Ejhazz Milford

Kevin Plis

CPSC 2150

17 June 2020

Project IV: Adding a GUI

Requirements Analysis

- TicTacToeGame.java
    - Functional Requirements
        - The program alternates control between two players: *As a user, I can alternate turns between the other player, so I can continue the game.*
        - The program informs whose turn it is: *As a user, I can see whose turn it is, so that player can proceed and use their token accordingly.*
        - The player selects the column they would like: *As a user, I can select a column to place my token in, so I can continue the game and progress towards a win or tie.*
        - *As a user, I can select between two and ten players.*
        - *As a user, I can set how many rows the game board has, so I have more control.*
        - *As a user, I can set how many columns the game board has, so I have more control.*
        - *As a user, I can select a fast implementation, so the program can run fast.*
        - *As a user, I can select a memory-efficient implementation, so the program allocates minimal memory.*
        - The player places a marker in their selected column: *As a user, I can place a token in a selected column, so I can continue the game and progress towards a win or tie.*
        - The program prints an error message when space is unavailable and asks the player to select a new column: *As a user, I can see when a column is full, so I can reselect a column that is not full.*
        - The program checks to see if a player has won: As a user, I can know when a player has won, so I can quit the game or start a new one.
        - The program prints a congratulatory message upon victory and prompts the player(s) to play again: *As a user, I can see when a player has won, so I can quit the game or start a new one.*
        - The player(s) choose whether to quit or play again: *As a user, I can quit the game or start a new one.*

        - The program checks to see if the game has ended in a tie if a player has not won: *As a user, I can know if the game has resulted in a win or tie, so I can quit the game or start a new one.*
        - The program prints a message indicating that the game has ended in a tie if no player has won and prompt the player(s) to play again: *As a user, I can see a message indicating that the game has ended in a tie if no player has won, so I can quit the game or start a new one.*
        - The program prints the current board and prompts the next player for their move if the game has not resulted in a win or tie: *As a user, I can see a visual representation of the board, and know when it is my turn, so I can continue the game.*
    - Non-Functional Requirements
        - User interaction will occur through a GUI to promote a fluid experience.
        - The program must be written in Java.
        - The user will interact with an interface.
- BoardPosition.java
    - Functional Requirements
        - The program tracks an individual cell for a board: *As a user, I can select an individual cell, so I can make an informed choice with my token.*
        - The program checks if two cells are equal: *As a user, I can know if a cell of interest is a valid cell, so I can avoid placing a token outside of the board.*
        - The program prints the row and column of an individual cell: *As a user, I can see the row and column of a cell, so I can make an informed choice with my token.*
    - Non-Functional Requirements
        - The player(s) cannot go out of the range of the board.
        - The code must be written in Java
- GameBoard.java
    - Functional Requirements
        - The program checks if a column is full: *As a user, I can know if a column is full, so I can reselect a column that has space available.*
        - The program checks if the last token placed resulted in a win: *As a user, I can know if the last token placed resulted in a win, so I can quit the game or start a new one.*
        - The player places a token in a coordinate: *As a user, I can place a token in a selected coordinated, so I can continue the game.*
        - The program checks if the last token placed resulted in having specified number of tokens in a row horizontally: *As a user, I can know if the game has resulted in a win having five in a row horizontally, so I can quit the game or start a new one.*

- The program checks if the last token placed resulted in having a specified number of tokens in a row vertically: *As a user, I can know if the game has resulted in a win having five in a row vertically, so I can quit the game or start a new one.*
- The program checks if the last token placed resulted in having specified number of tokens in a row diagonally: *As a user, I can know if the game has resulted in a win having five in a row diagonally, so I can quit the game or start a new one.*
- The program checks for the character that is in a position of the game board: *As a user, I can know if a cell of interest is populated, so I can make an informed choice with my token.*
- The program checks whether the player(s) are at the position: *As a user, I can know which player token is in a populated space, so I can make an informed choice with my token or know when five in a row have been achieved.*
- The program creates a visual representation of the current gameboard: *As a user, I can create a visual representation of the current gameboard, so I can make an informed choice with my token.*
- The program checks if the gameboard results in a tie: *As a user, I can know if the last token placed resulted in a tie, so I can quit the game or start a new one.*

- Non-Functional Requirements
  - The code must be written in Java
  - The player(s) cannot place a token in a populated cell.
  - The player(s) cannot place a token in a column that is full.
  - The player(s) cannot place a token anywhere on the board if the last token placed resulted in a win or tie.

Test Plan

Test Case	Input	Output	Reason
public void testGameBoard_row50_col50_win10()	row = 50, column = 50, numToWin = 10	A game board with 50 rows, 50 columns, and a condition where 10 tokens in a row are needed to win.	**Routine Case**
public void testGameBoard_row3_col3_win3()	row = 3, column = 3, numToWin = 3	A game board with three rows, three columns, and a condition where three tokens in a row are needed to win.	**Boundary Case**: Test if the program can build a game board with the minimum number of rows, columns, and wins.
public void testGameBoard_row100_col100_win25()	row = 100, column = 100 numToWin = 25	A game board with 100 rows, 100 columns, and a condition where 25 tokens in a row are needed to win.	**Boundary Case**: Test if the program can build a game board with the maximum number of rows, columns, and wins.
public void		A game board with three	**Challenging Case**: Test

testGameBoard_row3_col100_win3()		rows, 100 columns, and a condition where three tokens in a row are needed to win.	if the program can build a game board with an asymmetric number of rows, columns, and wins.
public void testCheckSpace_middle()	A 100 by 100 game board (gb) with a single token in the middle, pos = new BoardPosition(49,49)	gb.checkSpace(pos) returns **true**	**Routine Case**: Test if the program can check a space in the middle of the game board.
public void testCheckSpace_topLeft()	A 100 by 100 game board (gb) with a single token in the top left, pos = new BoardPosition(0,0)	gb.checkSpace(pos) returns **true**	**Boundary Case**: Test if the program can check a space in the top left corner of the game board.
public void testCheckSpace_bottomRight()	A 100 by 100 game board (gb) with a single token in the bottom right, pos = new BoardPosition(99,99)	gb.checkSpace(pos) returns **true**	**Boundary Case**: Test if the program can check a space in the bottom right corner of the game board.
public void testHorizontalWin_row3_col3_win3_top()	A three by three game board (gb) with three tokens in a horizontal row at the top, pos = new BoardPosition(0,2) which is the latest position taken before checking for win, token = 'X'	gb.checkHorizontalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can compute a win condition on a game board with the minimum number of rows, columns, and tokens in a row necessary.
testHorizontalWin_row3_col3_win3_bottom()	A three by three game board (gb) with three tokens in a horizontal row at the bottom, pos = new BoardPosition(2,2) which is the latest position taken before checking for win, token = 'Y'	gb.checkHorizontalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a horizontal row on bottom of a game board with a minimum number of rows, columns, and necessary tokens in a row.
public void testHorizontalWin_row100_col100_win25_top()	A 100 by 100 game board (gb) with 25 tokens in a horizontal row at the top, pos = new BoardPosition(0,99) which is the latest position taken before checking for win, token = 'X'	gb.checkHorizontalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a horizontal row on top of a game board with a maximum number of rows, columns, and

			necessary tokens in a row.
public void testHorizontalWin_row100_col100_win25_bottom()	A 100 by 100 game board (gb) with 25 tokens in a horizontal row at the bottom, pos = new BoardPosition(99,99) which is the latest position taken before checking for win, token = 'Y'	gb.checkHorizontalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a horizontal row on bottom of a game board with a maximum number of rows, columns, and necessary tokens in a row.
public void testCheckVerticalWin_row3_col3_win3_left()	A three by three game board (gb) with three tokens in a vertical row at the left, pos = new BoardPosition(2,0) which is the latest position taken before checking for win, token = 'X'	gb.checkVerticalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a vertical row on the left side of a game board with a minimum number of rows, columns, and tokens necessary to win.
public void testCheckVerticalWin_row3_col3_win3_right()	A three by three game board (gb) with three tokens in a vertical row at the right, pos = new BoardPosition(2,2) which is the latest position taken before checking for win, token = 'Y'	gb.checkVerticalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a vertical row on the right side of a game board with a minimum number of rows, columns, and tokens necessary to win.
public void testCheckVerticalWin_row100_col100_win25_left()	A 100 by 100 game board (gb) with 25 tokens in a vertical row at the left, pos = new BoardPosition(99,0) which is the latest position taken before checking for win, token = 'X'	gb.checkVerticalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a win condition with tokens placed in a vertical row on the left side of a game board with a maximum number of rows, columns, and tokens necessary to win.
public void testCheckVerticalWin_row100_col100_win25	A 100 by 100 game board (gb) with 25 tokens in a vertical row at the	gb.checkVerticalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a

_right()	right, pos = new BoardPosition(99,99) which is the latest position taken before checking for win, token = 'Y'		win condition with tokens placed in a vertical row on the right side of a game board with a maximum number of rows, columns, and tokens necessary to win.
public void testCheckDiagonalWin_row3_col3_win3_topLeftToBottomRight()	A three by three game board (gb) with three tokens in a diagonal row going from top left to bottom right, pos = new BoardPosition(2,2) which is the latest position taken before checking for win, token = 'X'	gb.checkDiagonalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a diagonal win condition with the winning row spanning from the top left to the bottom right on a game board with a minimum number of rows, columns, and necessary tokens in a row.
public void testCheckDiagonalWin_row3_col3_win3_topRightToBottomLeft()	A three by three game board (gb) with three tokens in a diagonal row going from top right to bottom left, pos = new BoardPosition(2,0) which is the latest position taken before checking for win, token = 'Y'	gb.checkDiagonalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a diagonal win condition with the winning row spanning from the top right to the bottom left on a game board with a minimum number of rows, columns, and necessary tokens in a row.
public void testCheckDiagonalWin_row100_col100_win25_topLeftToBottomRight()	A 100 by 100 game board (gb) with 25 tokens in a diagonal row going from top left to bottom right, pos = new BoardPosition(99,99) which is the latest position taken before checking for win, token = 'X'	gb.checkDiagonalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a diagonal win condition with the winning row spanning from the top left to the bottom right on a game board with a maximum number of rows, columns, and necessary tokens in a row.
public void testCheckDiagonalWin_row100_col100_win25_topRightToBottomLeft()	A three by three game board (gb) with three tokens in a diagonal row going from top right to bottom left, pos = new BoardPosition(99,0)	gb.checkDiagonalWin(pos, token) returns **true**	**Boundary Case**: Test if the program can correctly compute a diagonal win condition with the winning row spanning from the top

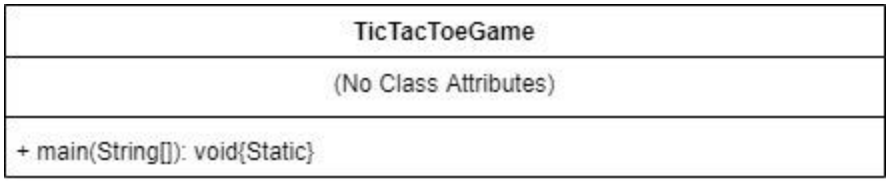
	which is the latest position taken before checking for win, token = 'Y'		right to the bottom left on a game board with a maximum number of rows, columns, and necessary tokens in a row.
public void testCheckDiagonalWin_row3_col3_win3_lastPosInMiddle()	A three by three game board (gb) with three tokens in a diagonal row where the last token placed is the middle token, pos = new BoardPosition(1,1) which is the latest position taken before checking for win, token = 'X'	gb.checkDiagonalWin(pos, token) returns **true**	**Challenging Case**: Test if the program can correctly compute a diagonal win condition when the last, winning token is placed in the back of the winning row.
public void testCheckDiagonalWin_row3_col3_win3_lastPosInBack()	A three by three game board (gb) with three tokens in a diagonal row where the last token placed is the back token, pos = new BoardPosition(0,0) which is the latest position taken before checking for win, token = 'X'	gb.checkDiagonalWin(pos, token) returns **true**	**Challenging Case**: Test if the program can correctly compute a diagonal win condition when the last, winning token is placed in the middle of the winning row.
public void testCheckDiagonalWin_row3_col3_win3_playerBlock()	A three by three game board where opposing tokens block each other from winning, pos = new BoardPosition(1,1) which is the latest position taken before checking for win	gb.checkDiagonalWin(pos, token) returns **false**	**Challenging Case**: Test if the program can compute diagonal win condition as false when one player's attempt to place several tokens in a row is thwarted by another player.
public void testCheckForDraw_fullBoard_tokenX()	A game board full of 'X' tokens, pos = new BoardPosition(2,2) which is latest position before checking for draw	gb.checkForDraw() returns **true**	**Routine Case**: Test if the program can compute that a draw has occurred with a full board of "X" tokens.
public void testCheckForDraw_fullBoard_tokenY()	A game board full of 'X' tokens, pos = new BoardPosition(2,2) which is latest position before checking for draw	gb.checkForDraw() returns **true**	**Routine Case**: Test if the program can compute that a draw has occurred with a full board of "Y" tokens.
public void testCheckForDraw_fullBoard_mixTokens()	A game board full of 'X' and 'Y' tokens, pos = new BoardPosition(2,2) which is latest position before checking for draw	gb.checkForDraw() returns **true**	**Challenging Case**: Test if the program can compute that a draw has occurred with a full board of "X" and "Y" tokens.

