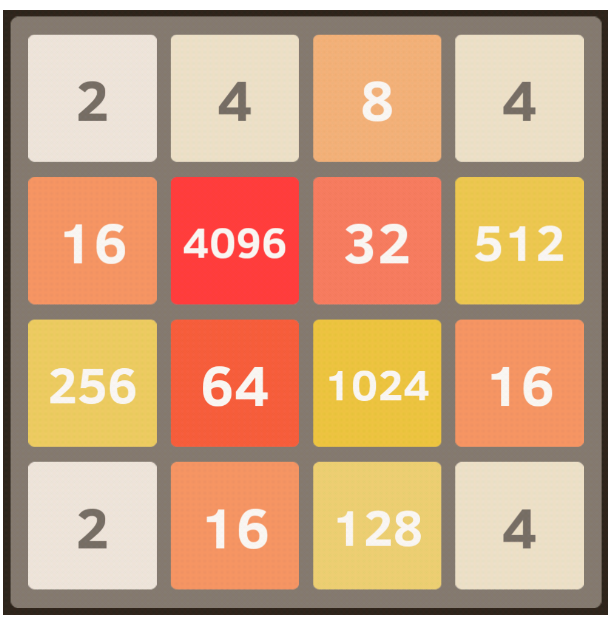
Assignment 4, Design Specification

Boming Jin jinb5

April 11, 2021

This Assignment4 specification contains modules, types, and methods for implementing the classic game *2048*. When starting the game. Users can simply press "w" "a" "s" "d" on the keyboard to slide all numbers on the board with one direction per sliding. Numbers can be slid vertically or horizontally. Boards will generate two numbers initially (which is 2 or 4) within random positions, also, after each sliding, the board will spawn a number 2 or 4 randomly. Users need to slide those numbers and add the same numbers together to get a final goal which is 2048. There is only one way about losing the game — when the board is full of numbers and there is no more possible moving, then users will fail this round. To simply run the game, just use "make demo" in the terminal to run the game.



