MATTHEW FRANCHI

SOFTWARE DEVELOPMENT FOUNDATIONS

MARCH 25, 2020

ConnectX Project Requirements Report

CONTENTS

PROJECT OVERVIEW

FUNCTIONAL REQUIREMENTS

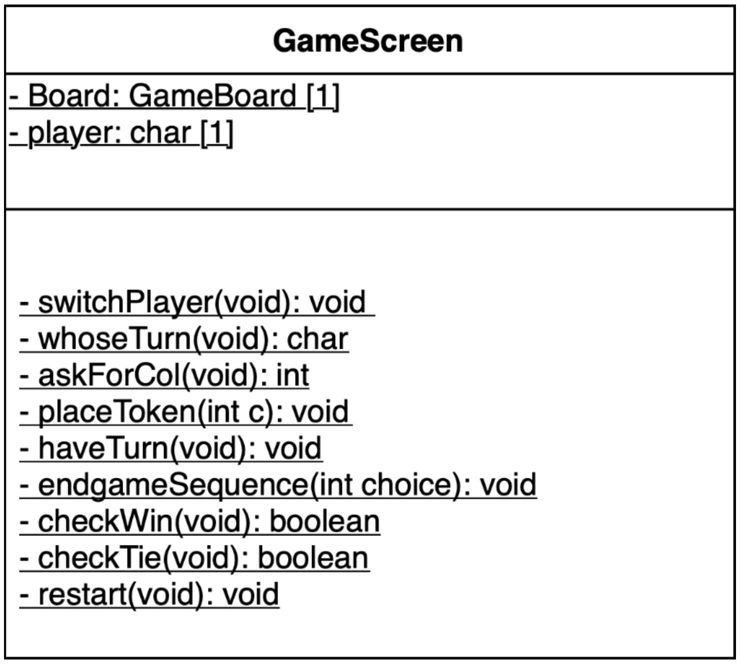
I. As a user, I can place my token in any available column, so that I can play the game
II. As a user, I can set the number of rows on the board, so that I have the ability to customize my game
III. As a user, I can set the number of columns on the board, so that I have the ability to customize my game
IV. As a user, I can set the (number to win) tokens, so that I have the ability to customize my game
V. As a user, I can set the number of players, so that I have the ability to customize my game
VI. As a user, I can set the token for each player, so that I know who is who
VII. As a user, I can place (number to win) tokens in vertical order
VIII. As a user, I can place (number to win) tokens in horizontal order
IX. As a user, I can place (number to win) tokens in diagonal order
X. As a user, I can see whose turn it is before each play, so that I know when it is and isn't my turn to play.
XI. As a user, I can see the current state of the board after each play, so that I can plan my next move.
XII. As a user, I can start a new game after the current one ends, so that I can have another chance to play from the beginning.
XIII. As a user, I can try to place four of my tokens in the same row, column, or diagonal order consecutively, so that I can win the game.
XIV. As a user, I can see if the game ended in a win
XV. As a user, I can see if the game ended in a tie
XVI. As a user, I can only enter my tokens in columns with space available, so that I can follow the rules of the game.
XVII. As a user, I can only enter tokens in the first available row in a column, so that I do not overwrite existing tokens.
XVIII. As a user, if I am the first one to specify a player token, then I will go first.
XIX. As a user, if someone wins the game, I can see the winning board.
XX. As a user, I can choose to not play a new game
XXI. As a user, I can choose to play a new game
XXII. As a user, I can choose the fast implementation / mode
XXIII. As a user, I can choose the memory-efficient implementation / mode
XXIV. As a user, I can be prompted to enter a new column if my first choice is invalid
XXV. As a user, I can be prompted to enter a new player token if my first choice was already taken
XXVI. As a user, I can be prompted to enter a new number of rows if my first input was out of range
XXVII. As a user, I can be prompted to enter a new number of columns if my first input was out of range

NON-FUNCTIONAL REQUIREMENTS

I. Must be implemented with the Java coding language
II. Must be able to run on the Clemson School of Computing Unix Environment
III. Must be able to run on a command-line interface
IV. Must be completely reliable; no crashes mid-game, when starting a new game, etc.
V. There should be a minimal, unnoticeable processing time between each turn
VI. The GameBoard and BoardPosition classes must follow the exact method signatures specified in the project guidelines document.
VII. The GameScreen class is the only class that can get input from the user or print to the console.
VIII. The project should have a high degree of adaptability and modularity, so that future additions are less complicated and easier.
IX. The project should keep the contents of the board private, as to avoid tampering.
X. The game should be extremely easy to play, and straightforward – in other words, someone with no prior experience with Connect4 (X) should be able to play the game.
XI. The project should be compiled using a makefile.
XII. Any prompts for user input should be clear and easy to understand
XIII. The game board must be an upright grid
XIV. All code must follow all best practices discussed in class
XV. All function signatures specified in the requirements document should be followed exactly
XVI. The number of rows on the board is greater than 3
XVII. The number of rows on the board is less than 100
XVIII. The number of columns on the board is greater than 3
XIX. The number of columns on the board is less than 100
XX. The number of tokens needed to win is greater than or equal to 3
XXI. The number of tokens needed to win is less than or equal to 25
XXII. The number of players is greater than or equal to 2
XXIII. The number of players is less than or equal to 10
XXIV. ConnectX should have a fast implementation
XXV. ConnectX should have a memory-efficient implementation
XXVI. The game should work with 2 to 10 players
XXVII. The program should not have any magic numbers
XXVIII. The memory-efficient (Map) implementation should not create keys for the blank space [‘ ‘]
XXIX. The game should not allow for a number of tokens needed to win greater than the number of rows
XXX. The game should not allow for a number of tokens needed to win greater than the number of columns
XXXI. The program should have descriptive comments
XXXII. The program should follow the principles of design by contract, utilizing Javadoc comments