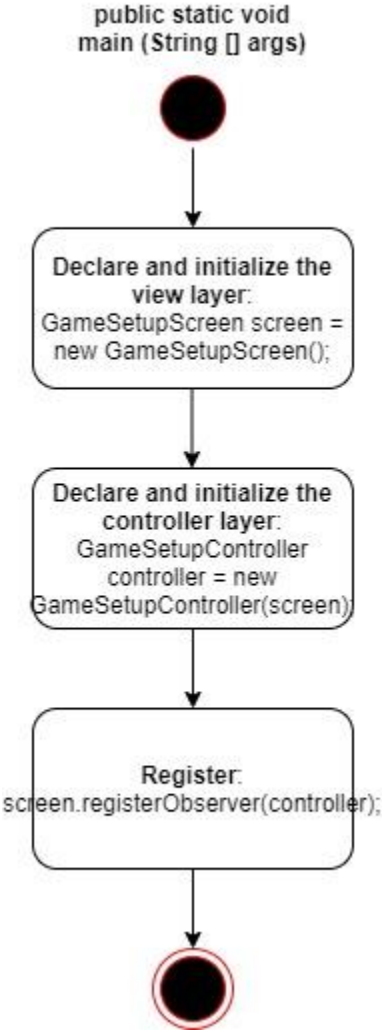
public void testCheckForDraw_emptyBoard()	An empty game board, pos = new BoardPosition(0,0) because no token has been placed	gb.checkForDraw() returns **false**	**Routine Case**: Test if the program can compute that a draw has not occurred on an empty board.
public void testWhatsAtPos_pos0_col0_tokenX()	A three by three game board (gb) with a single 'X' token at the top left, pos = new BoardPosition(0,0)	gb.whatsAtPos(pos) == 'X'	**Routine Case**: Test if the program can correctly identity that an "X" token is at a given position.
public void testWhatsAtPos_pos2_col2_tokenY()	A three by three game board (gb) with a single 'Y' token at the bottom right, pos = new BoardPosition(2,2)	gb.whatsAtPos(pos) == 'Y'	**Routine Case**: Test if the program can correctly identity that a "Y" token is at a given position at given position.
public void testWhatsAtPos_posRow1_posCol1_multipleTokens()	A three by three game board (gb) with a two different tokens, pos = new BoardPosition(2,2)	gb.whatsAtPos(pos) == 'Y'	**Challenging Case**: Test if the program can correctly identify that a "Y" token is at given position after placing down other types of player tokens (e.g. X, Y, O...).
public void testWhatsAtPos_posRow1_posCol1_tokenXMistakenForTokenY()	A three by three game board (gb) with a single 'X', pos = new BoardPosition(1,1)	gb.whatsAtPos(pos) == 'Y' returns **false**	**Challenging Case**: Test if the program mistakes "Y" to be at a given position that "X" is at.
public void testWhatsAtPos_posRow1_posCol1_tokenYMistakenForTokenX()	A three by three game board (gb) with a single 'Y', pos = new BoardPosition(1,1)	gb.whatsAtPos(pos) == 'X' returns **false**	**Challenging Case**: Test if the program mistakes "Y" to be at a given position that "X" is at.
public void testIsPlayerAtPos_pos0_col0_tokenX()	A three by three game board (gb) with a single 'X' token at the top left, pos = new, BoardPosition(0,0), token = 'X'	gb.isPlayerAtPos(pos, token) returns **true**	**Routine Case**: Test if the program can correctly identify that a "X" token is at a given position.
public void testIsPlayerAtPos_posRow2_posCol2_tokenY()	A three by three game board (gb) with a single 'Y' token at the bottom right, pos = new BoardPosition(2,2), token = 'Y'	gb.isPlayerAtPos(pos, token) returns **true**	**Routine Case**: Test if the program can correctly identify that a "Y" token is at a given position.
public void testIsPlayerAtPos_posRow1_posCol1_multip	A three by three game board (gb) with two types of tokens, , pos = new	gb.isPlayerAtPos(pos, 'Y') returns **true**	**Challenging Case**: Test if the program can correctly identify a

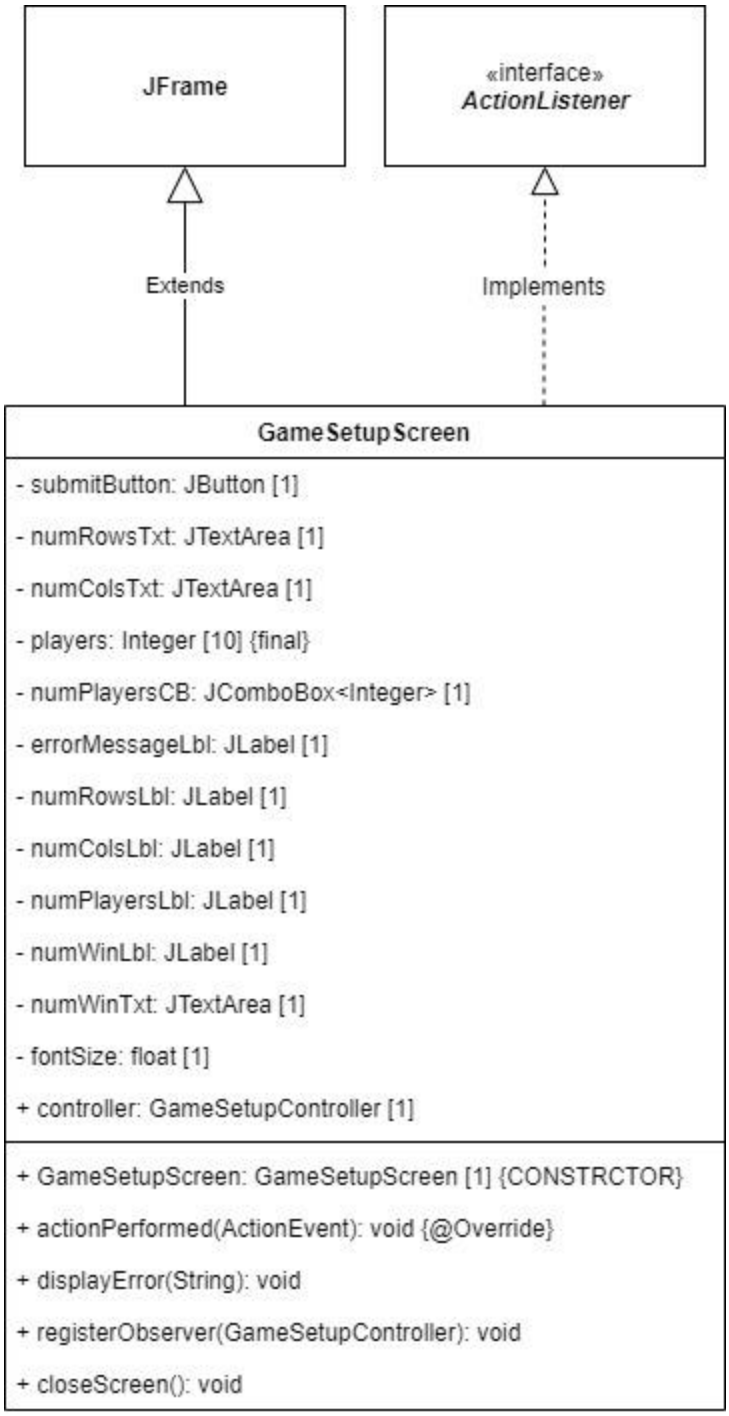
leTokens()	BoardPosition(1,1), token = 'X'		given player token at a given position after placing down different types of player tokens (e.g. X, Y, O...).
public void testIsPlayerAtPos_posRow1_posCol1_tokenXMistakenForTokenY()	A three by three game board (gb) with a single 'X' token at the bottom right, pos = new BoardPosition(1,1), token = 'X'	gb.isPlayerAtPos(pos, 'Y') returns **false**	**Challenging Case**: Test if the program mistakes a "X" token as a "Y" token.
public void testIsPlayerAtPos_posRow1_posCol1_tokenYMistakenForTokenX()	A three by three game board (gb) with a single 'Y' token at the bottom right, pos = new BoardPosition(1,1), token = 'Y'	gb.isPlayerAtPos(pos, 'X') returns **false**	**Challenging Case**: Test if the program mistakes a "Y" token as a "X" token.
public void testPlaceMarker_row3_col3_topLeft()	A 3 by 3 game board (gb) with a single token at the top left	gb.toString() will print a game board containing a tokne in the top left	**Boundary Case**: Test if the program can place a token at the top left corner of a game board with a minimum number of rows, columns and tokens necessary to win.
public void testPlaceMarker_row3_col3_bottomRight()	A 3 by 3 game board (gb) with a single token at the bottom right	gb.toString() will print a game board containing a tokne in the bottom right	**Boundary Case**: Test if the program can place a token at the bottom right corner of a game board with a minimum number of rows, columns and tokens necessary to win.
public void testPlaceMarker_row100_col100_topLeft()	A 100 by 100 game board (gb) with a single token at the top left	gb.toString() will print a game board containing a tokne in the top left	**Boundary Case**: Test if the program can place a token at the top left corner of a game board with a maximum number of rows, columns and tokens necessary to win.
public void testPlaceMarker_row100_col100_bottomRight()	A 100 by 100 game board (gb) with a single token at the bottom right	gb.toString() will print a game board containing a tokne in the bottom right	**Boundary Case**: Test if the program can place a token at the bottom right corner of a game board with a maximum number of rows, columns and tokens necessary to win.
public void	A game board (gb) with a	gb.toString() will print a	**Challenging Case**: Test

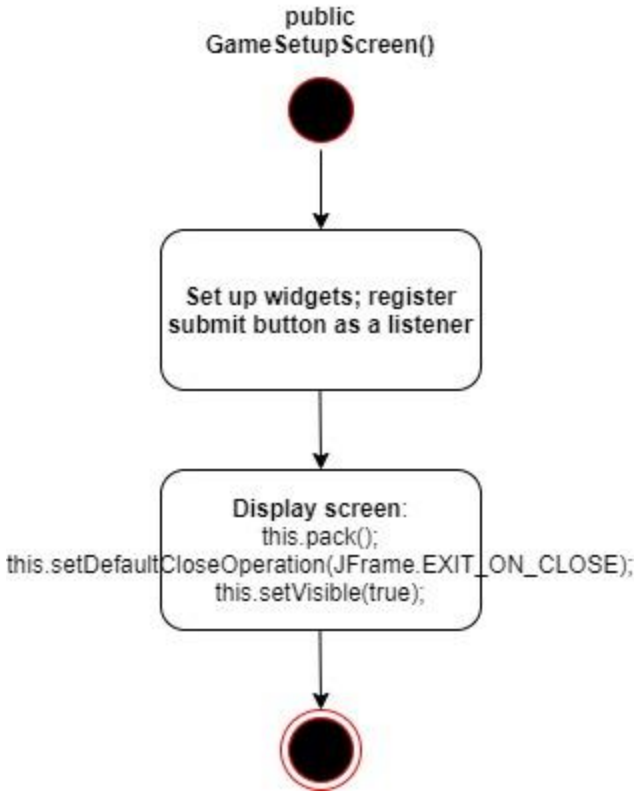
testPlaceMarker_row10_row10_fullRow()	full row of tokens	game board containing a full row of tokens	if the program can place a full row of tokens.

