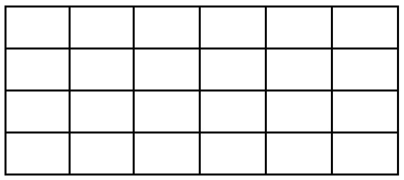
Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
| X|O| |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
	Output:
	
	`gb.isPlayerAtPos(new BoardPosition(row, col1) = p1`
	`gb.isPlayerAtPos(new BoardPosition(row, col2) = p2`
	`gb.isPlayerAtPos(new BoardPosition(row, col1) != p2`
	
	State of object stays the same.
		Reason:
		This function makes sure that multiple positions are working and it accurately reads at all of these different cases.
		
		Function Name:
		`testIsPlayerAtPosMultiplePlayers()`

`int Lrow =`
`gb.getNumRows() - 1;`		

Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
| X|O| |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
`int Lrow =`
`gb.getNumRows() - 1;`
	Output:
	
	`gb.isPlayerAtPos(new`
	`BoardPosition(row, col1)`
	`= p1`
	`gb.isPlayerAtPos(new`
	`BoardPosition(row, col2)`
	`= p2`
	`gb.isPlayerAtPos(new`
	`BoardPosition(row2,`
	`col2) != p1`
	
	State of object stays the same.
		Reason:
		This one is very similar to the last one but checks more boundary cases to make sure they are empty.
		
		Function Name:
		`testIsPlayerAtPosOverlap`
		`pingPositions()`

Input:
 	
State:
 	
| 0| 1| 2|
| | | |
| | | |
| X|O| |
 	
`char p1 = 'X';`
`char p2 = 'O';`
`char empty = ' ';`
`int col1 = 0;`
`int col2 = 1;`
`int col3 = 2;`
`int row = 0;`
	Output:
	
	`gb.isPlayerAtPos(new`
	`BoardPosition(row, col1)`
	`= p1`
	`gb.isPlayerAtPos(new`
	`BoardPosition(row, col2)`
	`= p2`
	`gb.isPlayerAtPos(new`
	`BoardPosition(row2,`
	`col1) != p2`
	`gb.isPlayerAtPos(new`
	`BoardPosition(row2,`
	`col2) != p1`
		Reason:
		This one is also very similar, but uses more assert falses to make sure its able to correctly distinguish the character
		
		Function Name:
		`testIsPlayerAtPosNonOver`
		`lappingPositions()`

`int Lrow = gb.getNumRows() - 1;`	State of object stays the same.	

Input: State:  Initial empty board	**Output: assertTrue(gb.checkIfFree(col)):** Expected to be true for any column (col), as the board is empty.	**Reason:** The test is checking if the method correctly identifies that all cols are free on an empty board. *Note*: since the board is empty, all columns should be free. **Function Name:** `testCheckIfFreeEmptyBoard`

Input:
State:

X					

Output: assertTrue(gb.checkIfFree(col)): Expected to be true for columns other than the one where the token was dropped (col).

Reason: The test checks if the method correctly identifies that a column is free if it's not the column where a token has been dropped.

Function Name:

`testCheckIfFreeColumnPartiallyFull`

Input:
State:

X					
X					
X					

Output:
assertFalse(gb.checkIfFree(col1)): Column 0 is not free.
assertTrue(gb.checkIfFree(col2)): Column 1 is free.

Reason:
The test checks if the method correctly identifies that a column is not free if it is full.
Function Name:

`testCheckIfFreeColumnFull`

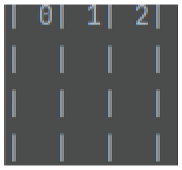
Input:
State:

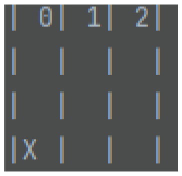
X	X	X			
x	X	X			
X	X	X			

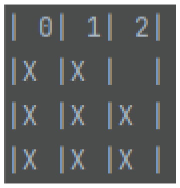
Output:
assertFalse(gb.checkIfFree(col1)): Column 0 is not free.
assertFalse(gb.checkIfFree(col2)): Column 1 is not free.

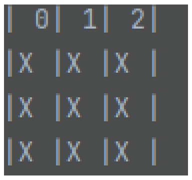
Reason:
The test checks if the method correctly identifies that all cols are not free when the entire board is full.
Function Name:

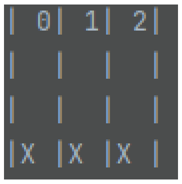
X | X | X | | |	**assertFalse(gb.checkIfFree(col3)):** Column 2 is not free.	`testCheckIfFreeBoardFull`

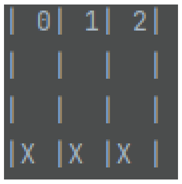
Input:
 	
State:

`| 0| 1| 2|`
`|  |  |  |`
`|  |  |  |`
`|  |  |  |`
	Output:
	checkTie = false
	State of the board stays the same
		Reason:
		Checks that check tie does not give true on an empty board
		Function Name:
		`testCheckTieEmptyBoard`

Input:
 	
State:
 	

`| 0| 1| 2|`
`|  |  |  |`
`|  |  |  |`
`|X |  |  |`
	Output:
	checkTie = false
	State of the board stays the same
		Reason:
		Checks that a dropped token does not cause a false positive
		Function Name:
		`testCheckTiePartiallyFullBoard`

Input:
 	
State:

`| 0| 1| 2|`
`|X |X |  |`
`|X |X |X |`
`|X |X |X |`
	Output:
	checkTie = false
	State of the board stays the same
		Reason:
		Checks that we don't get a false positive with only one spot left open
		Function Name:
		`testCheckTieAlmostFullBoard`

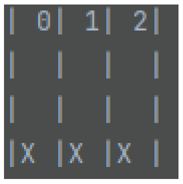
Input: State:  | 0| 1| 2| |X |X |X | |X |X |X | |X |X |X |	**Output:** checkTie = true State of the board stays the same	**Reason:** Checks that the function recognizes a full board as a tie **Function Name:** `testCheckTieFullBoard`

Input: State:  | 0| 1| 2| | | | | | | | | |X |X |X |	**Output:** checkHorizWin = true State of the board is unchanged	**Reason:** Checks if the function works from the left to the right **Function Name:** `testHorizontalWinLeftToRight`

Input: State:  | 0| 1| 2| | | | | | | | | |X |X |X |	**Output:** checkHorizWin = true State of the board is unchanged	**Reason:** Checks if the function works from the right to the left **Function Name:** `testHorizontalWinRightToLeft`

Input:

State:



```
| 0| 1| 2|
|  |  |  |
|  |  |  |
|X |X |X |
```

Output:
checkHorizWin = true

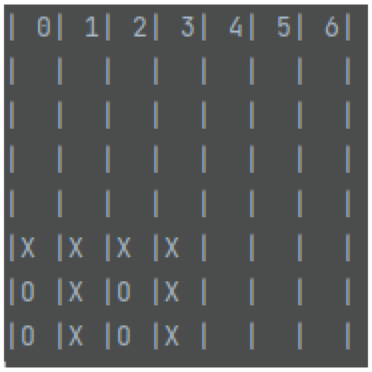
State of the board is unchanged

Reason:
Checks from the middle position

Function Name:
`testHorizontalWinMiddle`

Input:

State:



```
| 0| 1| 2| 3| 4| 5| 6|
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|X |X |X |X |  |  |  |
|O |X |O |X |  |  |  |
|O |X |O |X |  |  |  |
```

Output:
checkHorizWin = true

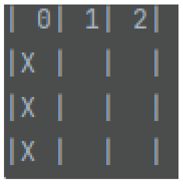
State of the board is unchanged

Reason:
Checks bigger board and number to win

Function Name:
`testHorizontalWinMedium`

Input:

State:



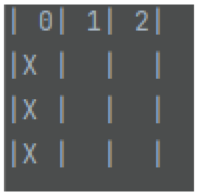
```
| 0| 1| 2|
|X |  |  |
|X |  |  |
|X |  |  |
```

Output:
checkVertWin = true

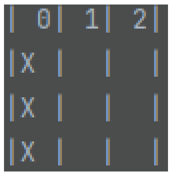
State of the board is unchanged

Reason:
Checks from top position to bottom position

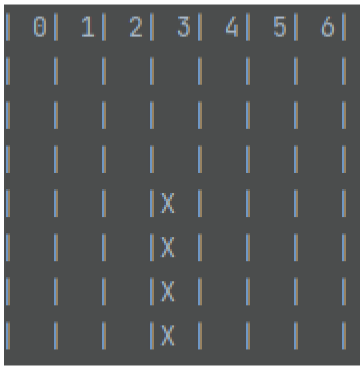
Function Name:
`testVerticalWinBottomToTop`

Input:	**Output:**	**Reason:**
State: 	checkVertWin = true State of the board is unchanged	Checks from top position to bottom position **Function Name:** `testVerticalWinTopToBottom`

```
| 0| 1| 2|
|X |  |  |
|X |  |  |
|X |  |  |
```

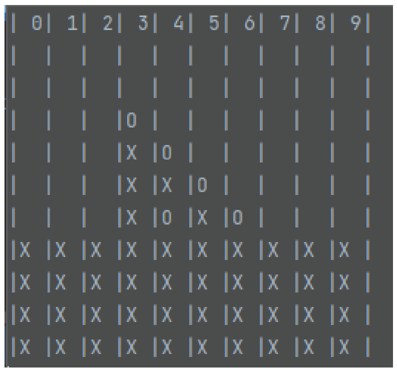
Input:	**Output:**	**Reason:**
State: 	checkVertWin = true State of the board is unchanged	Checks from middle position **Function Name:** `testVerticalWinFromMiddle`

```
| 0| 1| 2|
|X |  |  |
|X |  |  |
|X |  |  |
```