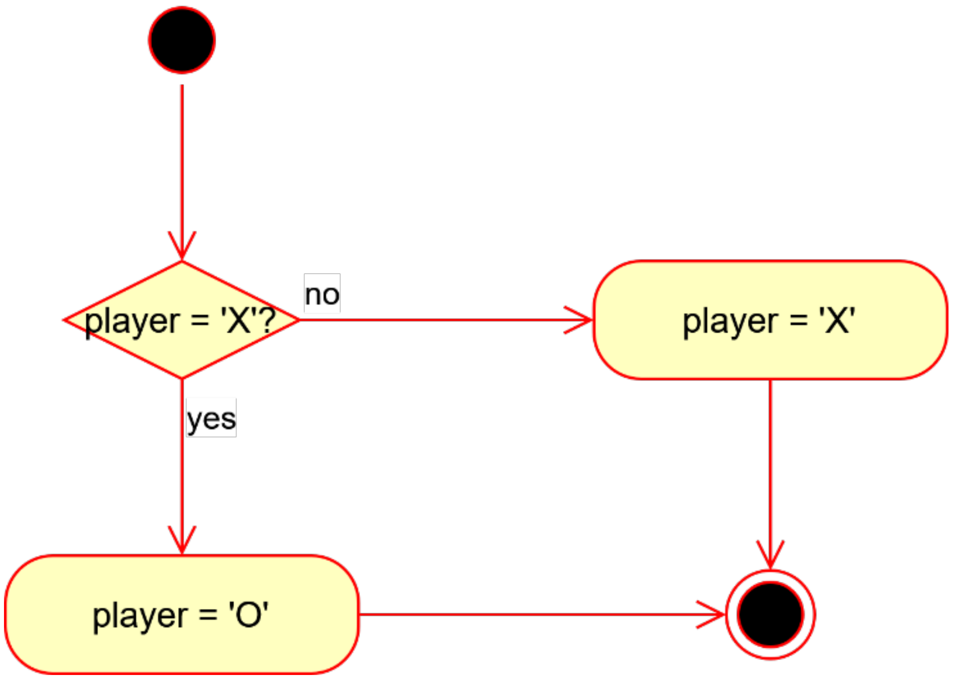
GAMESCREEN CLASS

UML CLASS DIAGRAM

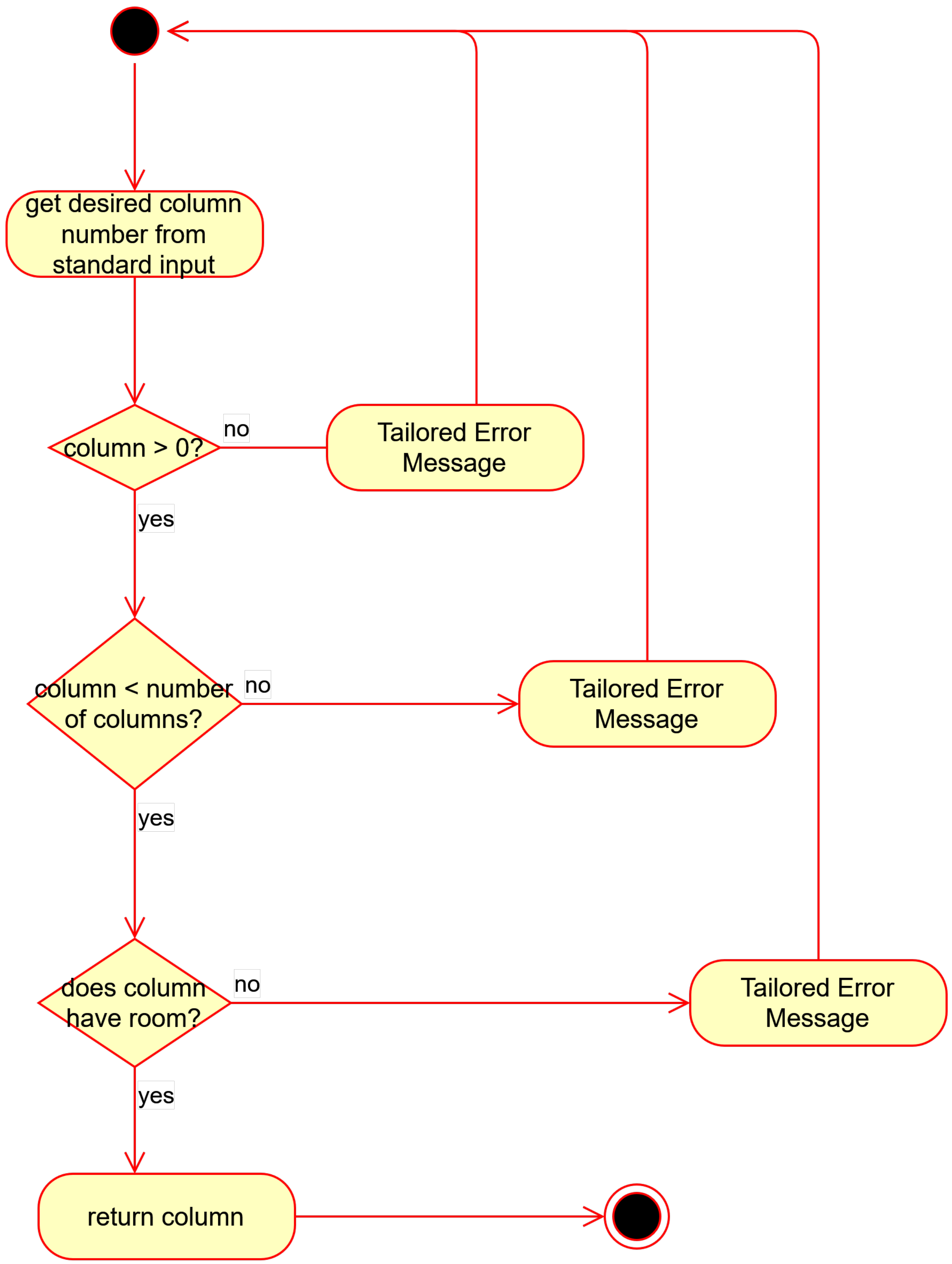


UML ACTIVITY DIAGRAMS

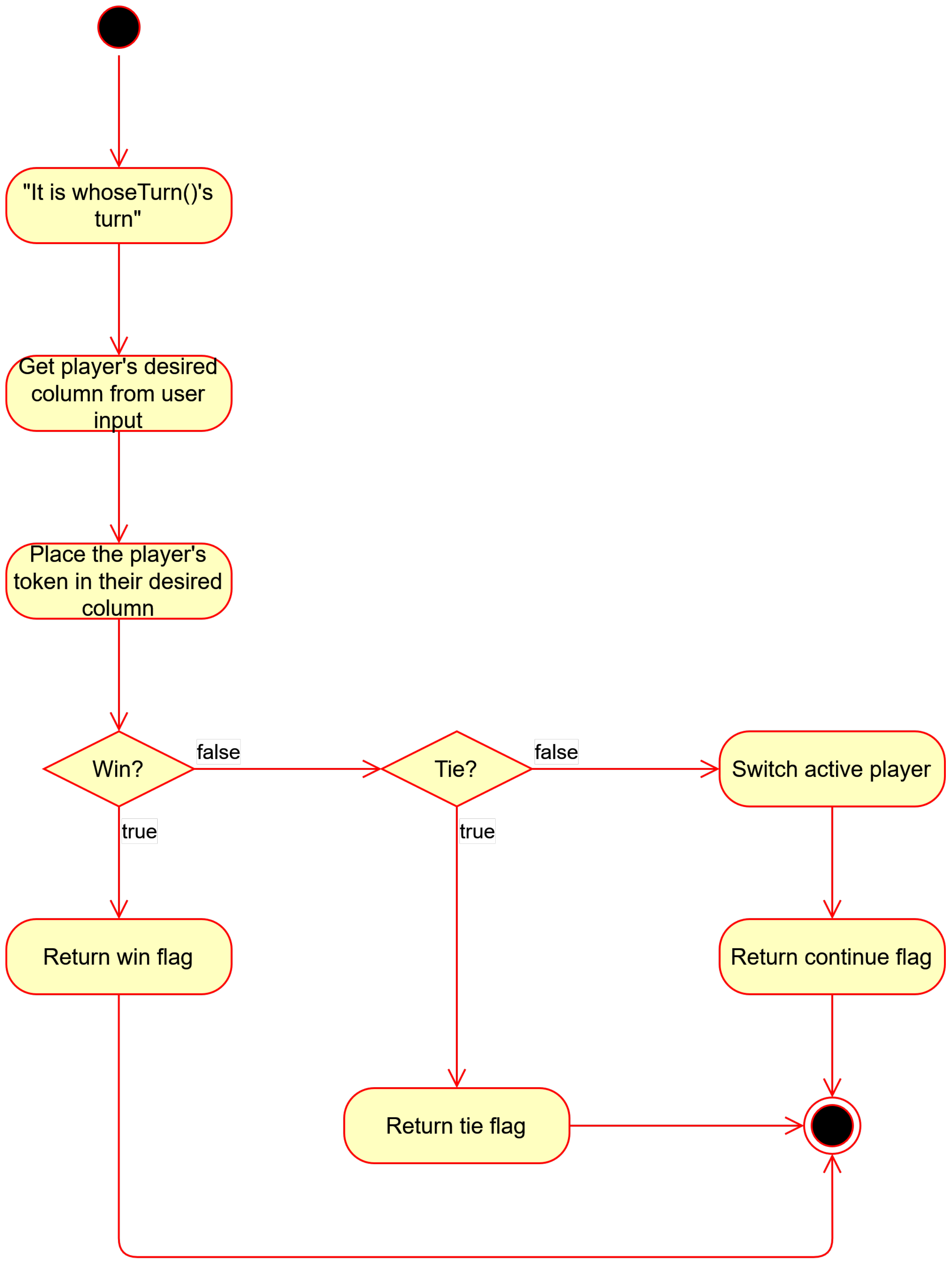
SWITCHPLAYER



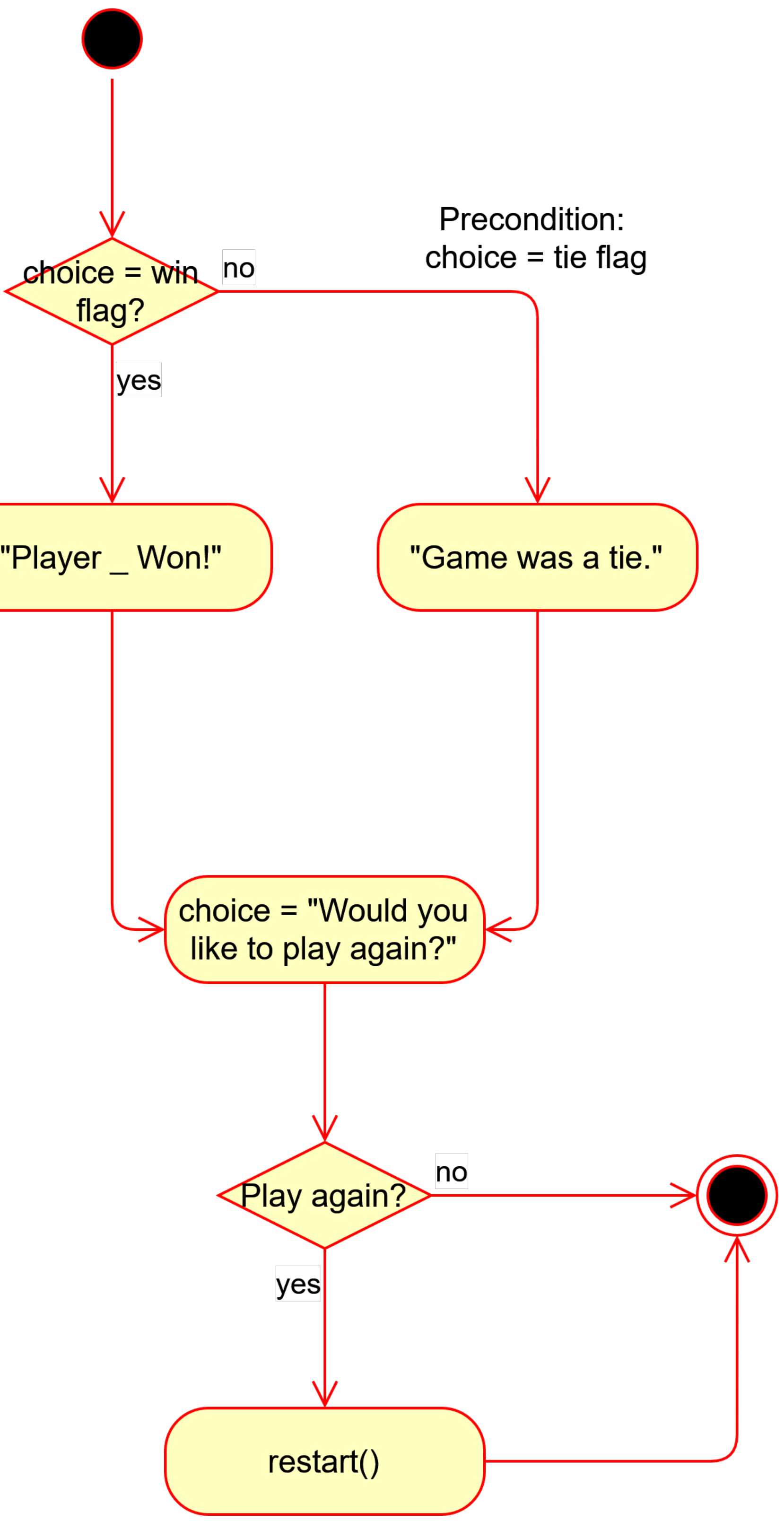
ASKFORCOL


get desired column number from standard input
column > 0?
no
yes
Tailored Error Message
column < number of columns?
no
yes
Tailored Error Message
does column have room?
no
yes
Tailored Error Message
return column

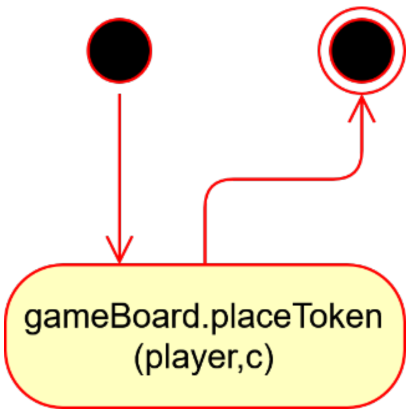
HAVETURN


"It is whoseTurn()'s turn"
Get player's desired column from user input
Place the player's token in their desired column
Win?
false
true
Tie?
false
true
Switch active player
Return win flag
Return continue flag
Return tie flag