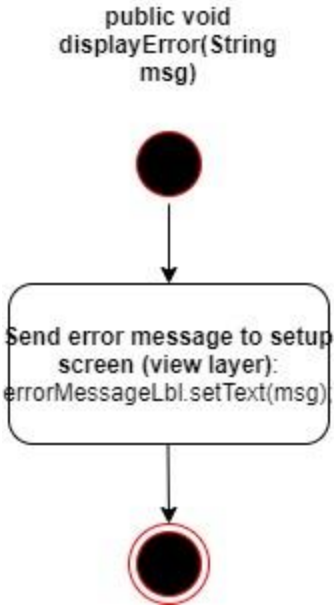
Design

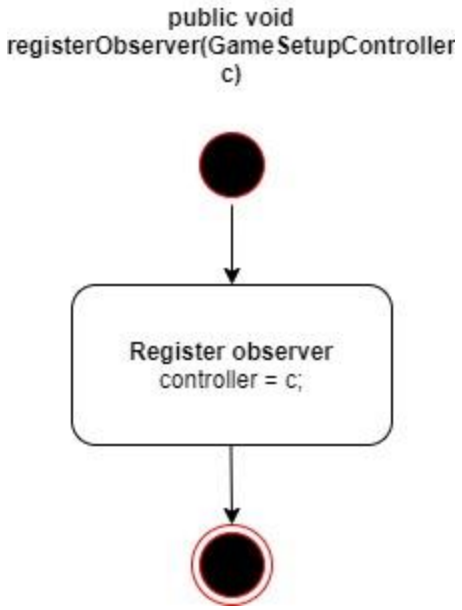

TicTacToeGame
(No Class Attributes)
+ main(String[]): void{Static}

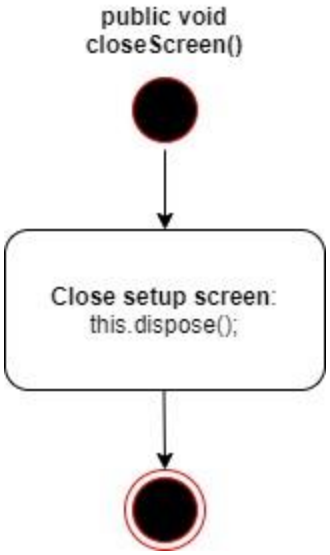

public static void
main (String [] args)
Declare and initialize the view layer:
GameSetupScreen screen = new GameSetupScreen();
Declare and initialize the controller layer:
GameSetupController controller = new GameSetupController(screen);
Register:
screen.registerObserver(controller);

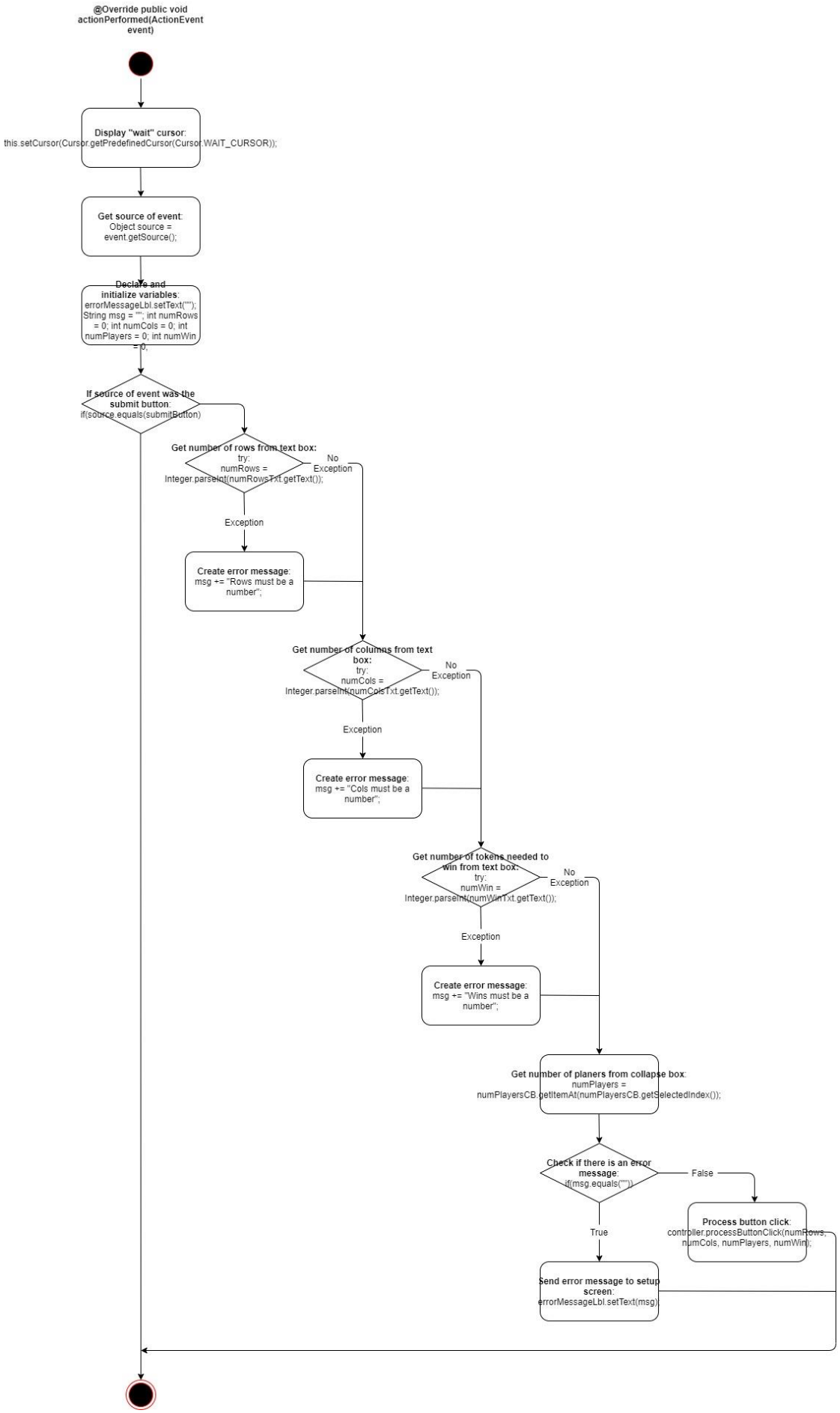

JFrame
«interface»
ActionListener
Extends
Implements
GameSetupScreen
- submitButton: JButton [1]
- numRowsTxt: JTextArea [1]
- numColsTxt: JTextArea [1]
- players: Integer [10] {final}
- numPlayersCB: JComboBox<Integer> [1]
- errorMessageLbl: JLabel [1]
- numRowsLbl: JLabel [1]
- numColsLbl: JLabel [1]
- numPlayersLbl: JLabel [1]
- numWinLbl: JLabel [1]
- numWinTxt: JTextArea [1]
- fontSize: float [1]
+ controller: GameSetupController [1]
+ GameSetupScreen: GameSetupScreen [1] {CONSTRCTOR}
+ actionPerformed(ActionEvent): void {@Override}
+ displayError(String): void
+ registerObserver(GameSetupController): void
+ closeScreen(): void


public GameSetupScreen()
Set up widgets; register submit button as a listener
Display screen:
this.pack();
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setVisible(true);

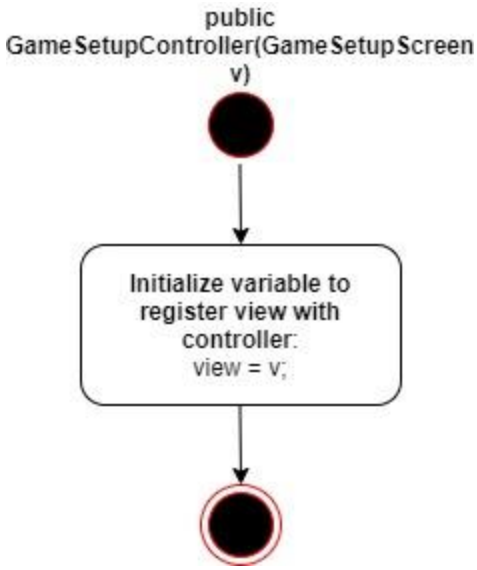

public void displayError(String msg)
Send error message to setup screen (view layer):
errorMessageLbl.setText(msg);

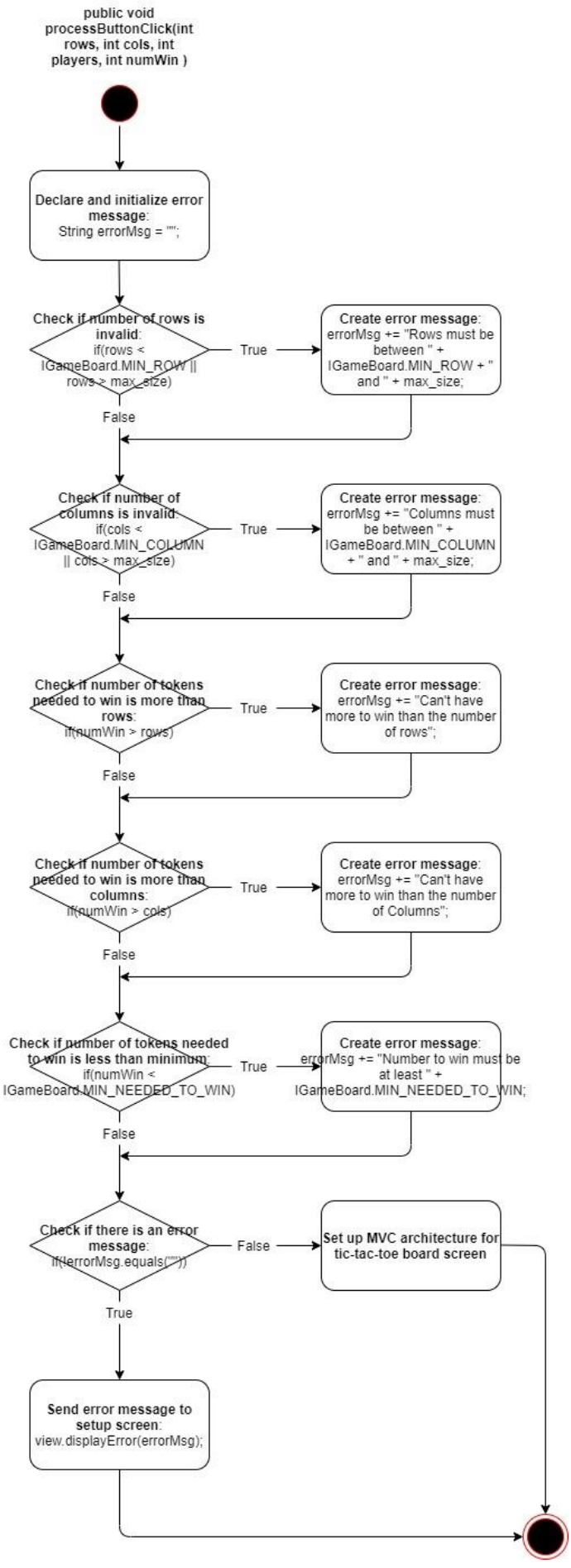

public void registerObserver(GameSetupController c)
Register observer
controller = c;

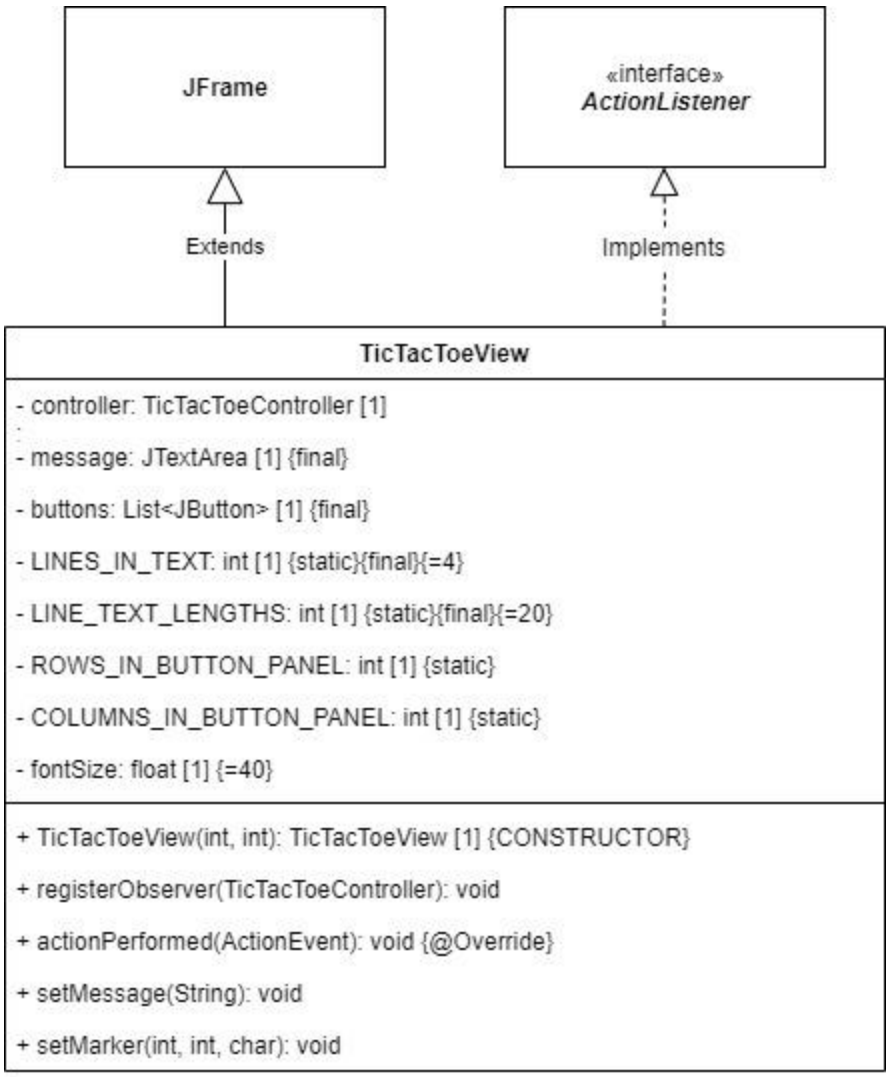

public void closeScreen()
Close setup screen:
this.dispose();