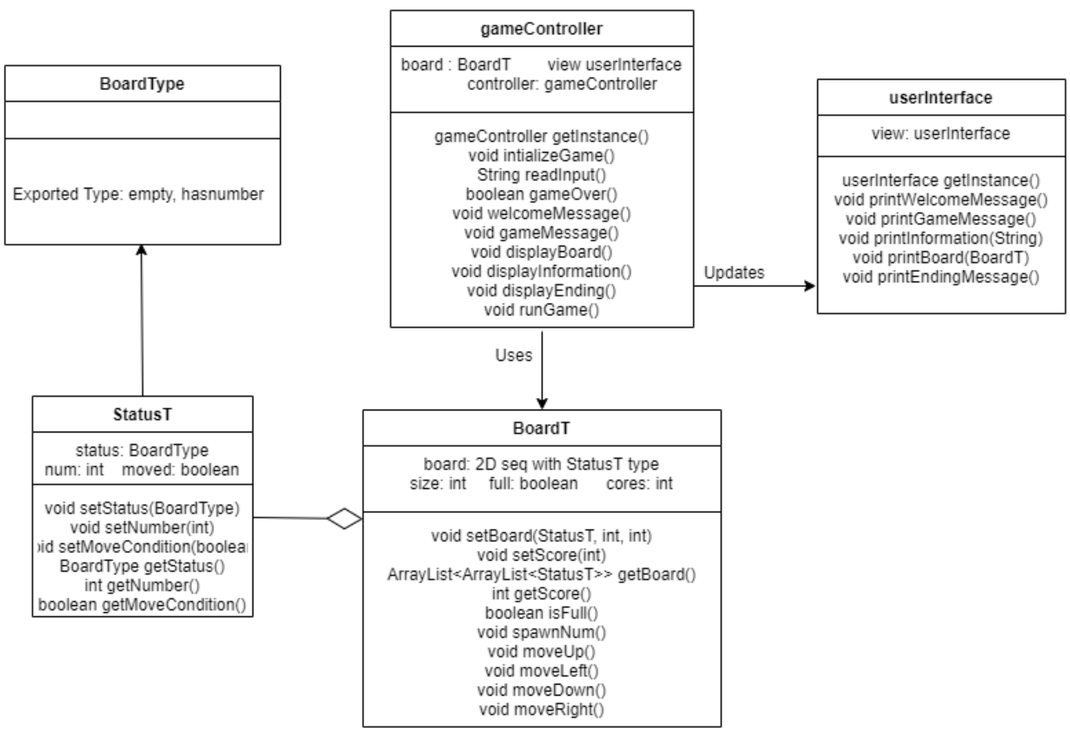
Picture found on Google

1 Overview of the design

My design applies Model View Controller(MVC) design pattern, Singleton Design pattern. The *gameController* (controller), *BoardT* (model), *userInterface* (view) composed MVC. The MVC design pattern was designed as the following path: my *StatusT* moudle stores states of cell in the board, and *BoardT* module uses *StatusT* as the state variables then manipulate the status of each cell of the board. My *userInterface* module built for display any information of the game in text version. The *gameController* module is for controlling the input and the output actions.

For getInstance() method in both *gameController* and *userInterface*, it is in order to obtain an abstract object during the running time.

A UML diagram for the architecture of my classes



Likely Changes my design considers:

- Grid size for harder level of the game *2048*
- Some good GUI for users to play game comfortably
- Multiple operations for operating the game (i.e. arrow keys to control the direction, mouse buttons to restart games, etc.)
- Store the best Score of the user
- Check how many moves that users did
- Rank scores for different users

BoardType Module

Module

BoardType

Uses

N/A

Syntax

Exported Constants

None

Exported Types

BoardType = {empty, hasnumber}

//empty represent that the cell is 0(empty), hasnumber says that the cell has some number in it(i.e. 2, 4, 8, 16, ...)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

StatusT Module

Module

StatusT

Uses

BoardType

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
StatusT	BoardType, $\mathbb{N}$	BoardT	
setStatus	BoardType		
setNumber	$\mathbb{N}$		
setMoveCondition	$\mathbb{B}$		
getStatus		BoardType	
getNumber		$\mathbb{N}$	
getMoveCondition		$\mathbb{B}$	

Semantics

State Variables

status: BoardType
num: $\mathbb{N}$
moved: $\mathbb{B}$
//status represent if the current cell is empty or not, num represents the number that this cell stored, moved represent if the current cell been moved in one user's action(without

this will add all numbers together in one action i.e. 0 2 2 4 turns to 0 0 0 8 when the user only slides the board to the right once)

State Invariant

None

Assumptions

When initializing a new StatusT as the cell of the board, I assume those cells not moved(moved by users, like slide to right, etc.)

Access Routine Semantics

new StatusT(t, i):

- transition: status, num := t, i   //BoardType t, int i
- output: out := self

setStatus(t):

- transition: status := t   //BoardType t
- exception: none

setNumber(i):

- transition: num := i
- exception: none

setMoveCondition(b):

- transition: moved := b
- exception: none

getStatus():

- output: out := status
- exception: none

getNumber():

- output: out := num
- exception: none

getMoveCondition():

- output: out := moved
- exception: none

BoardT Module

Module

BoardT

Uses

StatusT

Syntax

Exported Constants

size of the seq of (seq of StatusT) = 4 //Size of the board is 4 x 4

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT		BoardT	
BoardT	seq of (seq of StatusT)	BoardT	
setBoard	StatusT, $\mathbb{N}$, $\mathbb{N}$		
setScore	$\mathbb{N}$		
getBoard		seq of (seq of StatusT)	
getScore		$\mathbb{N}$	
isFull		$\mathbb{B}$	
moveUp			
moveLeft			
moveDown			
moveRight			

Semantics

State Variables

board: seq of (seq of StatusT)
full: $\mathbb{B}$

score : $\mathbb{N}$

State Invariant

None

Assumptions

Assume when running the game, construct BoardT at first before other operations.
When initializing will spawn 2 numbers(2 or 4) in random positions of the board(by using a local function) and the number of others is 0(empty). This means the status of StatusT is empty, and the number of StatusT is 0.
Assume there is a *Random()* function get the random numbers that I want.