ENDGAMESEQUENCE

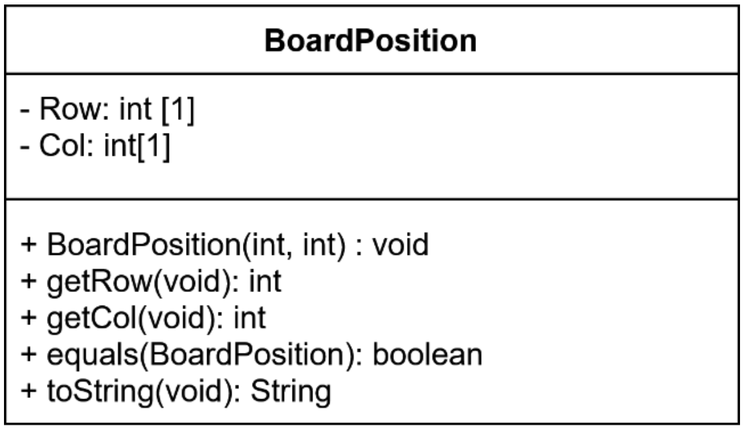


PLACETOKEN



BOARDPOSITION CLASS

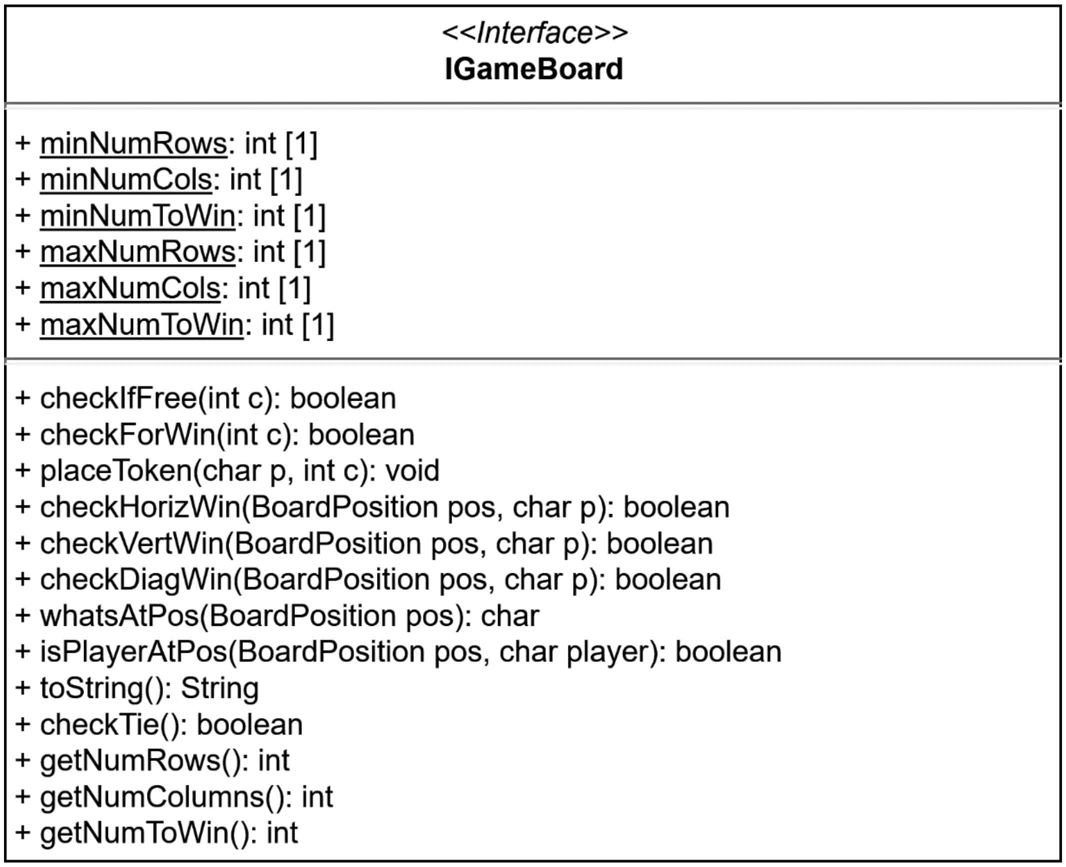
UML CLASS DIAGRAM



BoardPosition
- Row: int [1] - Col: int[1]
+ BoardPosition(int, int) : void + getRow(void): int + getCol(void): int + equals(BoardPosition): boolean + toString(void): String

IGAMEBOARD INTERFACE

UML INTERFACE DIAGRAM



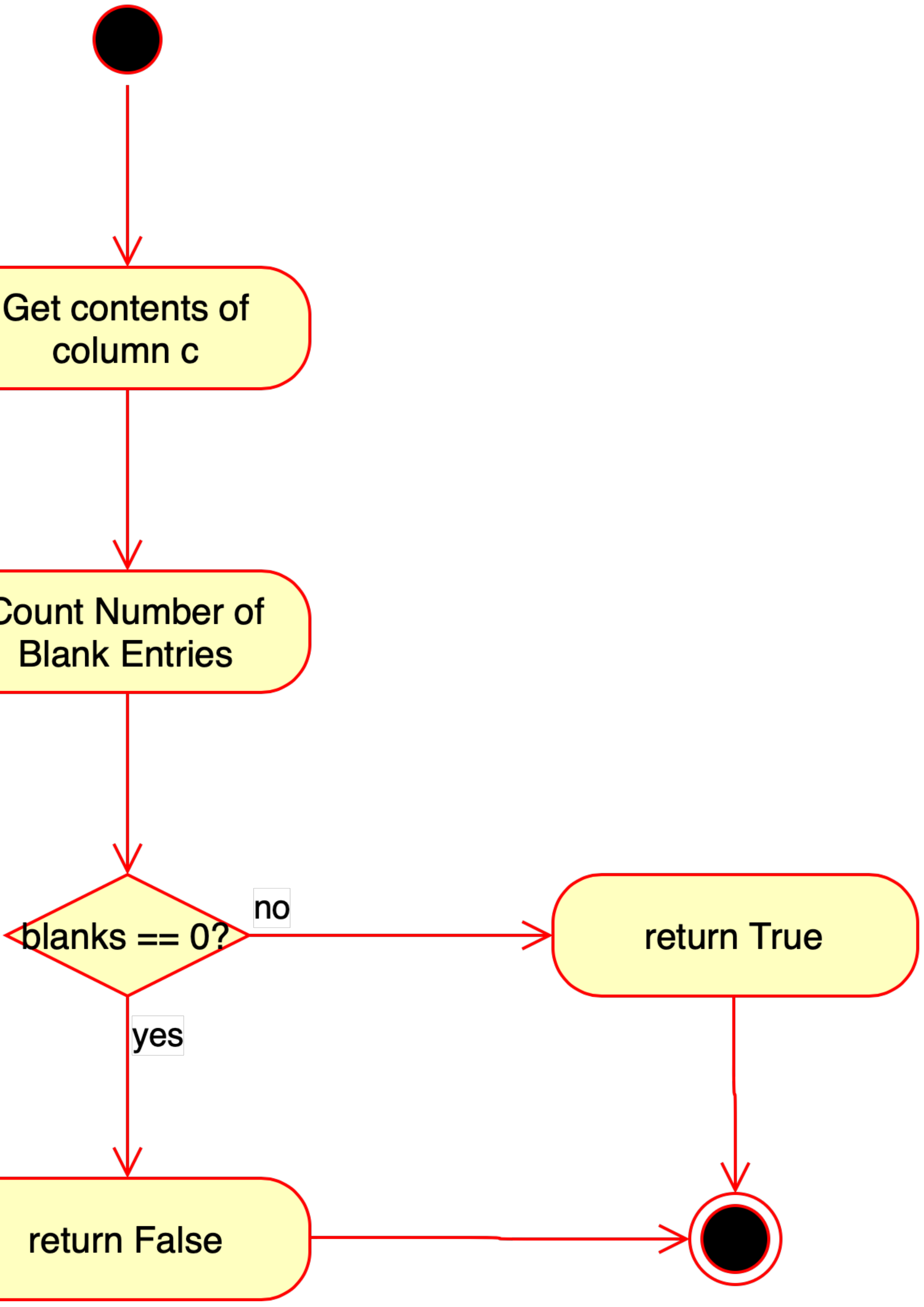
<<Interface>>
IGameBoard

```
+ minNumRows: int [1]
+ minNumCols: int [1]
+ minNumToWin: int [1]
+ maxNumRows: int [1]
+ maxNumCols: int [1]
+ maxNumToWin: int [1]
```

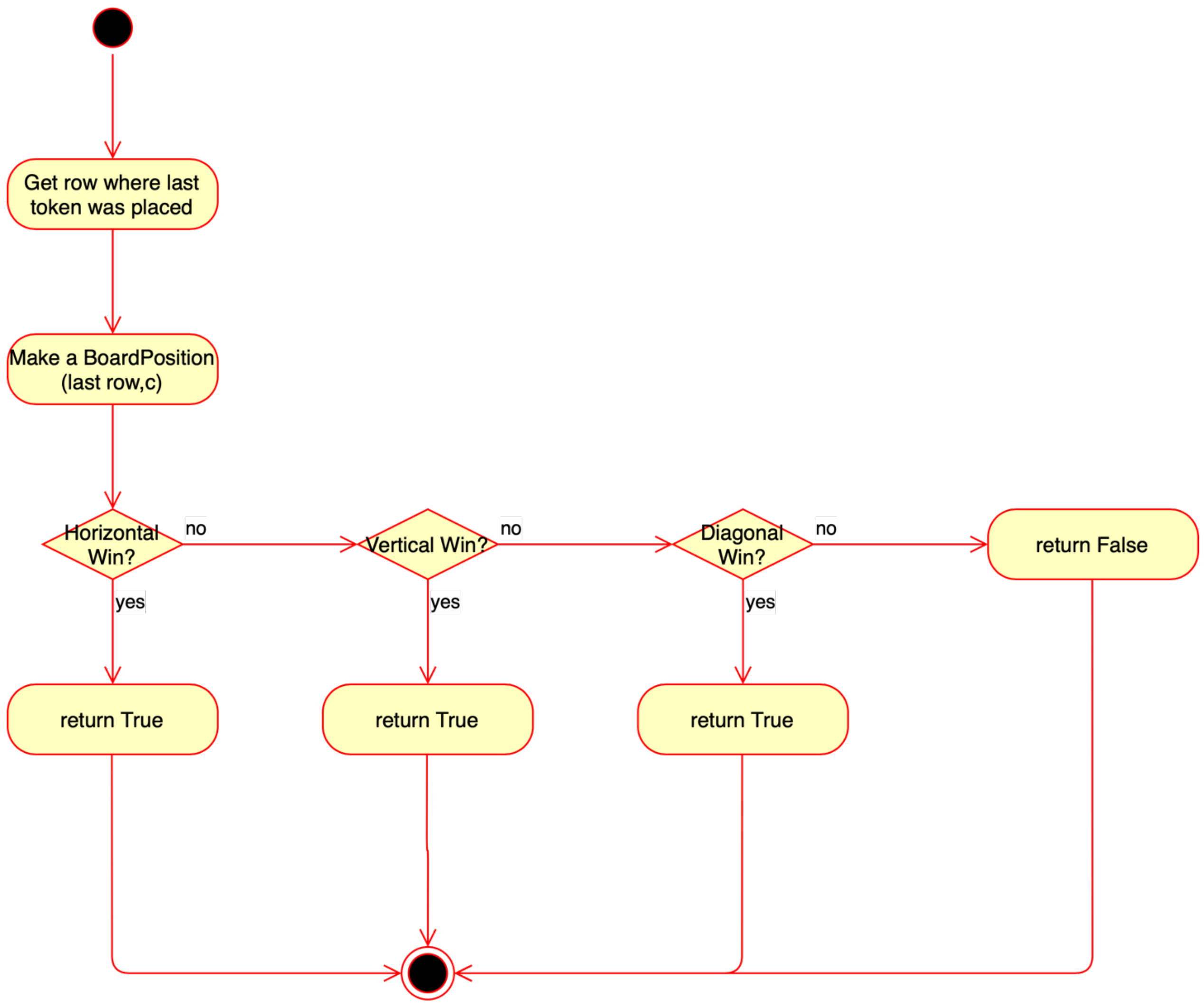
```
+ checkIfFree(int c): boolean
+ checkForWin(int c): boolean
+ placeToken(char p, int c): void
+ checkHorizWin(BoardPosition pos, char p): boolean
+ checkVertWin(BoardPosition pos, char p): boolean
+ checkDiagWin(BoardPosition pos, char p): boolean
+ whatsAtPos(BoardPosition pos): char
+ isPlayerAtPos(BoardPosition pos, char player): boolean
+ toString(): String
+ checkTie(): boolean
+ getNumRows(): int
+ getNumColumns(): int
+ getNumToWin(): int
```