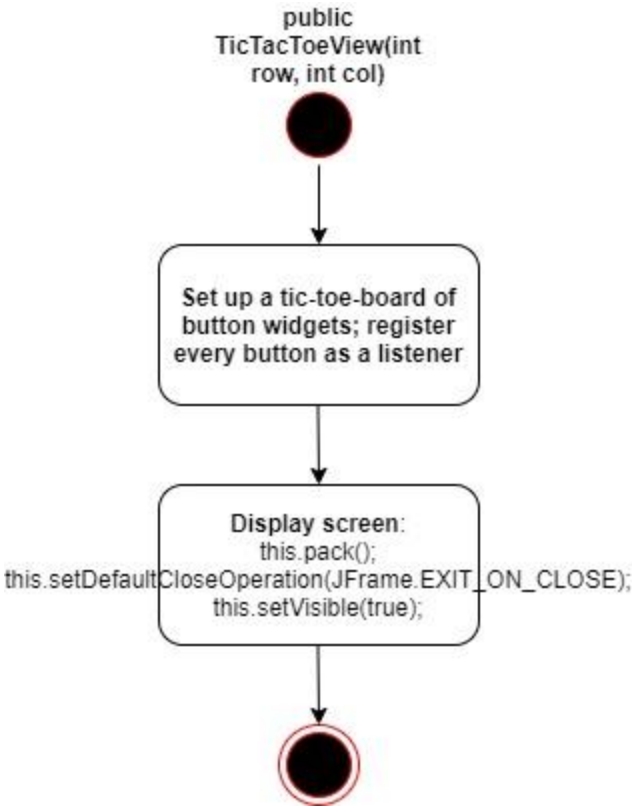

@Override public void actionPerformed(ActionEvent event)
Display "wait" cursor:
this.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
Get source of event:
Object source = event.getSource();
Declare and initialize variables:
errorMessageLbl.setText("");
String msg = ""; int numRows = 0; int numCols = 0; int numPlayers = 0; int numWin = 0;
If source of event was the submit button:
if(source.equals(submitButton)
Get number of rows from text box:
try:
numRows = Integer.parseInt(numRowsTxt.getText());
No Exception
Exception
Create error message:
msg += "Rows must be a number";
Get number of columns from text box:
try:
numCols = Integer.parseInt(numColsTxt.getText());
No Exception
Exception
Create error message:
msg += "Cols must be a number";
Get number of tokens needed to win from text box:
try:
numWin = Integer.parseInt(numWinTxt.getText());
No Exception
Exception
Create error message:
msg += "Wins must be a number";
Get number of planers from collapse box:
numPlayers = numPlayersCB.getItemAt(numPlayersCB.getSelectedIndex());
Check if there is an error message:
if(msg.equals(""))
False
True
Process button click:
controller.processButtonClick(numRows, numCols, numPlayers, numWin);
Send error message to setup screen:
errorMessageLbl.setText(msg);

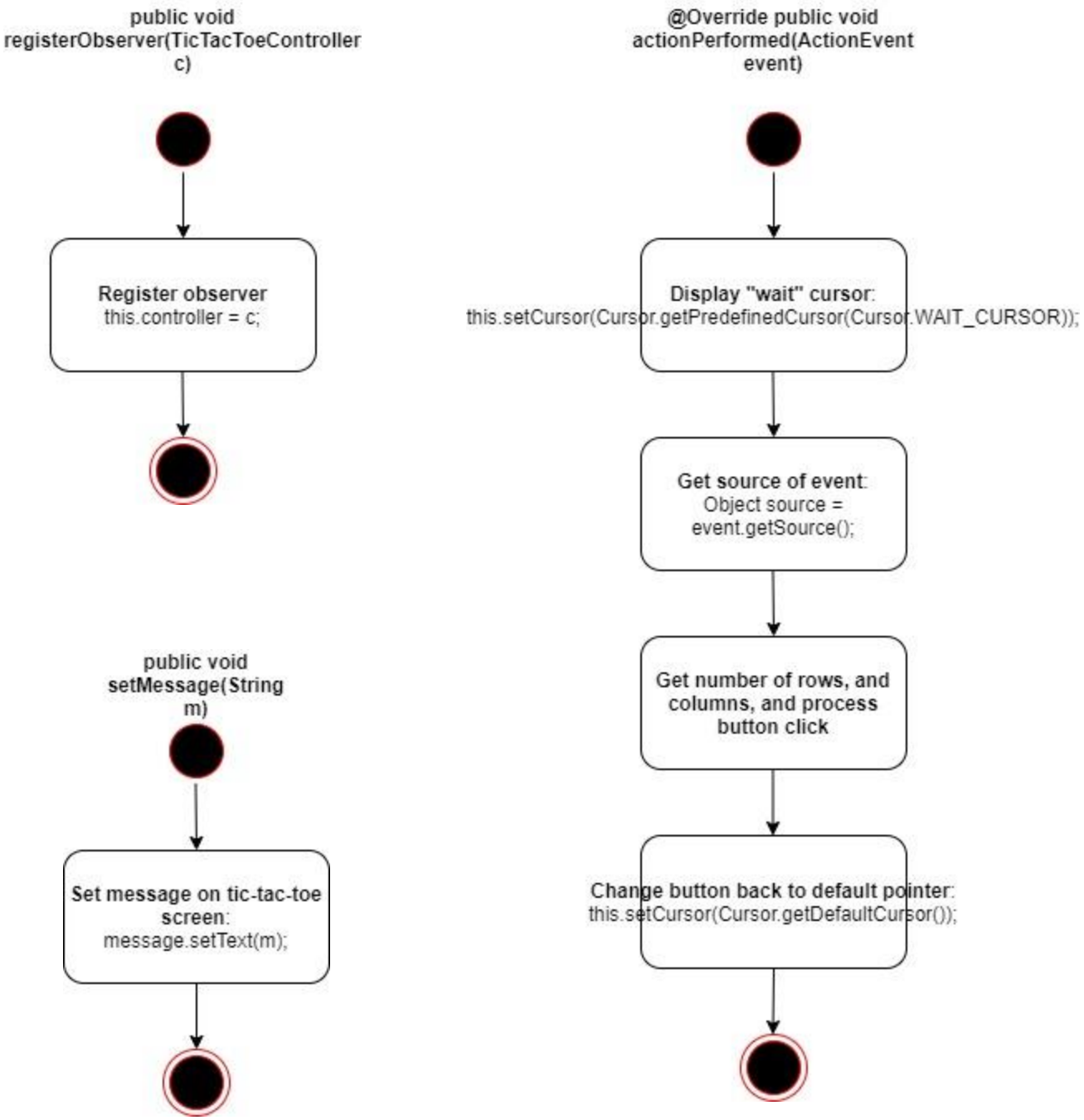
GameSetupController
- view: GameSetupScreen [1] - max_size: int [1] {=20} - MEM_CUTOFF: int [1] {final}{=64}
+ GameSetupController(GameSetupScreen): GameSetupController [1] {CONSTRUCTOR} + processButtonClick(int, int, int, int): void

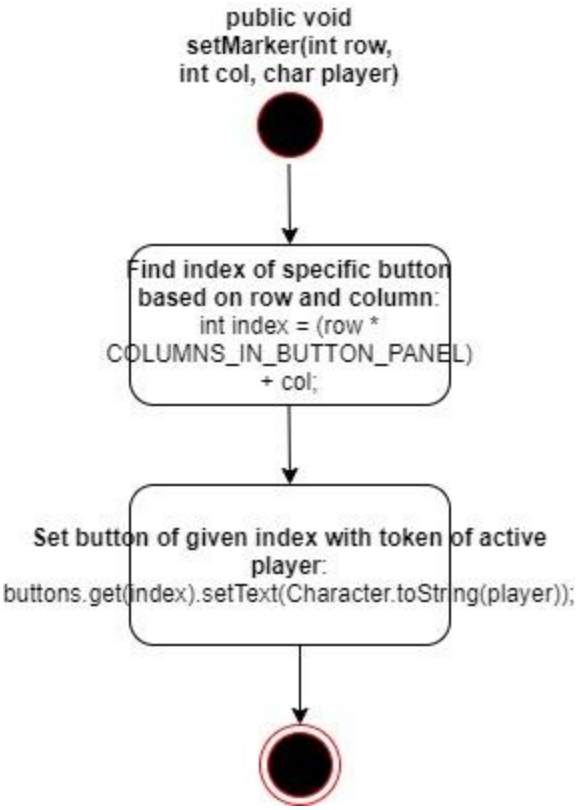



public void processButtonClick(int rows, int cols, int players, int numWin)
Declare and initialize error message: String errorMsg = "";
Check if number of rows is invalid: if(rows < IGameBoard.MIN_ROW || rows > max_size)
True
Create error message: errorMsg += "Rows must be between " + IGameBoard.MIN_ROW + " and " + max_size;
False
Check if number of columns is invalid: if(cols < IGameBoard.MIN_COLUMN || cols > max_size)
True
Create error message: errorMsg += "Columns must be between " + IGameBoard.MIN_COLUMN + " and " + max_size;
False
Check if number of tokens needed to win is more than rows: if(numWin > rows)
True
Create error message: errorMsg += "Can't have more to win than the number of rows";
False
Check if number of tokens needed to win is more than columns: if(numWin > cols)
True
Create error message: errorMsg += "Can't have more to win than the number of Columns";
False
Check if number of tokens needed to win is less than minimum: if(numWin < IGameBoard.MIN_NEEDED_TO_WIN)
True
Create error message: errorMsg += "Number to win must be at least " + IGameBoard.MIN_NEEDED_TO_WIN;
False
Check if there is an error message: if(!errorMsg.equals(""))
False
Set up MVC architecture for tic-tac-toe board screen
True
Send error message to setup screen: view.displayError(errorMsg);

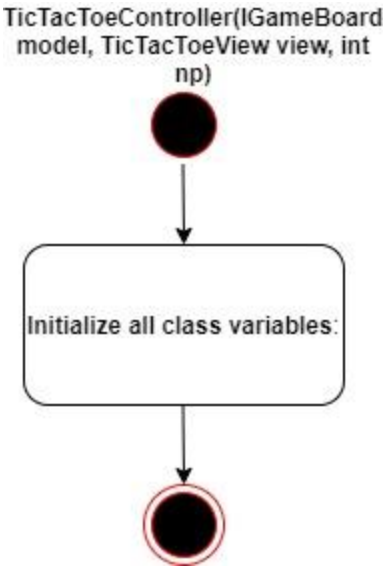

JFrame
«interface»
ActionListener
Extends
Implements
TicTacToeView
- controller: TicTacToeController [1]
- message: JTextArea [1] {final}
- buttons: List<JButton> [1] {final}
- LINES_IN_TEXT: int [1] {static}{final}{=4}
- LINE_TEXT_LENGTHS: int [1] {static}{final}{=20}
- ROWS_IN_BUTTON_PANEL: int [1] {static}
- COLUMNS_IN_BUTTON_PANEL: int [1] {static}
- fontSize: float [1] {=40}
+ TicTacToeView(int, int): TicTacToeView [1] {CONSTRUCTOR}
+ registerObserver(TicTacToeController): void
+ actionPerformed(ActionEvent): void {@Override}
+ setMessage(String): void
+ setMarker(int, int, char): void

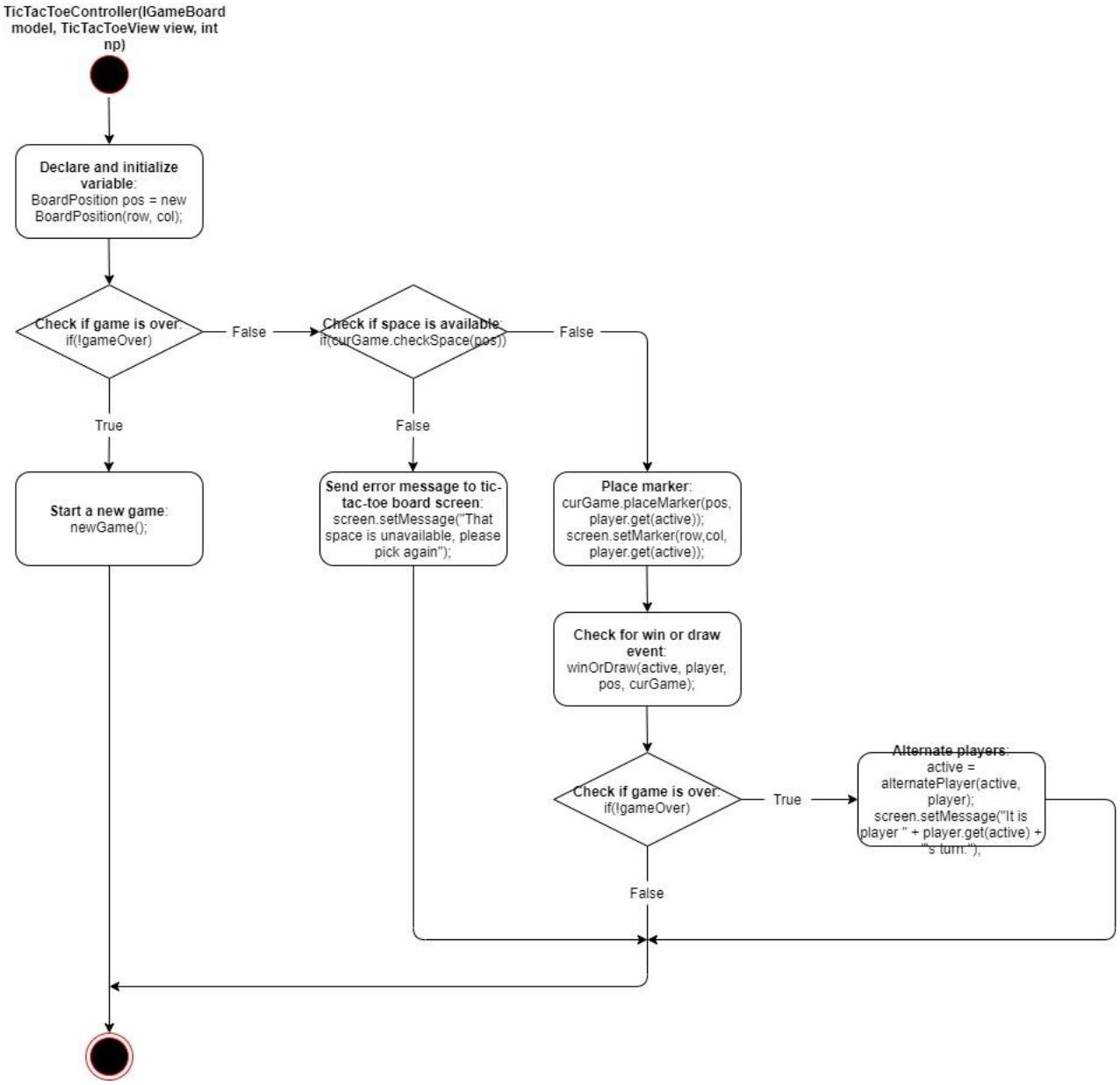

public TicTacToeView(int row, int col)
Set up a tic-toe-board of button widgets; register every button as a listener
Display screen:
this.pack();
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setVisible(true);