Access Routine Semantics

new BoardT():

- transition: board, score, full := seq[4] of (seq[4] of StatusT), 0, false
- output: out := self
- exception: none

new BoardT(b):

- transition: board, score, full := b, 0, false   // b is seq[4] of (seq[4] of StatusT)
- output: self
- exception: none

setBoard(s, r, c):

- transition: board[r][c] := s
  //s is StatusT. r, c are the target position of the board that we want to set(i.e. board[0][0] is the left top corner of the board)
- exception: none

setScore(s):

- transition: score := s   //s is the integer number

- exception: none

getBoard():

- output: out := seq[4] of (seq[4] of StatusT)
- exception: none

getScore():

- output: out := score
- exception: none

isFull():

- output: out := $(\forall x, y : \mathbb{N} \mid x \in [0...size-1] \wedge y \in [0...size-1]$: board[x][y] ! = 0)
  //if all cells of the board have number then the board is full
- exception: none

moveUp():

- transition: board := $(\forall x, y : \mathbb{N} \mid x \in [1...size-1] \wedge y \in [0...size-1]$: move("w", x, y, {true, false}))
  //when moving(moveUp(), moveRight(), etc.), will use this local function move() twice for reset the moved status for StatusT. if state x before y then that means using for loop for x first then using for loop for y, vice versa
- exception: none

moveLeft():

- transition: board := $(\forall y, x : \mathbb{N} \mid y \in [1...size-1] \wedge x \in [0...size-1]$: move("w", x, y, {true, false}))
- exception: none

moveDown():

- transition: board := $(\forall x, y : \mathbb{N} \mid x \in [size-2...0] \wedge y \in [0...size-1]$: move("w", x, y, {true, false}))
- exception: none

moveDown():

- transition: board := $(\forall y, x : \mathbb{N} \mid y \in [size-2...0] \wedge x \in [0...size-1]$: move("w", x, y, {true, false}))
- exception: none

Local Functions

- spawnNum() *equiv* $(x, y : \mathbb{N} \mid x = Random(0-3) \wedge y = Random(0-3)$: board.get(x).set(y, StatusT))
  //Generate number in random cells of the board by using Random() function, also if board.isFull() == true, then don't spawn new numbers

- move(String s, int x, int y, boolean moved)
  //when using this local function we are moving cells, for example, if we are moving up, then cells will move from up to down and move in the up direction one by one, other directions are similar to moving up with different directions

- swap: $StatusT \times StatusT \rightarrow StatusT \times StatusT$
  swap(current, next) $\equiv$
  $(\neg(current.number = 0) \wedge (next.number = 0) \rightarrow (next.number = current.number) \wedge (current.number = 0)$
  $\mid \neg(current.number = 0) \wedge \neg(next.number = 0) \wedge (current.moved = false) \wedge (next.moved = false) \rightarrow$
  $((current.number = next.number) \rightarrow (next.number = 2 * current.number) \wedge (current.number = 0)))$
  //if the number of the current position is not 0 and the number of the next position is 0 then we swap the position of these two cells
  //if the number of the current position is not 0 and the number of the next position is not 0 and they are equal then double the number of the next position, the number of the current position will be 0

userInterface Module

Module

userInterface

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		userInterface	
printWelcomeMessage			
printGameMessage			
printInformation			
printBoard	BoardT		
printEndingMessage			

Semantics

Environment Variables

window: part of screen to display everything related to the game

State Variables

view: userInterface

State Invariant

None

Assumptions

Construct this object before running the game(before other classes use the method)