ACTIVITY DIAGRAMS

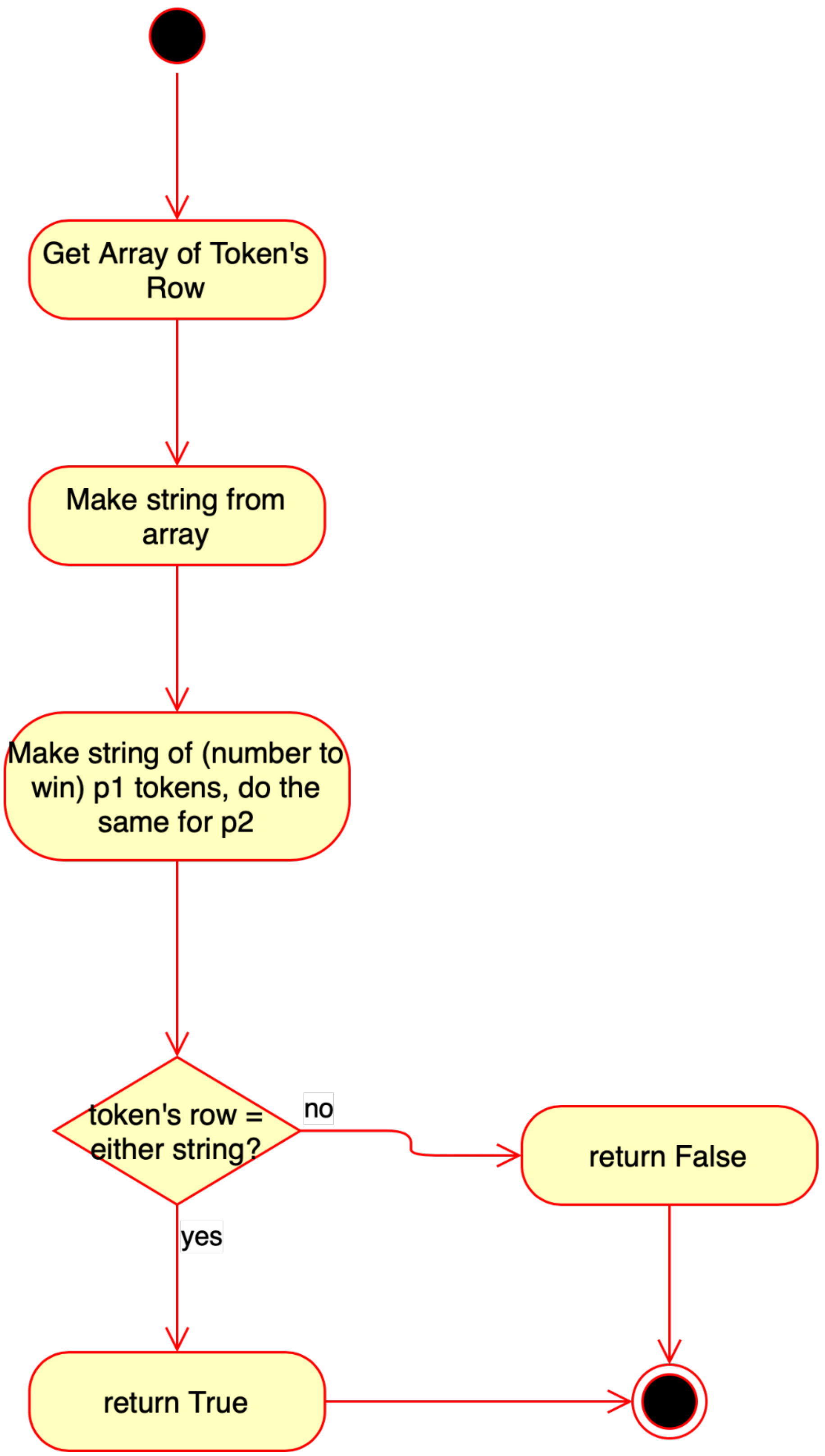
CHECKIFFREE



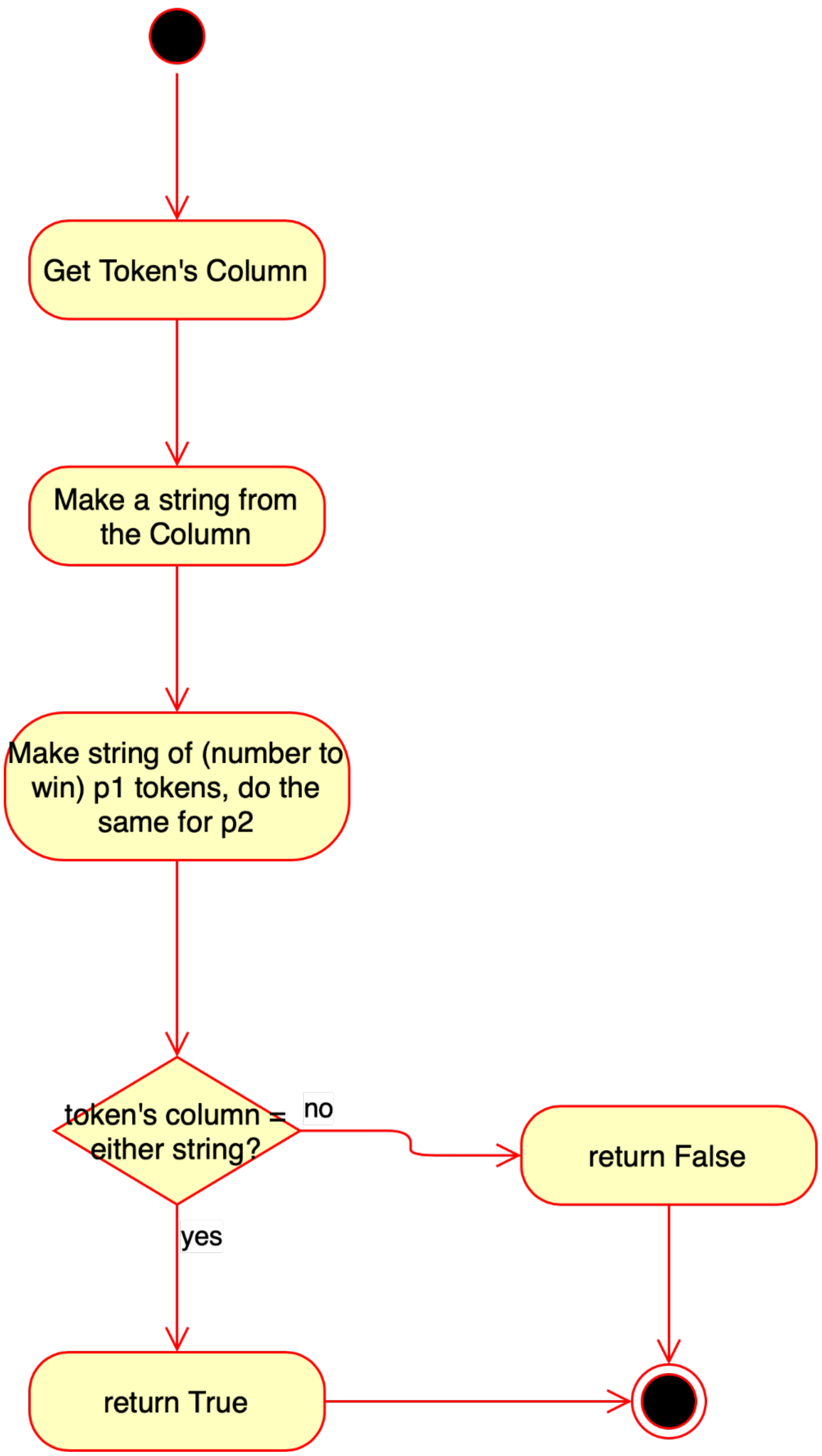
CHECKFORWIN



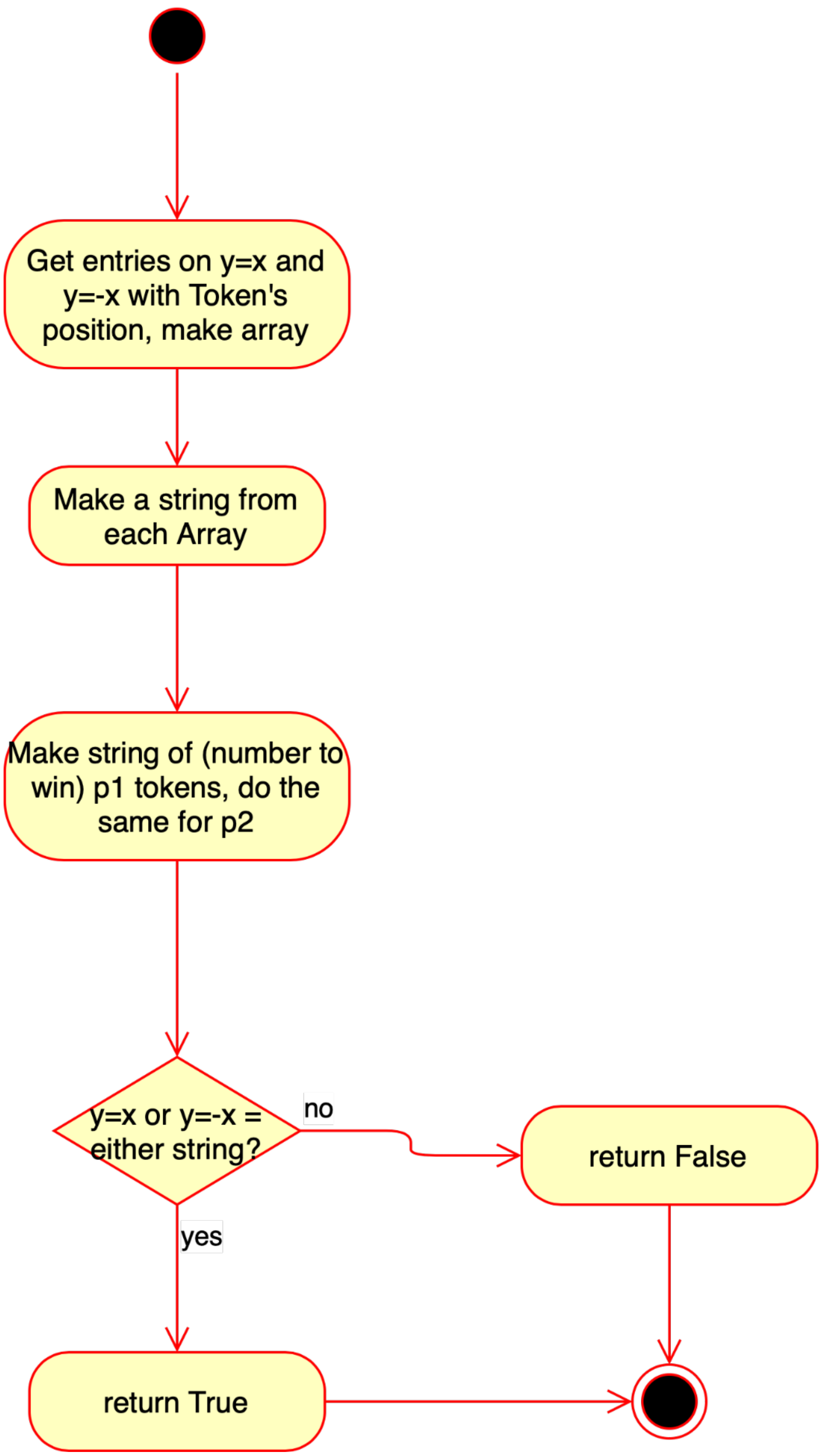
CHECKHORIZWIN


Get Array of Token's Row
Make string from array
Make string of (number to win) p1 tokens, do the same for p2
token's row = either string?
no
return False
yes
return True

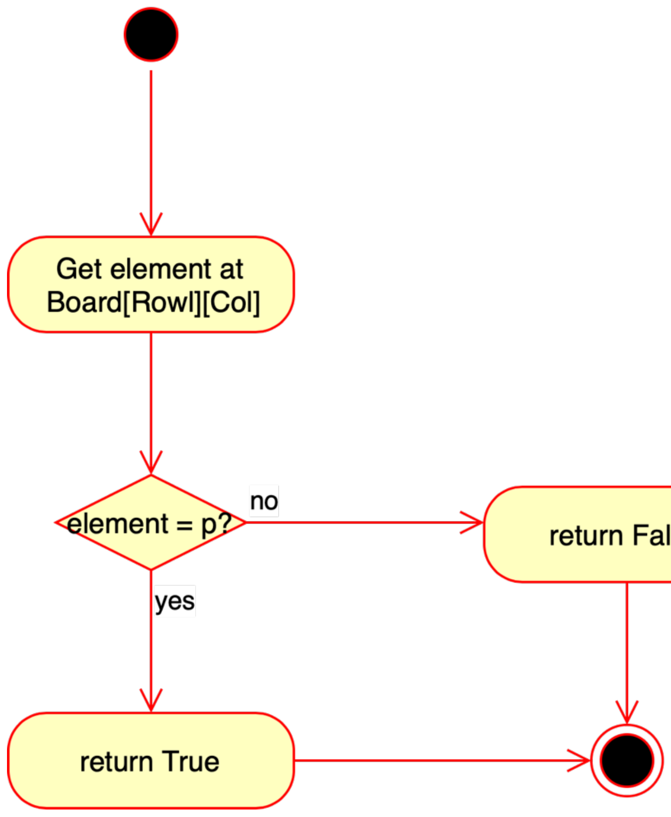
CHECKVERTWIN


Get Token's Column