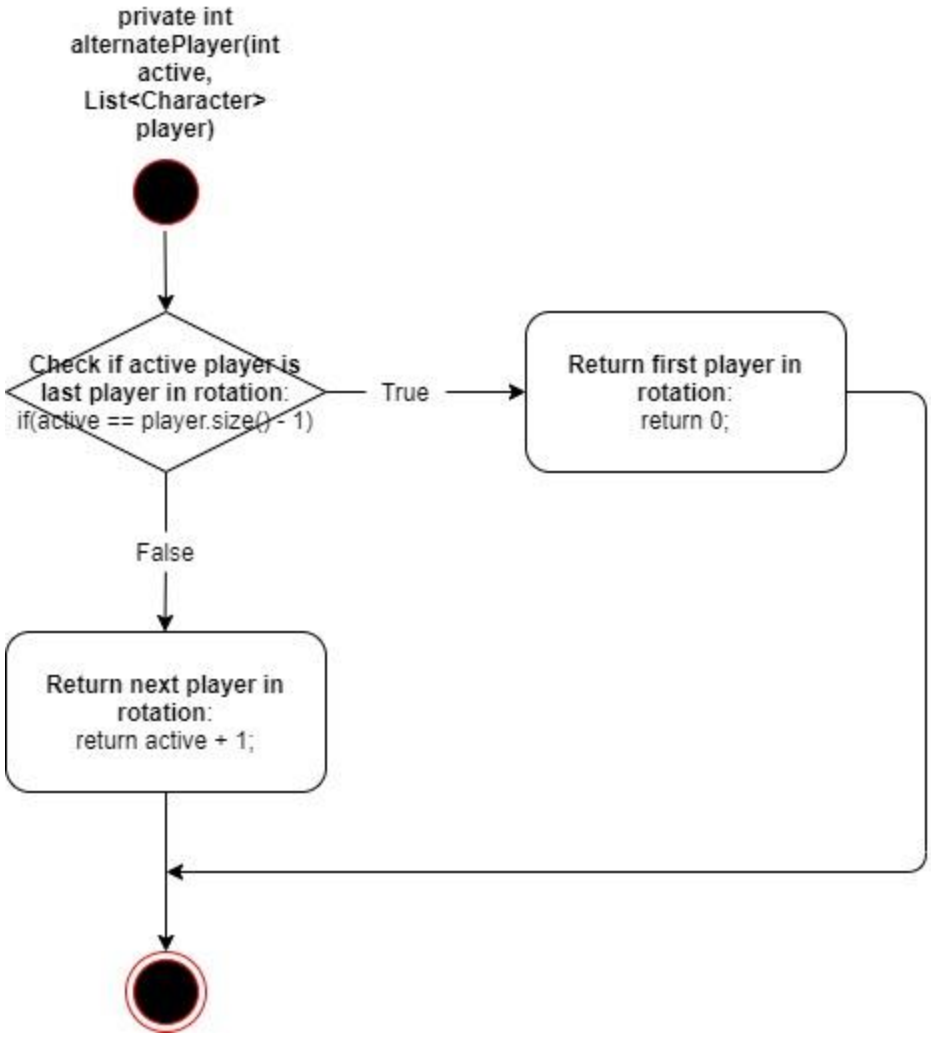

public void registerObserver(TicTacToeController c)
Register observer
this.controller = c;
@Override public void actionPerformed(ActionEvent event)
Display "wait" cursor:
this.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
Get source of event:
Object source = event.getSource();
Get number of rows, and columns, and process button click
Change button back to default pointer:
this.setCursor(Cursor.getDefaultCursor());
public void setMessage(String m)
Set message on tic-tac-toe screen:
message.setText(m);

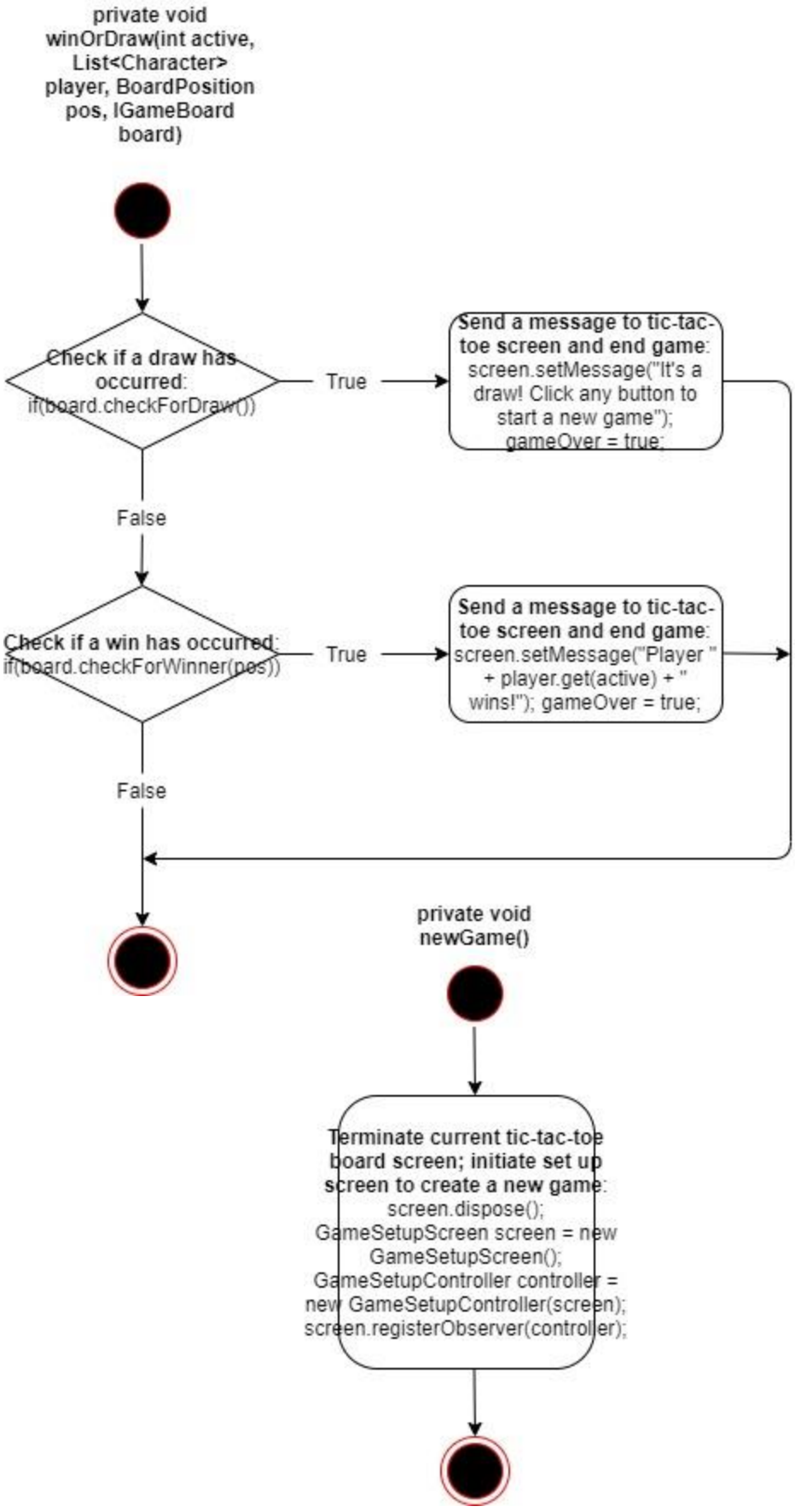

public void
setMarker(int row,
int col, char player)
Find index of specific button
based on row and column:
int index = (row *
COLUMNS_IN_BUTTON_PANEL)
+ col;
Set button of given index with token of active
player:
buttons.get(index).setText(Character.toString(player));

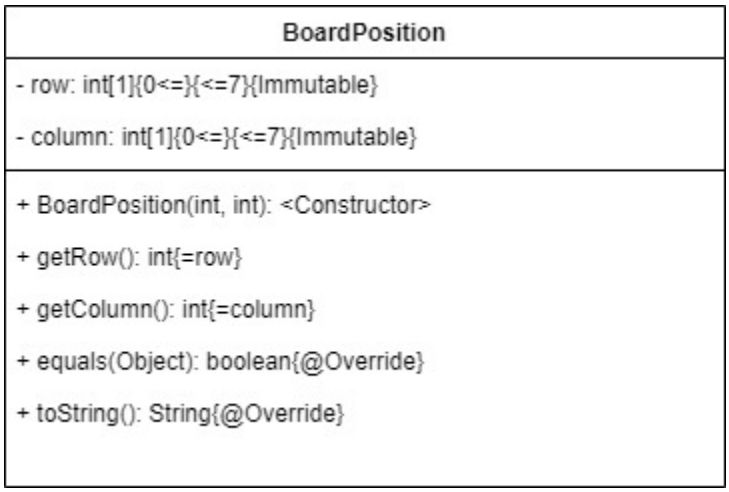
TicTacToeController
- active: int [1] - gameOver: boolean [1] - player: List<Character> [1] - curGame: IGameBoard [1] - screen: TicTacToeView [1]
+ TicTacToeController(IGameBoard, TicTacToeView, int): TicTacToeController [1] {CONSTRUCTOR} + processButtonClick(int, int): void - alternatePlayer(int, List<Character>): int [1] - winOrDraw(int, List<Character>, BoardPosition, IGameBoard): void - newGame(): void

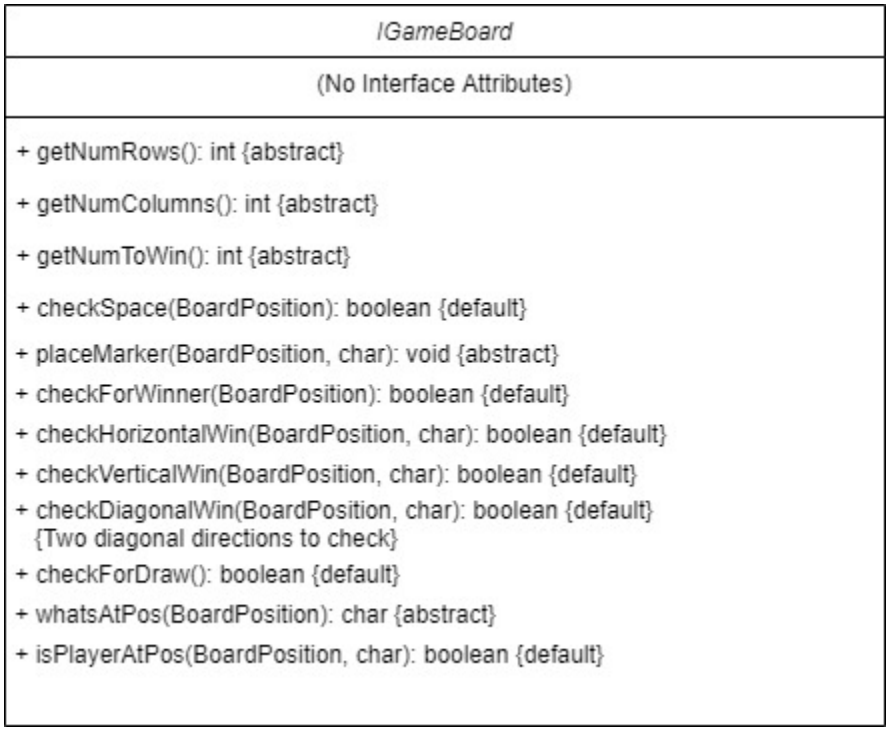


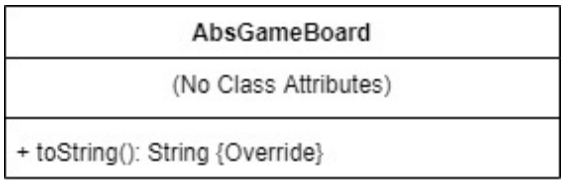

TicTacToeController(IGameBoard model, TicTacToeView view, int np)
Declare and initialize variable: BoardPosition pos = new BoardPosition(row, col);
Check if game is over: if(!gameOver)
False
Check if space is available: if(curGame.checkSpace(pos))
False
True
False
Start a new game: newGame();
Send error message to tic-tac-toe board screen: screen.setMessage("That space is unavailable, please pick again");
Place marker: curGame.placeMarker(pos, player.get(active)); screen.setMarker(row,col, player.get(active));
Check for win or draw event: winOrDraw(active, player, pos, curGame);
Check if game is over: if(!gameOver)
True
Alternate players: active = alternatePlayer(active, player); screen.setMessage("It is player " + player.get(active) + "'s turn.");
False