Access Routine Semantics

getInstance():

- transition: view := (view = null $\rightarrow$ new userInterface())
- output: self

printWelcomeMessage():

- transition: window := when user first time start the game or restart the game display this message

printGameMessage():

- transition: window := tell the instruction of the game to users

printInformation():

- transition: window := displays the information of the game to users(i.e. Score of the current round)

printBoard():

- transition: window := displays the board which is consists of numbers to users, each cell is *StatusT*, which stores the number of the cell. Numbers of cells could gotten by *getNumber()* method in *StatusT*. Note: board[x][y] is counting from left top corner.

printEndingMessage():

- transition: window := displays the ending message of the game when users choose to exit the game

gameController Module

gameController Module

Uses

BoardT, userInterface

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance	BoardT, userInterface	gameController	
initializeGame			
readInput		String	IllegalArgumentException
gameOver		$\mathbb{B}$	
welcomeMessage			
gameMessage			
displayBoard			
displayInformation			
displayEnding			
runGame			

Semantics

Environment Variables

keypressed: Scanner // read the input of the user

State Variables

board: BoardT view: userInterface controller: gameController

State Invariant

None

Assumptions

board and *view* already constructed before construct *gameController*

Access Routine Semantics

getInstance():

- transition: controller := (controller = null $\Rightarrow$ new userInterface(board, view))
- output: self

initializeGame():

- transition: board := new BoardT()
- output: none

readInput():

- output: keyboard : String, scanned from terminal which is entered by users
- exception: exc := (keyboard $\neq$ "w" $\wedge$ keyboard $\neq$ "a" $\wedge$ keyboard $\neq$ "s" $\wedge$ keyboard $\neq$ "d" $\wedge$ keyboard $\neq$ "r" $\wedge$ keyboard $\neq$ "e" $\rightarrow$ IllegalArgumentException)
  // "w", "a", "s", "d" represent for moving directions, "r" for restart the game, "e" for exit the game

gameOver():

- output: ($\forall$ row of seq of number $\in$ board : nextrow of seq of number $\neq$ row of seq of number) $\wedge$ ($\forall$ column of seq of number $\in$ board : nextcolumn of seq of number $\neq$ column of seq of number) $\rightarrow$ true

welcomeMessage():

- transition: view := view.printWelcomeMessage()

gameMessage():

- transition: view := view.printGameMessage()

displayBoard():

- transition: view := view.printBoard(board)

displayInformation():

- transition: view := view.printInformation()

displayEnding():

- transition: view := view.printEndingMessage()

runGame():

- transition: running the game, initialize board(game) first with welcome messages, then display the instruction, next let the user play the game, at the end let the user have the power to decide to restart or end the game

Critique of Design

- I choose to use StatusT to store the status of cells(i.e. if already moved in one action, numbers, if empty), which is convenient for me to combine multiple statuses, and easy to get or set for actually implementing. Also, the real condition of StatusT is hiding from other classes.

- Since I am only using controller and view modules once during the running time of the game, this offering fewer conflicts or unexpected state changes, hence it was less hard to test and easier to find some logical bugs during implementing time.

- For StatusT I think it is not essential because I chose to offer an enum class to represent the empty condition of the cell, but it is more easier to use 0 as empty cells, so enum class BoardType might not necessary for this design. I could establish another enum class that is used to combine with users' inputs, for example, if the user press "w", then the enum value could be "UP", etc.

- My StatusT might not provide generality to this design, since it worked for BoardT only, in other words, if I am going to add some new features, for instance, design the level of the game, design GUI for this game, then StatusT might not helpful for new classes adding.

- My modules are minimal since I think this is more helpful for me to check states of every objects, and easy to implement. Reduces the chance of meeting conflicts.