Make a string from the Column
Make string of (number to win) p1 tokens, do the same for p2
token's column = either string?
no
return False
yes
return True

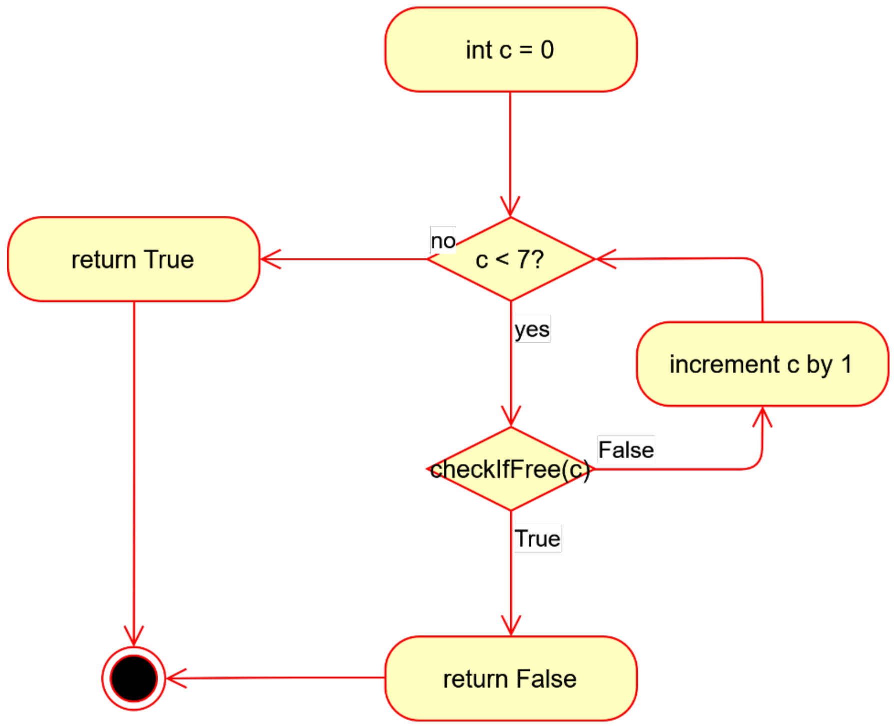
CHECKDIAGWIN


Get entries on y=x and y=-x with Token's position, make array
Make a string from each Array
Make string of (number to win) p1 tokens, do the same for p2
y=x or y=-x = either string?
no
return False
yes
return True