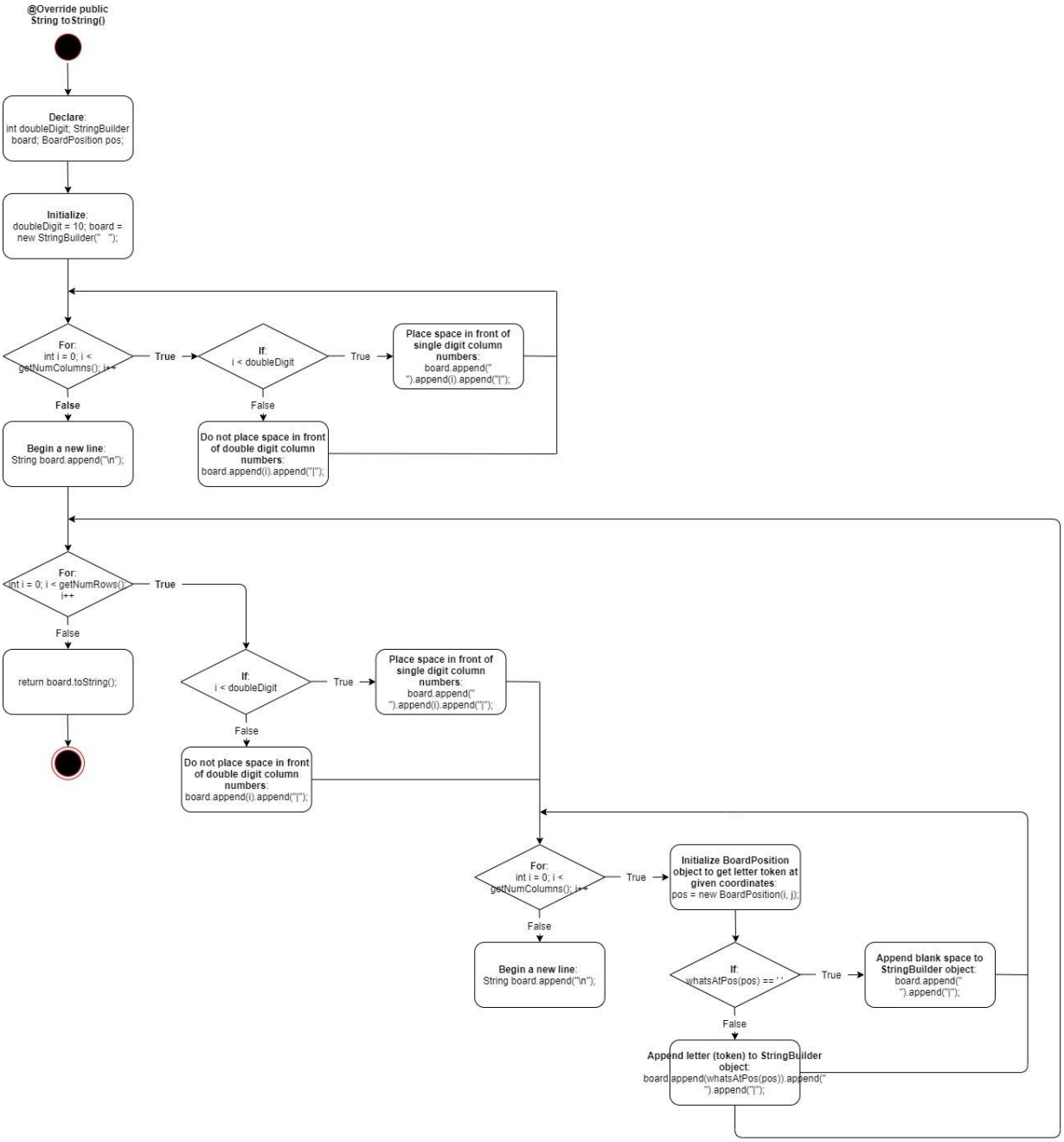

private int alternatePlayer(int active, List<Character> player)
Check if active player is last player in rotation:
if(active == player.size() - 1)
True
Return first player in rotation:
return 0;
False
Return next player in rotation:
return active + 1;

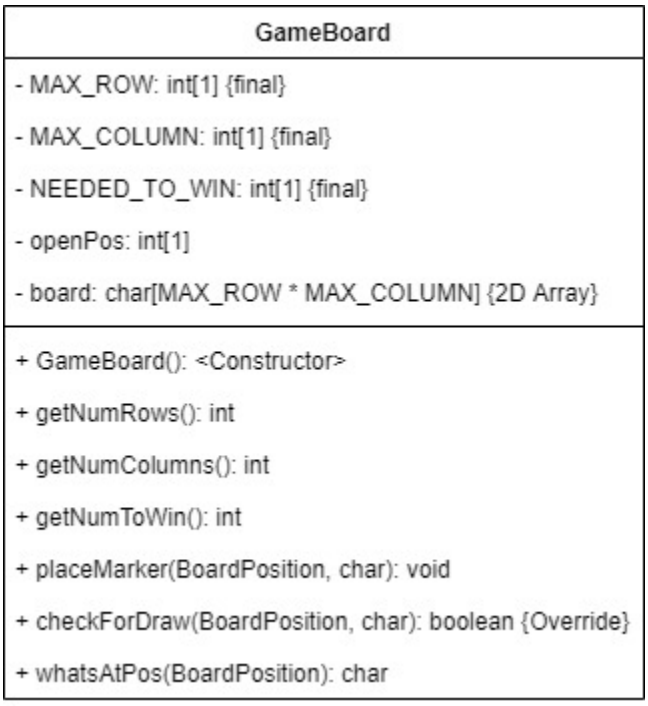

private void winOrDraw(int active, List<Character> player, BoardPosition pos, IGameBoard board)
Check if a draw has occurred: if(board.checkForDraw())
True
Send a message to tic-tac-toe screen and end game: screen.setMessage("It's a draw! Click any button to start a new game"); gameOver = true;
False
Check if a win has occurred: if(board.checkForWinner(pos))
True
Send a message to tic-tac-toe screen and end game: screen.setMessage("Player " + player.get(active) + " wins!"); gameOver = true;
False
private void newGame()
Terminate current tic-tac-toe board screen; initiate set up screen to create a new game: screen.dispose(); GameSetupScreen screen = new GameSetupScreen(); GameSetupController controller = new GameSetupController(screen); screen.registerObserver(controller);

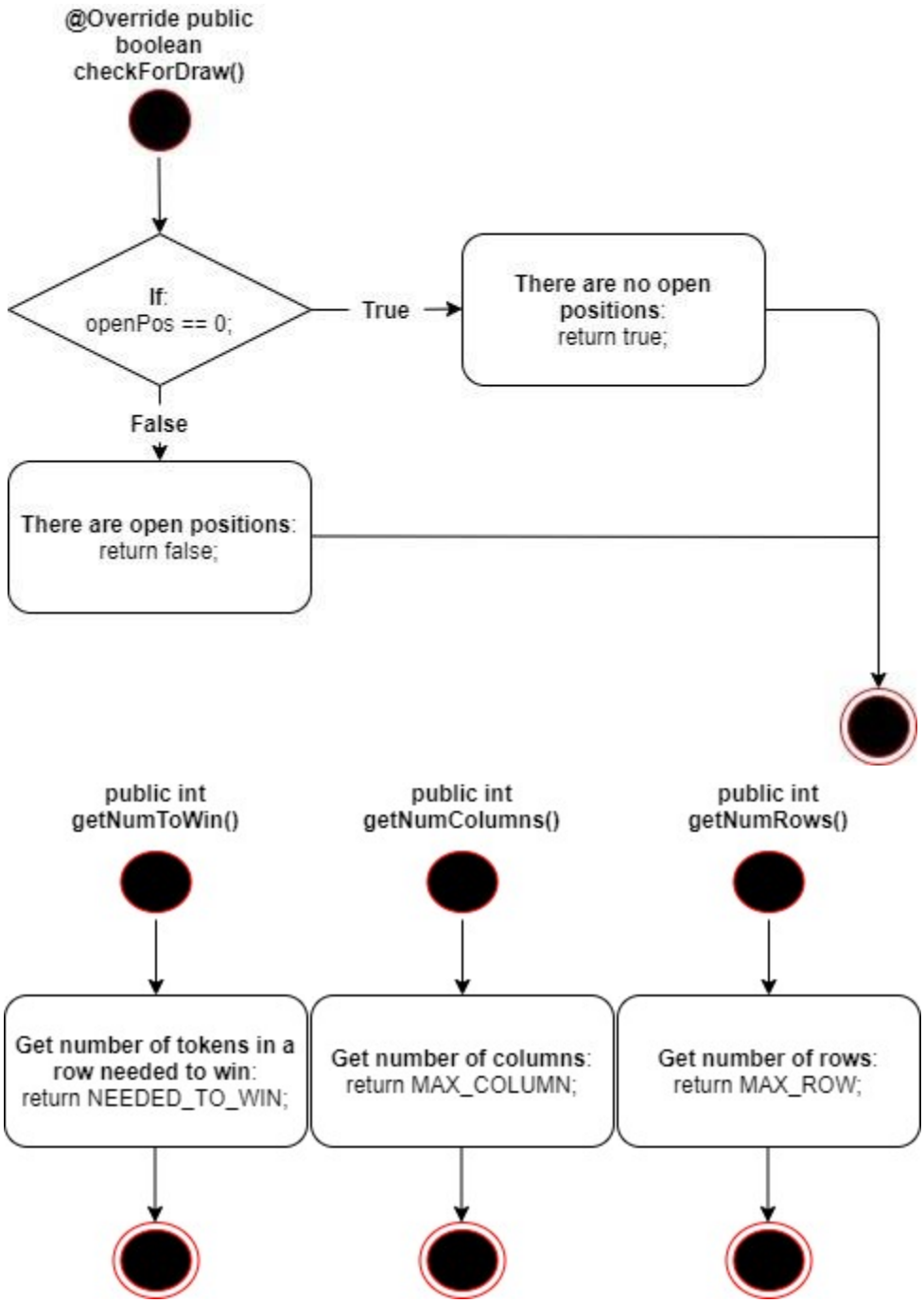

BoardPosition
- row: int[1]{0<=}{<=7}{Immutable}
- column: int[1]{0<=}{<=7}{Immutable}
+ BoardPosition(int, int): <Constructor>
+ getRow(): int{=row}
+ getColumn(): int{=column}
+ equals(Object): boolean{@Override}
+ toString(): String{@Override}

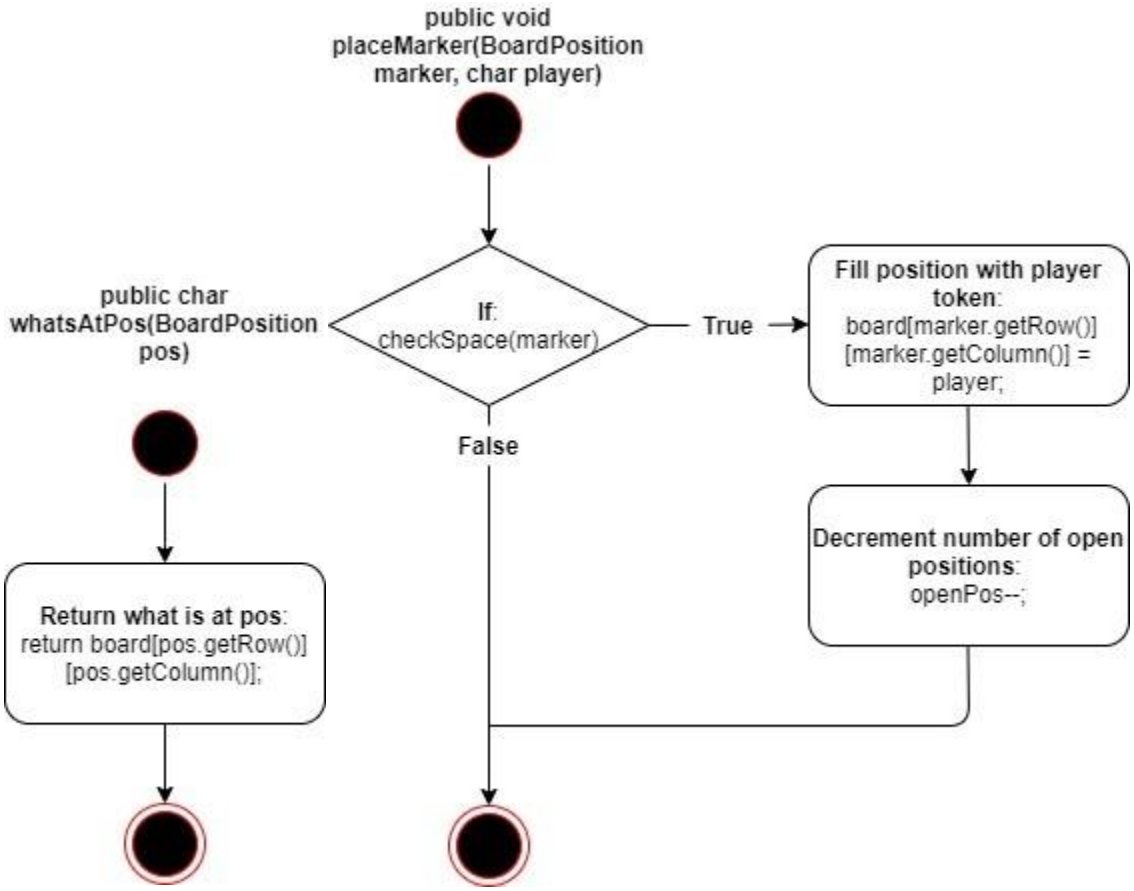

IGameBoard
(No Interface Attributes)
+ getNumRows(): int {abstract}
+ getNumColumns(): int {abstract}
+ getNumToWin(): int {abstract}
+ checkSpace(BoardPosition): boolean {default}
+ placeMarker(BoardPosition, char): void {abstract}
+ checkForWinner(BoardPosition): boolean {default}
+ checkHorizontalWin(BoardPosition, char): boolean {default}
+ checkVerticalWin(BoardPosition, char): boolean {default}
+ checkDiagonalWin(BoardPosition, char): boolean {default}
{Two diagonal directions to check}
+ checkForDraw(): boolean {default}
+ whatsAtPos(BoardPosition): char {abstract}
+ isPlayerAtPos(BoardPosition, char): boolean {default}