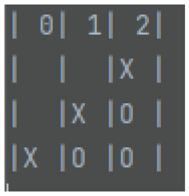
Input:	**Output:**	**Reason:**
State: 	checkVertWin = true State of the board is unchanged	Checks with bigger board and win size **Function Name:** `testVerticalWinDifferentWinSize`

```
| 0| 1| 2| 3| 4| 5| 6|
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |X |  |  |  |
|  |  |  |X |  |  |  |
|  |  |  |X |  |  |  |
|  |  |  |X |  |  |  |
```

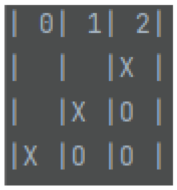
Input:	**OutPut:**	**Reason:**

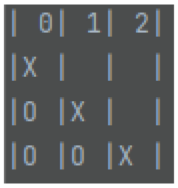
State:

`| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|`
`|   |   |   |   |   |   |   |   |   |   |`
`|   |   |   |   |   |   |   |   |   |   |`
`|   |   |   |O  |   |   |   |   |   |   |`
`|   |   |   |X  |O  |   |   |   |   |   |`
`|   |   |   |X  |X  |O  |   |   |   |   |`
`|   |   |   |X  |O  |X  |O  |   |   |   |`
`|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  |X  |X  |X  |X  |X  |`
`|X  |X  |X  |X  |X  |X  |X  |X  |X  |X  |`
	checkDiagWin = true
	
	State of the board is unchanged
		Checks with a board that has had multiple inputs in all columns
		Function Name:
		`testDiagonalWinMedium1`

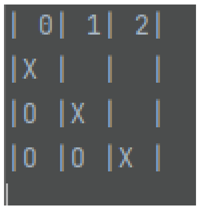
Input:	**OutPut:**	**Reason:**

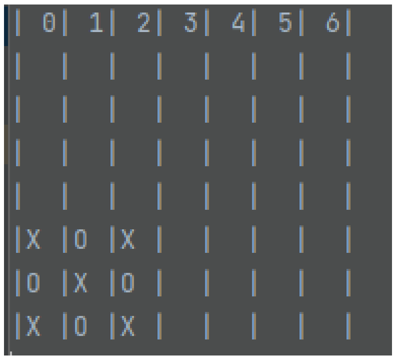
State:

`| 0| 1| 2|`
`|   |   |X  |`
`|   |X  |O  |`
`|X  |O  |O  |`
	checkDiagWin = true
	
	State of the board is unchanged
		Checks if function works from lower left to upper right
		Function Name:
		`testDiagonalWinBottomLeftToUpperRight`

Input:	**OutPut:**	**Reason:**

State:

`| 0| 1| 2|`
`|   |   |X  |`
`|   |X  |O  |`
`|X  |O  |O  |`
	checkDiagWin = true
	
	State of the board is unchanged
		Checks if the function works from the upper right to the lower left
		Function Name:
		`testDiagonalWinUpperRightToBottomLeft`

Input:
 	
State:
 	

`| 0| 1| 2|`
`|X |  |  |`
`|O |X |  |`
`|O |O |X |`
	OutPut:
	checkDiagWin = true
	
	State of the board is unchanged
		Reason:
		Checks if function works from lower Right to upper left
		Function Name:
		`testDiagonalWinlowerRightToUpperLeft`

Input:
 	
State:
 	

`| 0| 1| 2|`
`|X |  |  |`
`|O |X |  |`
`|O |O |X |`
	OutPut:
	checkDiagWin = true
	
	State of the board is unchanged
		Reason:
		Checks if it works from upperLeft to LowerRight
		Function Name:
		`testDiagonalWinUpperLeftToLowerRight`

Input:
 	
State:

`| 0| 1| 2| 3| 4| 5| 6|`
`|  |  |  |  |  |  |  |`
`|  |  |  |  |  |  |  |`
`|  |  |  |  |  |  |  |`
`|  |  |  |  |  |  |  |`
`|X |O |X |  |  |  |  |`
`|O |X |O |  |  |  |  |`
`|X |O |X |  |  |  |  |`
	OutPut:
	checkDiagWin = false
	
	State of the board is unchanged
		Reason:
		Checks if we get a false positive when it is one token away from a win
		Function Name:
		`testDiagonalWinFalse`

Input:

State:

```
| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | O|
|  |  |  |  |  |  |  |  | O| X|
|  |  |  |  |  |  |  | O| X| X|
|  |  |  |  |  |  | O| X| X| X|
```

OutPut:

checkDiagWin = true

State of the board is unchanged

Reason:

Checks with a bigger board and a bigger win num and check against the right side of the board

Function Name:

testDiagonalWinMedium2