ISPLAYERATPOS

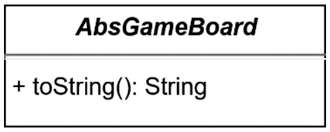


CHECKTIE



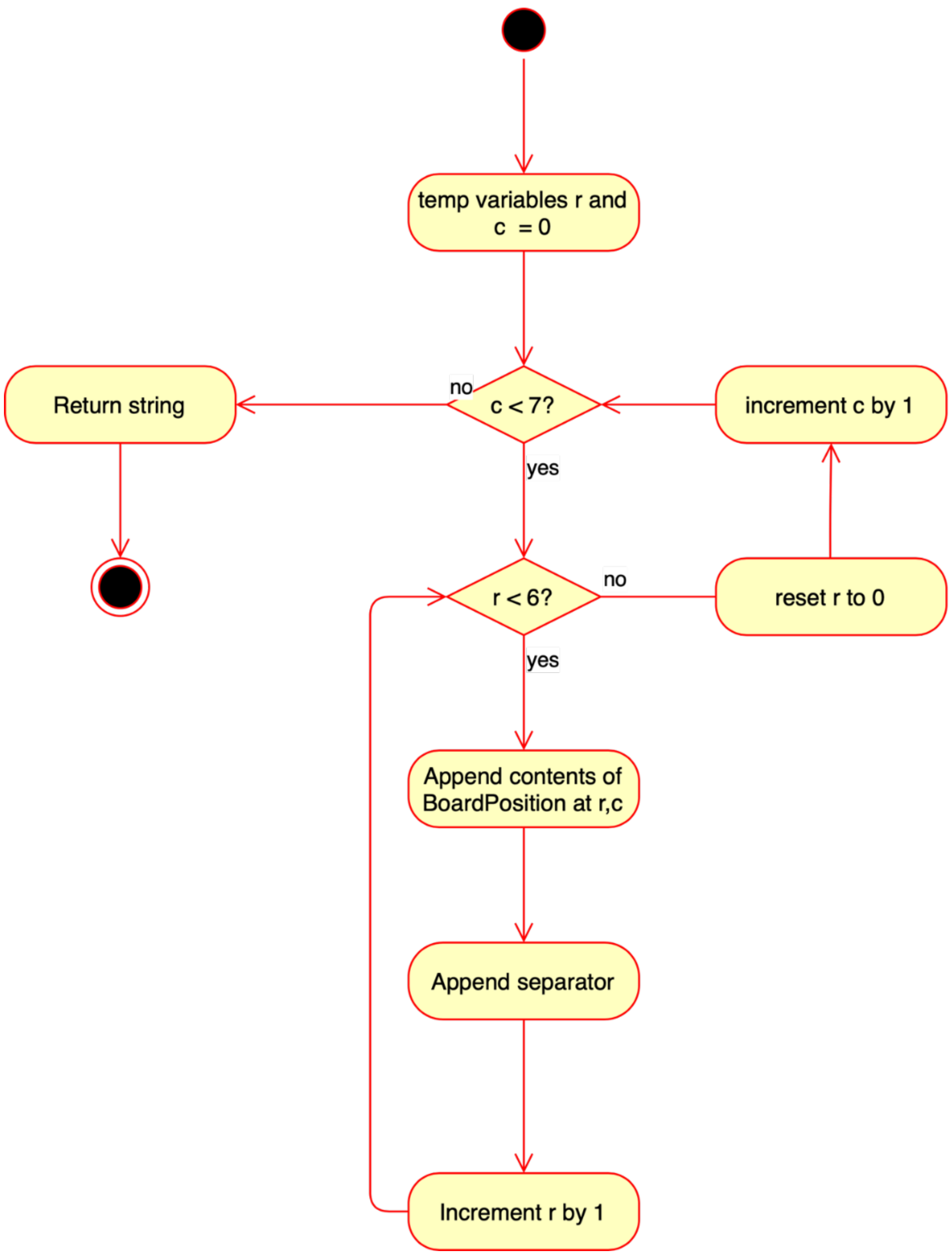
ABSGAMEBOARD CLASS

UML CLASS DIAGRAM



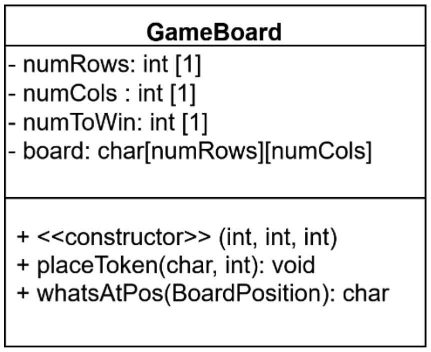
UML ACTIVITY DIAGRAM

TOSTRING



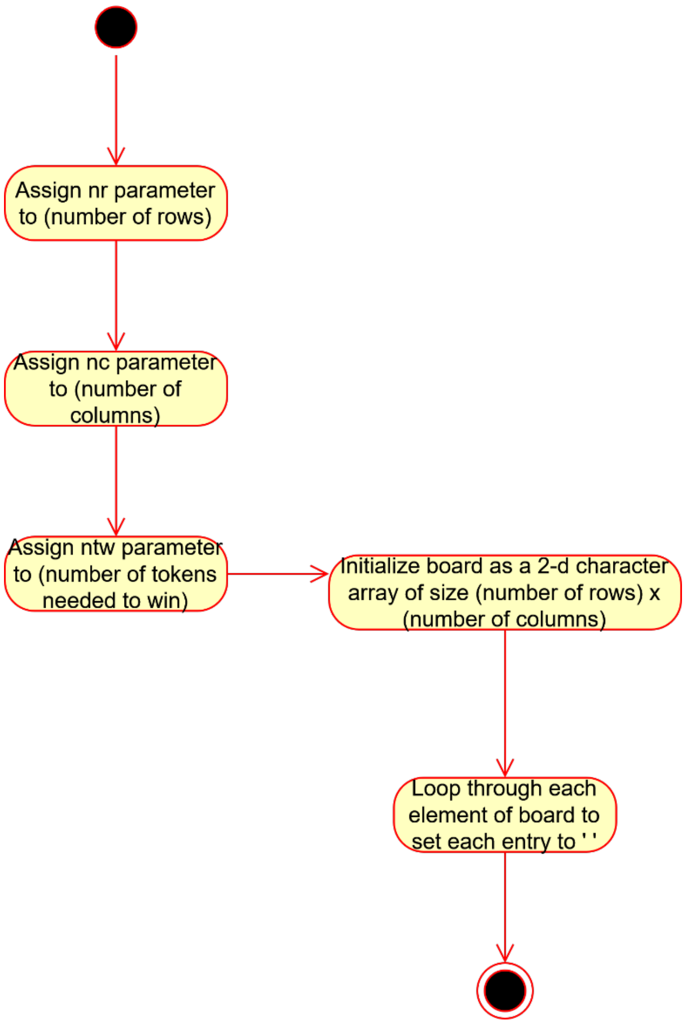
GAMEBOARD CLASS

UML CLASS DIAGRAM

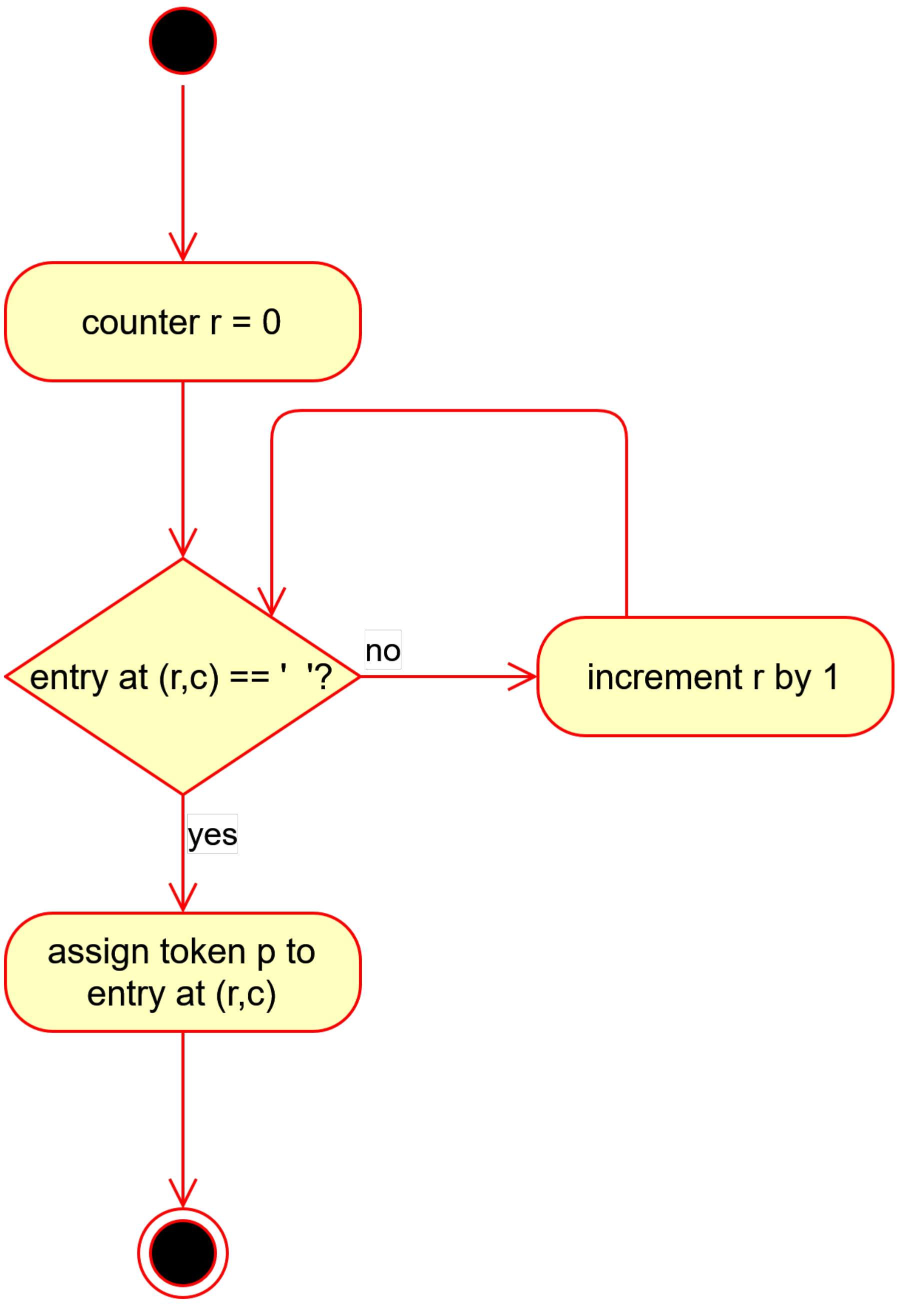

GameBoard
- numRows: int [1]
- numCols : int [1]
- numToWin: int [1]
- board: char[numRows][numCols]
+ <<constructor>> (int, int, int)
+ placeToken(char, int): void
+ whatsAtPos(BoardPosition): char

UML ACTIVITY DIAGRAMS

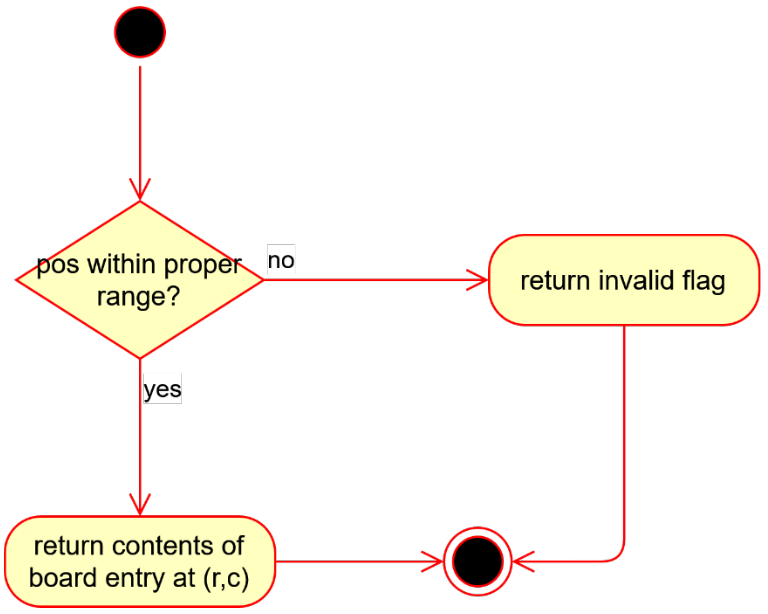
CONSTRUCTOR


Assign nr parameter to (number of rows)
Assign nc parameter to (number of columns)
Assign ntw parameter to (number of tokens needed to win)
Initialize board as a 2-d character array of size (number of rows) x (number of columns)
Loop through each element of board to set each entry to ' '

PLACETOKEN

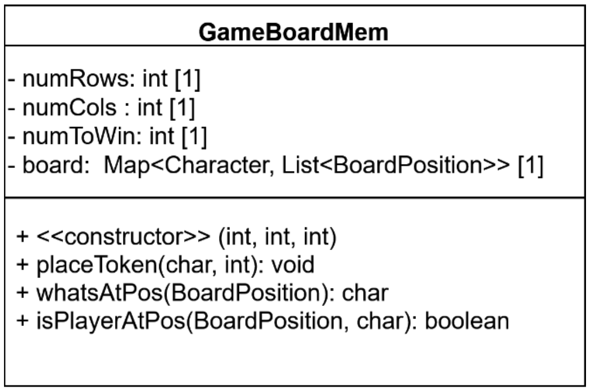

counter r = 0
entry at (r,c) == ' '?
no
increment r by 1
yes
assign token p to entry at (r,c)