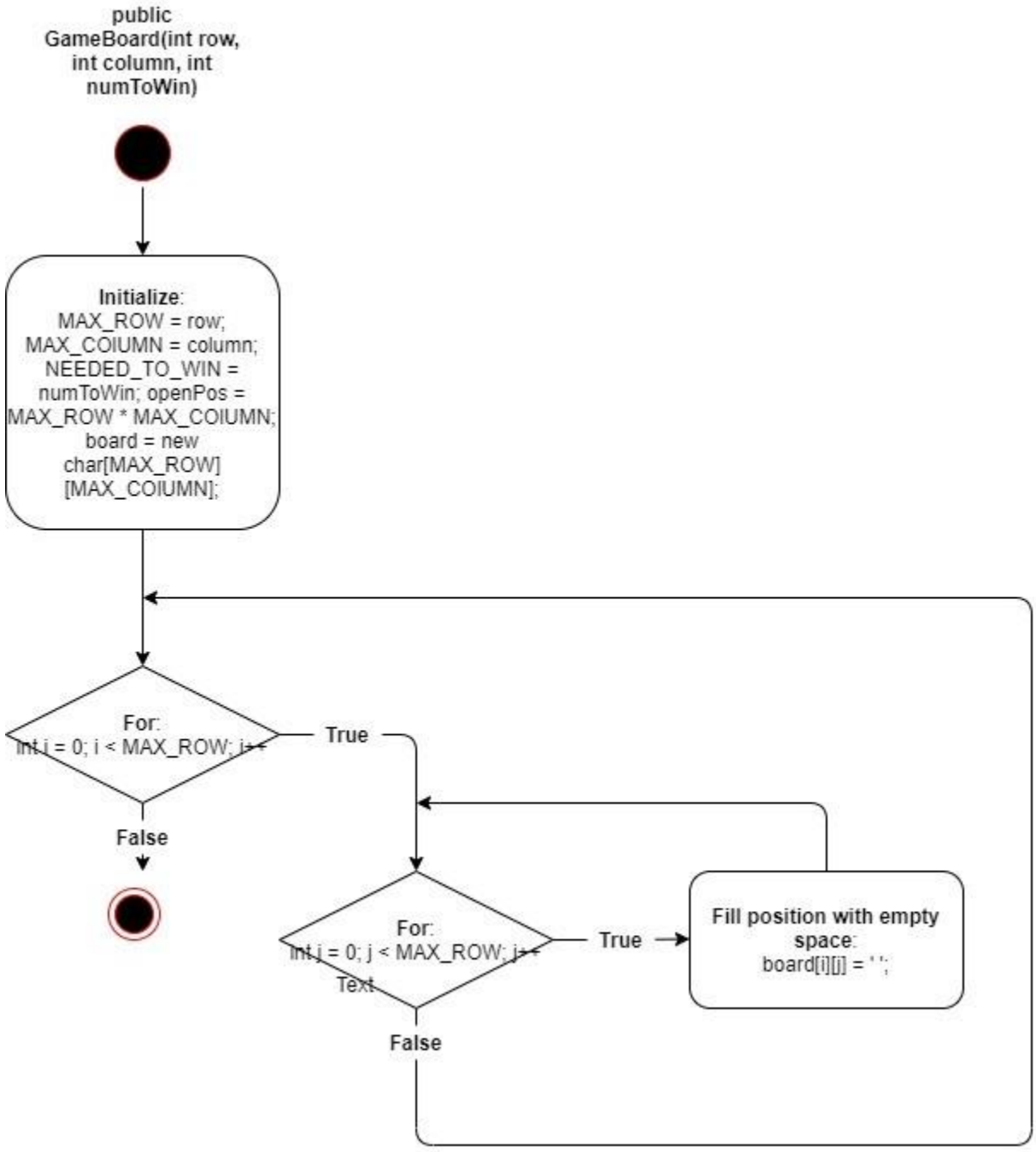

AbsGameBoard
(No Class Attributes)
+ toString(): String {Override}

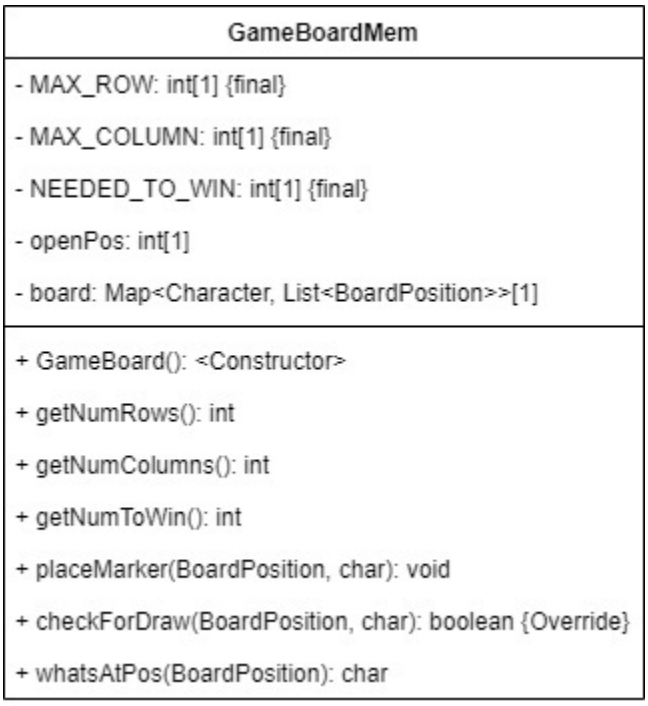

@Override public String toString()
Declare: int doubleDigit; StringBuilder board; BoardPosition pos;
Initialize: doubleDigit = 10; board = new StringBuilder(" ");
For: int i = 0; i < getNumColumns(); i++
True
If: i < doubleDigit
True
Place space in front of single digit column numbers: board.append(" ").append(i).append("|");
False
Do not place space in front of double digit column numbers: board.append(i).append("|");
False
Begin a new line: String board.append("\n");
For: int i = 0; i < getNumRows(); i++
True
If: i < doubleDigit
True
Place space in front of single digit column numbers: board.append(" ").append(i).append("|");
False
Do not place space in front of double digit column numbers: board.append(i).append("|");
False
return board.toString();
For: int i = 0; i < getNumColumns(); i++
True
Initialize BoardPosition object to get letter token at given coordinates: pos = new BoardPosition(i, j);
False
Begin a new line: String board.append("\n");
If: whatsAtPos(pos) == ' '
True
Append blank space to StringBuilder object: board.append(" ").append("|");
False
Append letter (token) to StringBuilder object: board.append(whatsAtPos(pos)).append(" ").append("|");

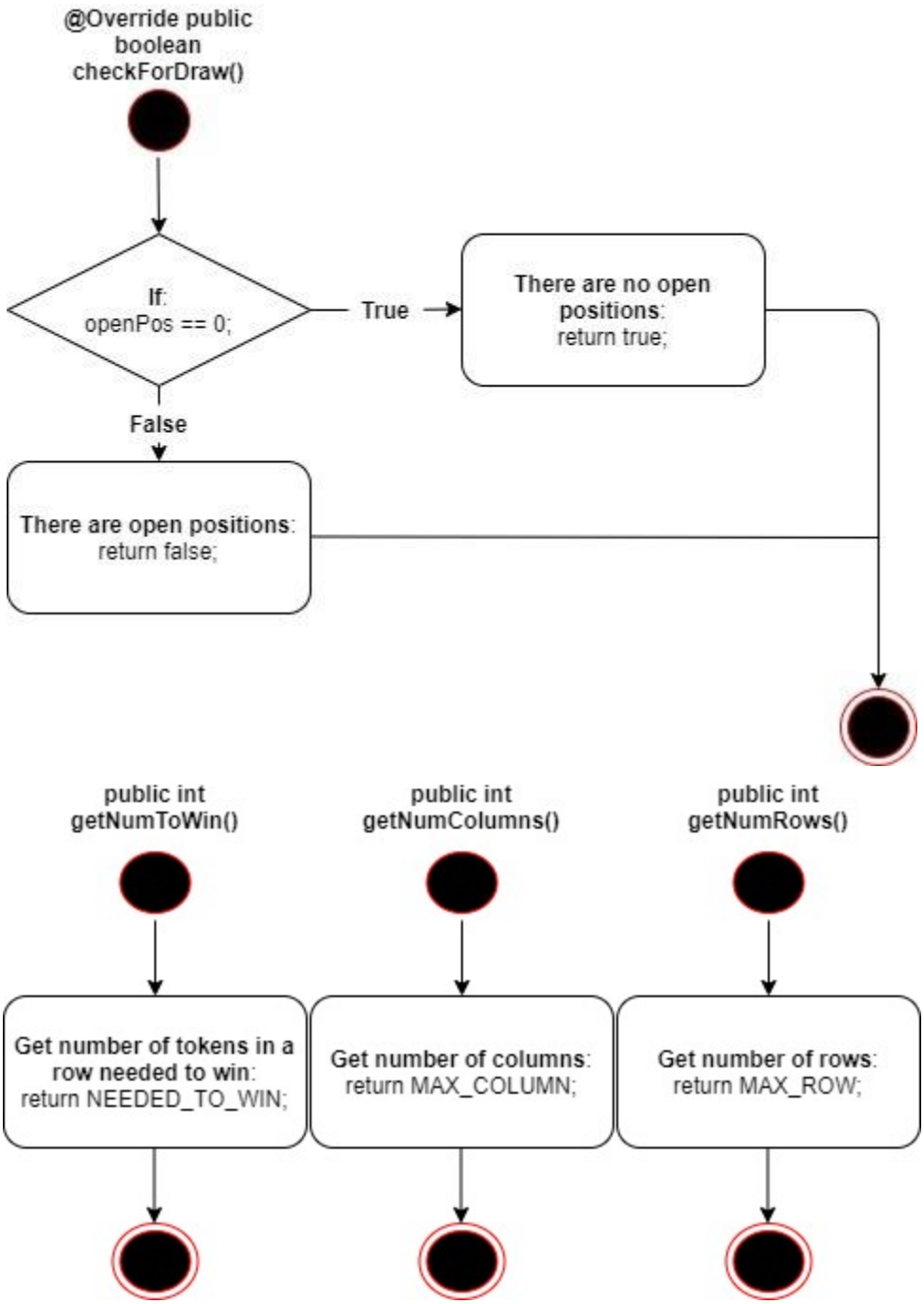

GameBoard
- MAX_ROW: int[1] {final}
- MAX_COLUMN: int[1] {final}
- NEEDED_TO_WIN: int[1] {final}
- openPos: int[1]
- board: char[MAX_ROW * MAX_COLUMN] {2D Array}
+ GameBoard(): <Constructor>
+ getNumRows(): int
+ getNumColumns(): int
+ getNumToWin(): int
+ placeMarker(BoardPosition, char): void
+ checkForDraw(BoardPosition, char): boolean {Override}
+ whatsAtPos(BoardPosition): char