WHATSATPOS



GAMEBOARDMEM CLASS

UML CLASS DIAGRAM



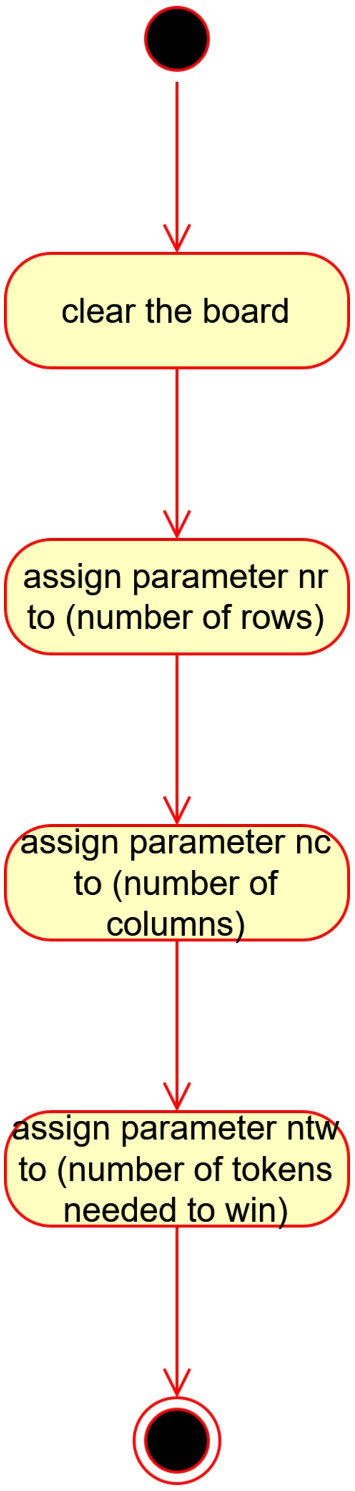
GameBoardMem

- numRows: int [1]
- numCols : int [1]
- numToWin: int [1]
- board: Map<Character, List<BoardPosition>> [1]

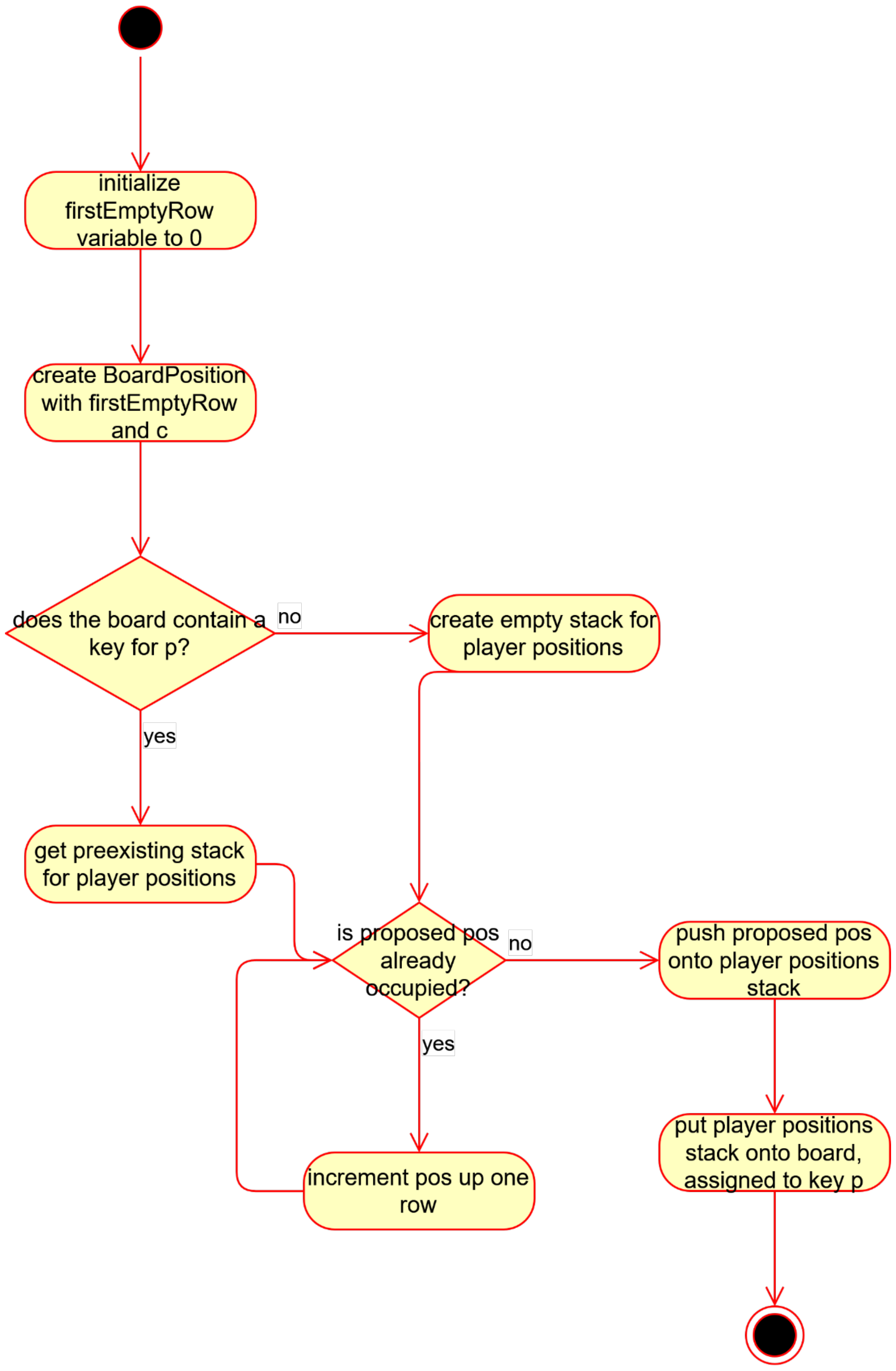
+ <<constructor>> (int, int, int)
+ placeToken(char, int): void
+ whatsAtPos(BoardPosition): char
+ isPlayerAtPos(BoardPosition, char): boolean

UML ACTIVITY DIAGRAMS

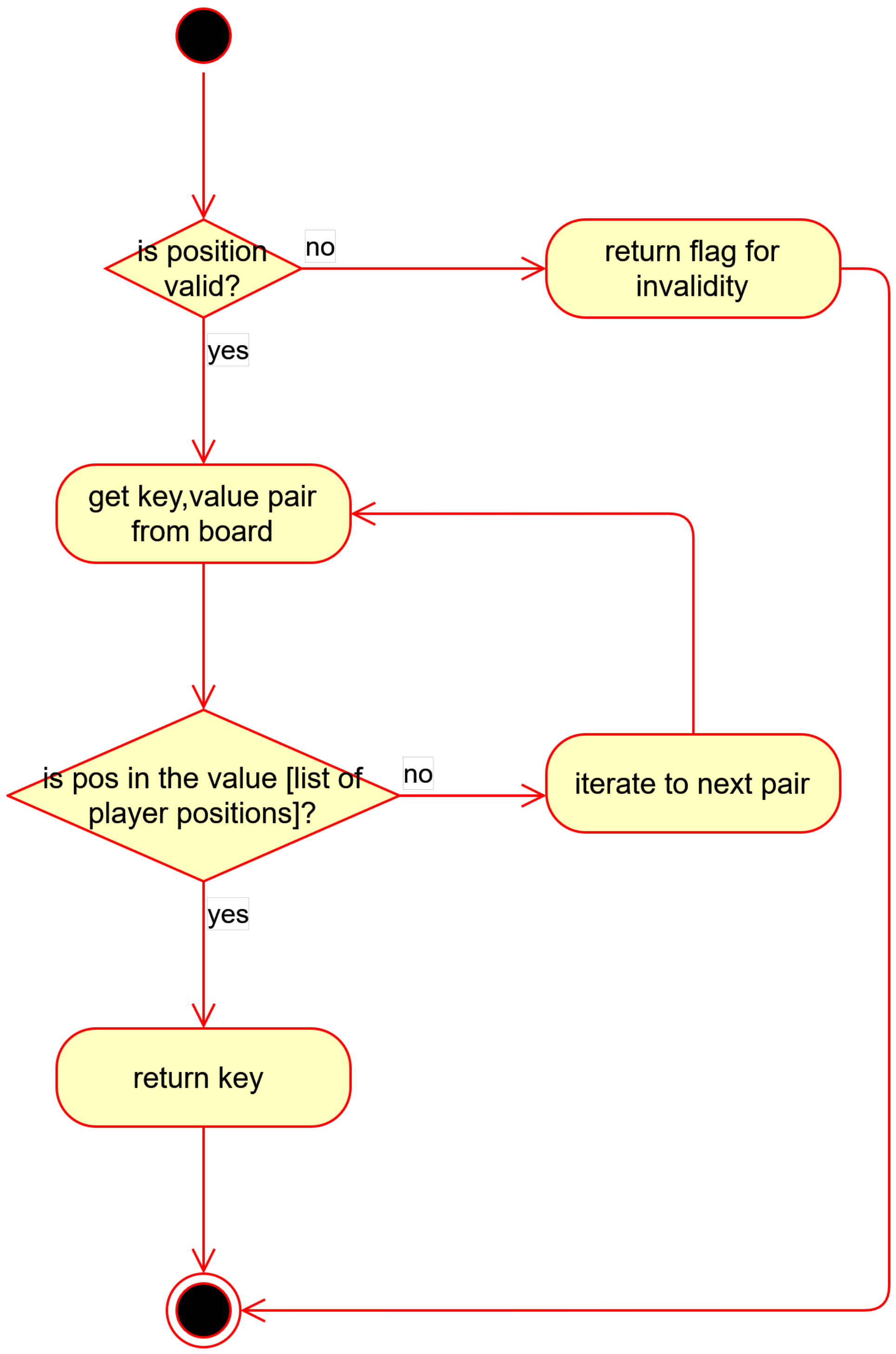
CONSTRUCTOR


clear the board
assign parameter nr to (number of rows)
assign parameter nc to (number of columns)
assign parameter ntw to (number of tokens needed to win)