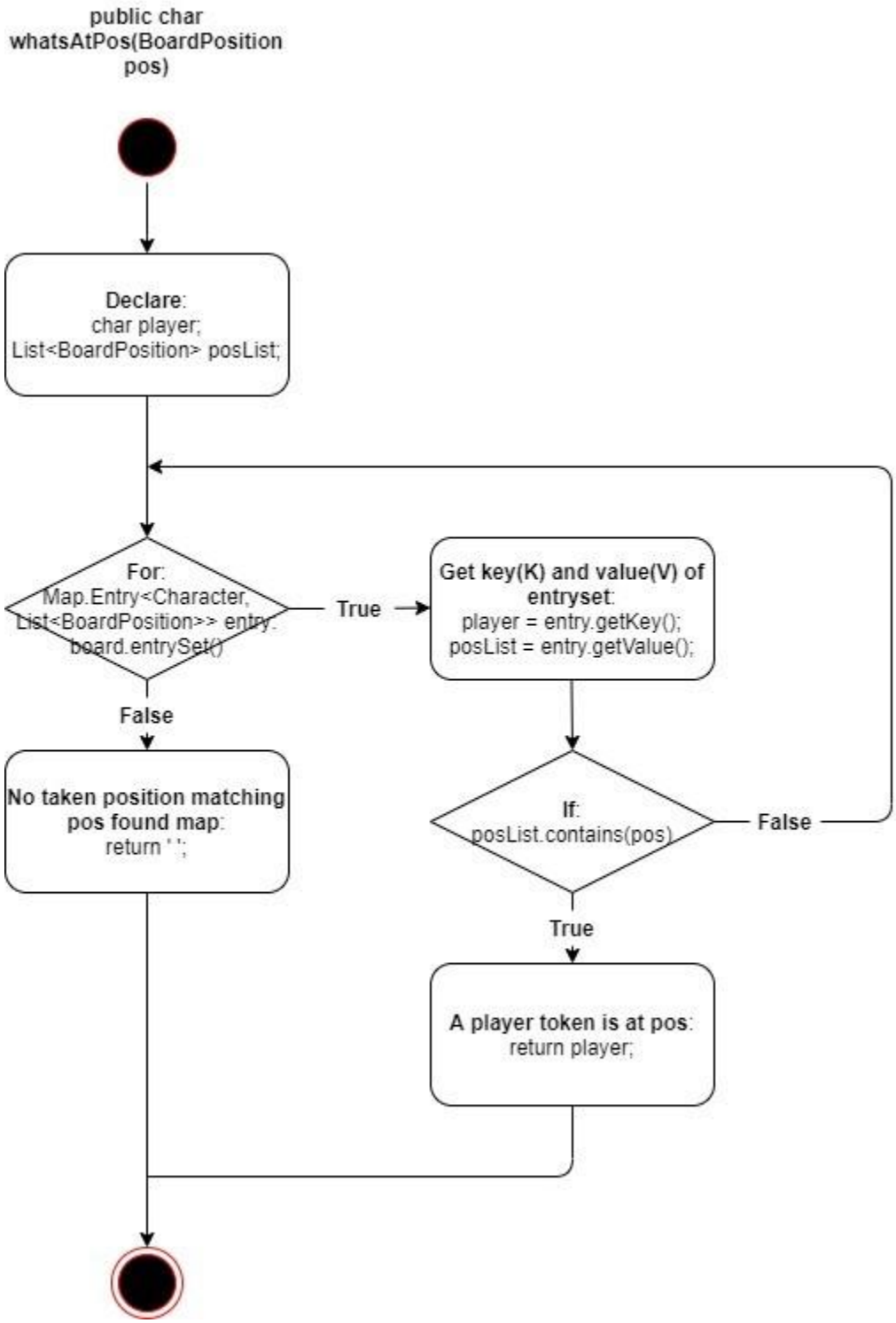

@Override public boolean checkForDraw()
If: openPos == 0;
True
There are no open positions: return true;
False
There are open positions: return false;
public int getNumToWin()
public int getNumColumns()
public int getNumRows()
Get number of tokens in a row needed to win: return NEEDED_TO_WIN;
Get number of columns: return MAX_COLUMN;
Get number of rows: return MAX_ROW;

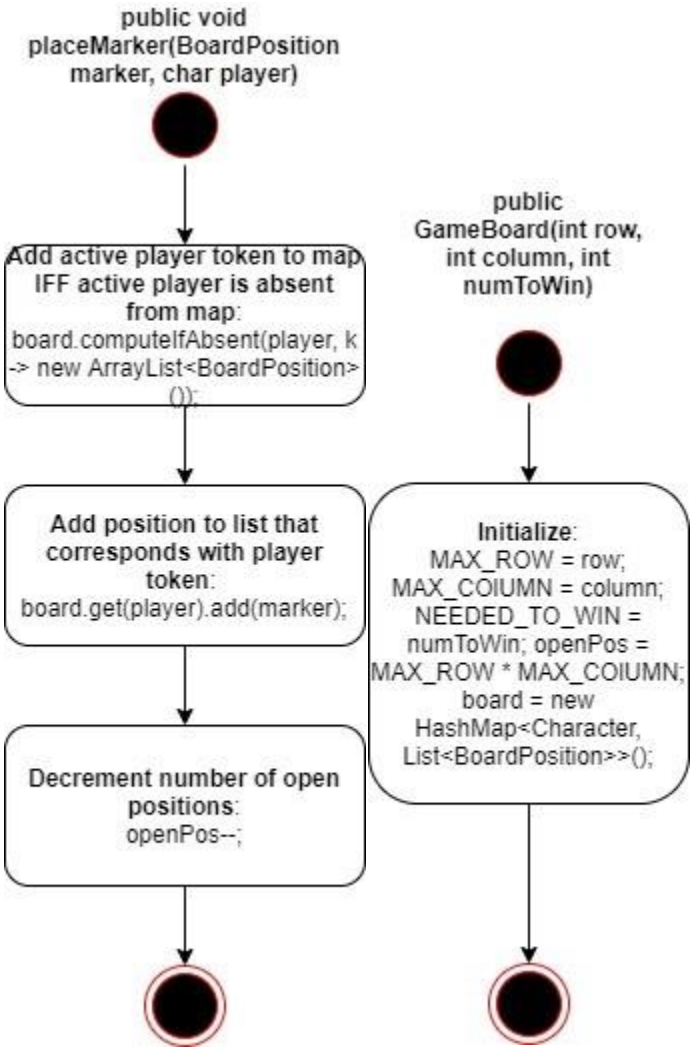

public void placeMarker(BoardPosition marker, char player)
If: checkSpace(marker)
True
Fill position with player token: board[marker.getRow()][marker.getColumn()] = player;
Decrement number of open positions: openPos--;
False
public char whatsAtPos(BoardPosition pos)
Return what is at pos: return board[pos.getRow()][pos.getColumn()];

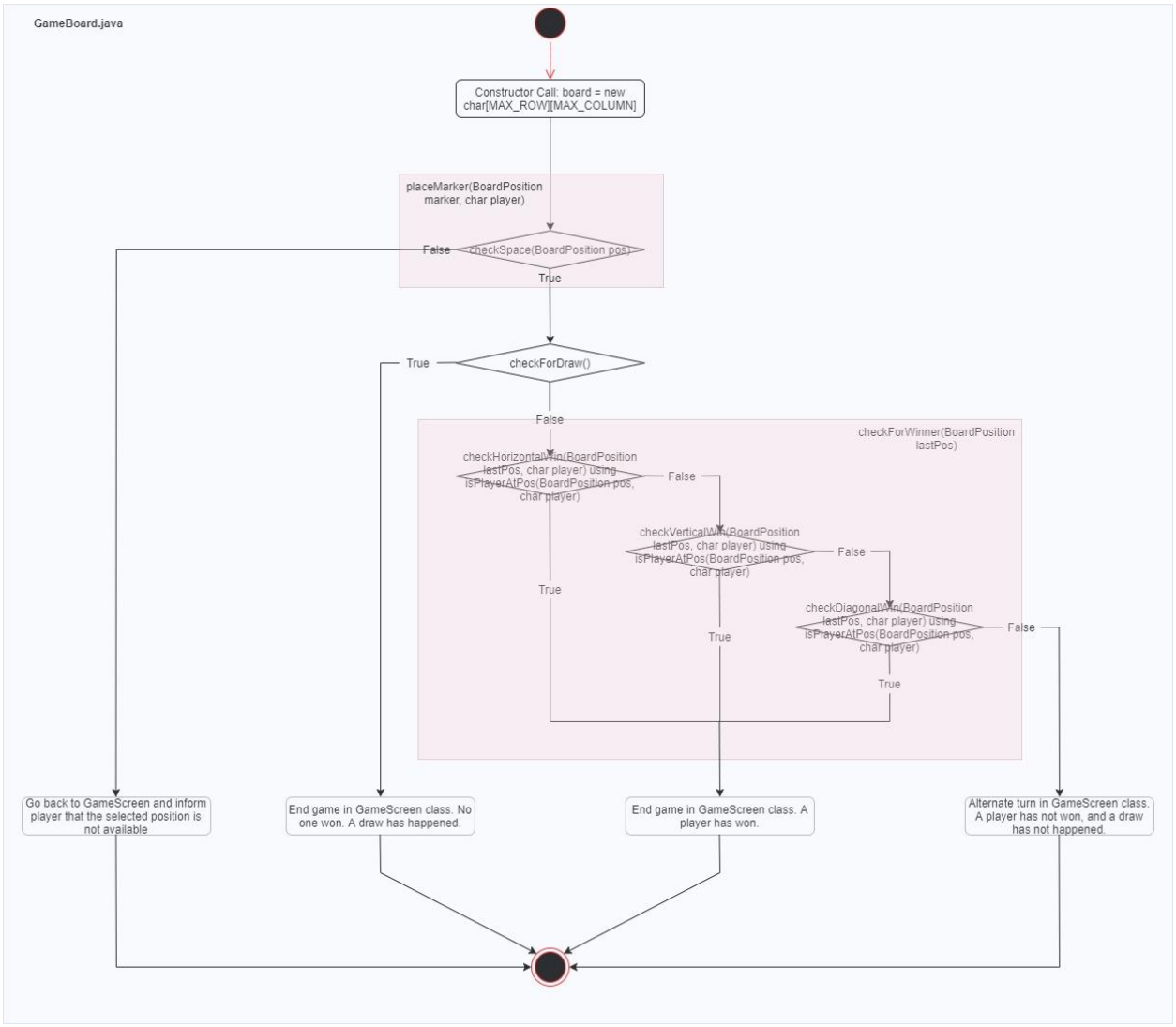

public GameBoard(int row, int column, int numToWin)
Initialize:
MAX_ROW = row;
MAX_COlUMN = column;
NEEDED_TO_WIN = numToWin; openPos = MAX_ROW * MAX_COlUMN;
board = new char[MAX_ROW] [MAX_COlUMN];
For:
int i = 0; i < MAX_ROW; i++
True
False
For:
int j = 0; j < MAX_ROW; j++
True
Fill position with empty space:
board[i][j] = ' ';
False


GameBoardMem
- MAX_ROW: int[1] {final}
- MAX_COLUMN: int[1] {final}
- NEEDED_TO_WIN: int[1] {final}
- openPos: int[1]
- board: Map<Character, List<BoardPosition>>[1]
+ GameBoard(): <Constructor>
+ getNumRows(): int
+ getNumColumns(): int
+ getNumToWin(): int
+ placeMarker(BoardPosition, char): void
+ checkForDraw(BoardPosition, char): boolean {Override}
+ whatsAtPos(BoardPosition): char


@Override public boolean checkForDraw()
If: openPos == 0;
True
There are no open positions: return true;
False
There are open positions: return false;
public int getNumToWin()
public int getNumColumns()
public int getNumRows()
Get number of tokens in a row needed to win: return NEEDED_TO_WIN;
Get number of columns: return MAX_COLUMN;
Get number of rows: return MAX_ROW;


public char whatsAtPos(BoardPosition pos)
Declare:
char player;
List<BoardPosition> posList;
For:
Map.Entry<Character, List<BoardPosition>> entry: board.entrySet()
True
Get key(K) and value(V) of entryset:
player = entry.getKey();
posList = entry.getValue();
False
No taken position matching pos found map:
return ' ';
If:
posList.contains(pos)
False
True
A player token is at pos:
return player;


public void placeMarker(BoardPosition marker, char player)
Add active player token to map IFF active player is absent from map:
board.computeIfAbsent(player, k -> new ArrayList<BoardPosition>());
Add position to list that corresponds with player token:
board.get(player).add(marker);
Decrement number of open positions:
openPos--;
public GameBoard(int row, int column, int numToWin)
Initialize:
MAX_ROW = row;
MAX_COlUMN = column;
NEEDED_TO_WIN = numToWin; openPos = MAX_ROW * MAX_COlUMN;
board = new HashMap<Character, List<BoardPosition>>();


GameBoard.java
Constructor Call: board = new char[MAX_ROW][MAX_COLUMN]
placeMarker(BoardPosition marker, char player)
checkSpace(BoardPosition pos)
False
True
checkForDraw()
True
False
checkForWinner(BoardPosition lastPos)
checkHorizontalWin(BoardPosition lastPos, char player) using isPlayerAtPos(BoardPosition pos, char player)
False
checkVerticalWin(BoardPosition lastPos, char player) using isPlayerAtPos(BoardPosition pos, char player)
False
True
checkDiagonalWin(BoardPosition lastPos, char player) using isPlayerAtPos(BoardPosition pos, char player)
False
True
True
Go back to GameScreen and inform player that the selected position is not available
End game in GameScreen class. No one won. A draw has happened.
End game in GameScreen class. A player has won.
Alternate turn in GameScreen class. A player has not won, and a draw has not happened.