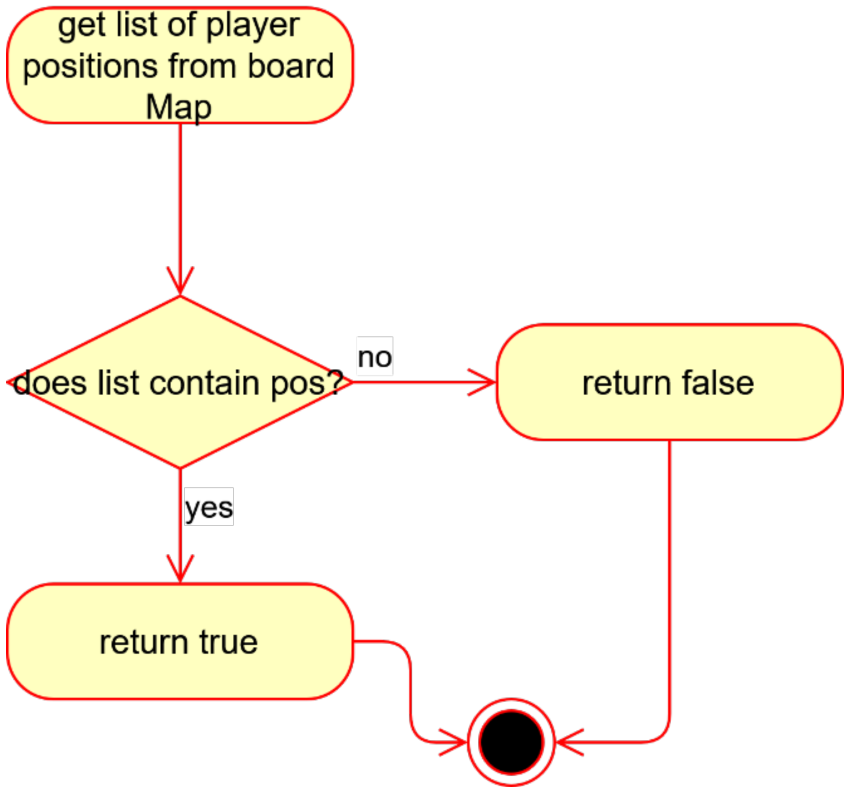
PLACETOKEN


initialize firstEmptyRow variable to 0
create BoardPosition with firstEmptyRow and c
does the board contain a key for p?
no
create empty stack for player positions
yes
get preexisting stack for player positions
is proposed pos already occupied?
no
push proposed pos onto player positions stack
yes
increment pos up one row
put player positions stack onto board, assigned to key p

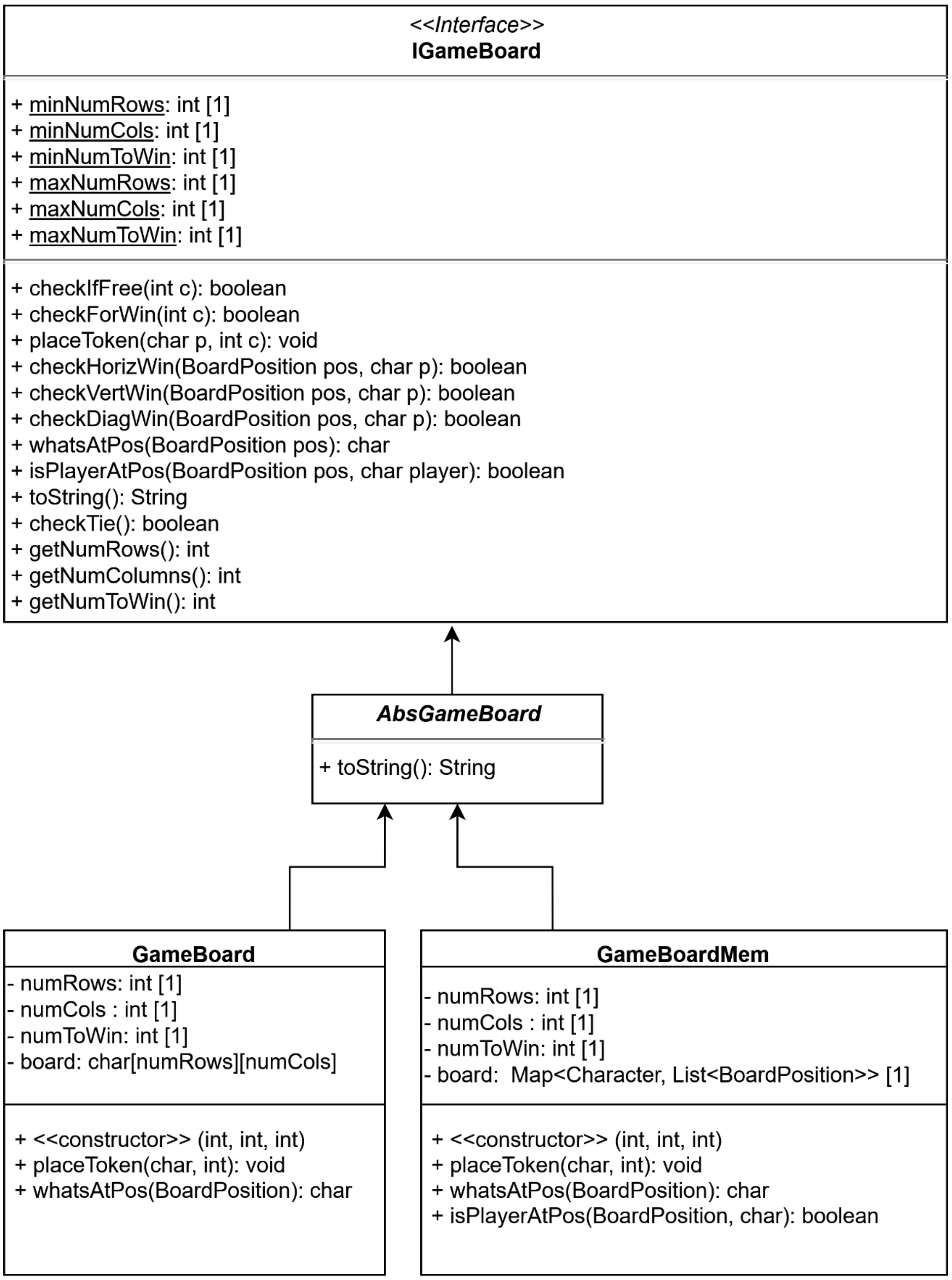
WHATSATPOS


is position valid?
no
return flag for invalidity
yes
get key,value pair from board
is pos in the value [list of player positions]?
no
iterate to next pair
yes
return key

ISPLAYERATPOS



UML CLASS-RELATIONSHIPS DIAGRAM



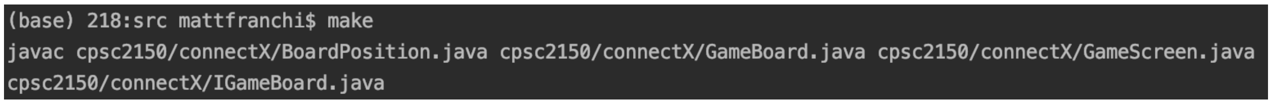
<<Interface>>
IGameBoard
+ minNumRows: int [1]
+ minNumCols: int [1]
+ minNumToWin: int [1]
+ maxNumRows: int [1]
+ maxNumCols: int [1]
+ maxNumToWin: int [1]
+ checkIfFree(int c): boolean
+ checkForWin(int c): boolean
+ placeToken(char p, int c): void
+ checkHorizWin(BoardPosition pos, char p): boolean
+ checkVertWin(BoardPosition pos, char p): boolean
+ checkDiagWin(BoardPosition pos, char p): boolean
+ whatsAtPos(BoardPosition pos): char
+ isPlayerAtPos(BoardPosition pos, char player): boolean
+ toString(): String
+ checkTie(): boolean
+ getNumRows(): int
+ getNumColumns(): int
+ getNumToWin(): int
AbsGameBoard
+ toString(): String
GameBoard
- numRows: int [1]
- numCols : int [1]
- numToWin: int [1]
- board: char[numRows][numCols]
+ <<constructor>> (int, int, int)
+ placeToken(char, int): void
+ whatsAtPos(BoardPosition): char
GameBoardMem
- numRows: int [1]
- numCols : int [1]
- numToWin: int [1]
- board: Map<Character, List<BoardPosition>> [1]
+ <<constructor>> (int, int, int)
+ placeToken(char, int): void
+ whatsAtPos(BoardPosition): char
+ isPlayerAtPos(BoardPosition, char): boolean

PROJECT COMPILING INSTRUCTIONS

ConnectX comes bundles with a GNU makefile that provides three functionalities:

MAKE DEFAULT

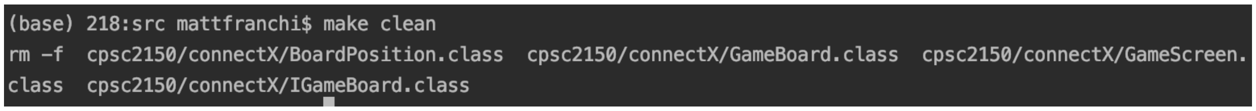
The default routine compiles all the project's .java files into .class files.



```
(base) 218:src mattfranchi$ make
javac cpsc2150/connectX/BoardPosition.java cpsc2150/connectX/GameBoard.java cpsc2150/connectX/GameScreen.java
cpsc2150/connectX/IGameBoard.java
```

MAKE RUN

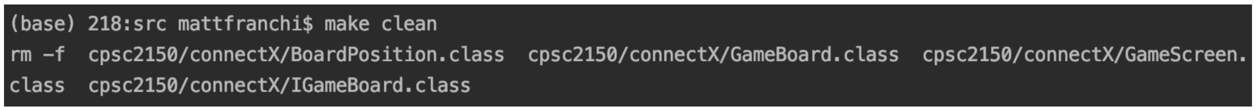
The run command executes the project's GameScreen class, which starts the ConnectX game. NOTE: the *default make* command needs to be run before *make run*.



```
(base) 218:src mattfranchi$ make clean
rm -f  cpsc2150/connectX/BoardPosition.class  cpsc2150/connectX/GameBoard.class  cpsc2150/connectX/GameScreen.
class  cpsc2150/connectX/IGameBoard.class
```

MAKE CLEAN

The clean command deletes all .class files in the project directory; NOTE: the code will have to be recompiled with the *make* command following the execution of this command.



```
(base) 218:src mattfranchi$ make clean
rm -f  cpsc2150/connectX/BoardPosition.class  cpsc2150/connectX/GameBoard.class  cpsc2150/connectX/GameScreen.
class  cpsc2150/connectX/IGameBoard.class
```