- When testing BoardT, since numbers spawning randomly around the board, so it is not really convenient to teat moveUP() and other similar functions. So I choose to reset every number in BoardT to be 0. But this might not enough for testing. Since we might meet unexpected bugs with more numbers.

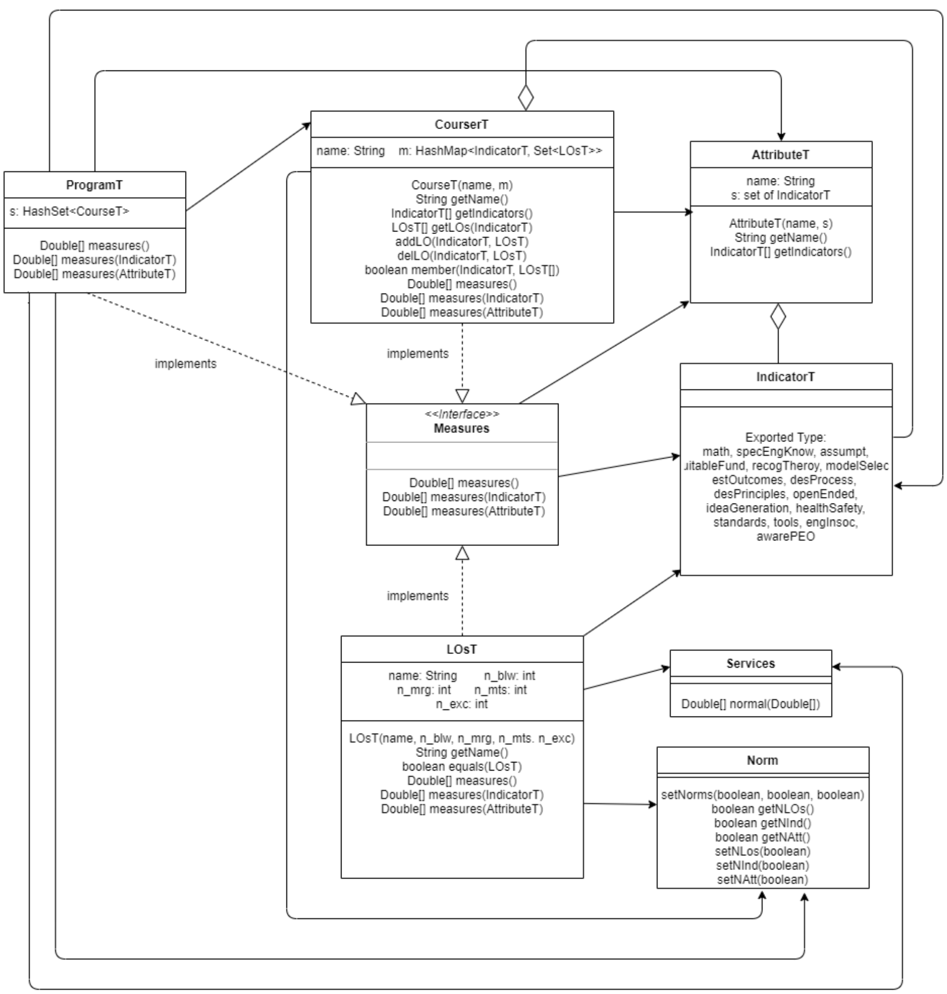
- I did not test gameController by using JUnit, since methods inside gameController are most from BoardT and userInterface, so I think gameController is not needed to be tested.

- I designed this game based on the idea of MVC. I found that MVC was really helpful for me to maintain the program and reduces the risk when changing some parts of the implementation. MVC basically could be separated into three parts: The model which handling the inside of the data, status, logic, etc. Also model encapsulates them. The Controller handles the interaction between users' inputs and reactions of the model(how the model behave with specific actions).

- By applying MVC to design the game, I think my modules could be said high cohesion and low coupling, since each module has related functionalities inside, and, each module looks mostly independent of others. This property guarantees to reduce the risk of changing little will impact many others.

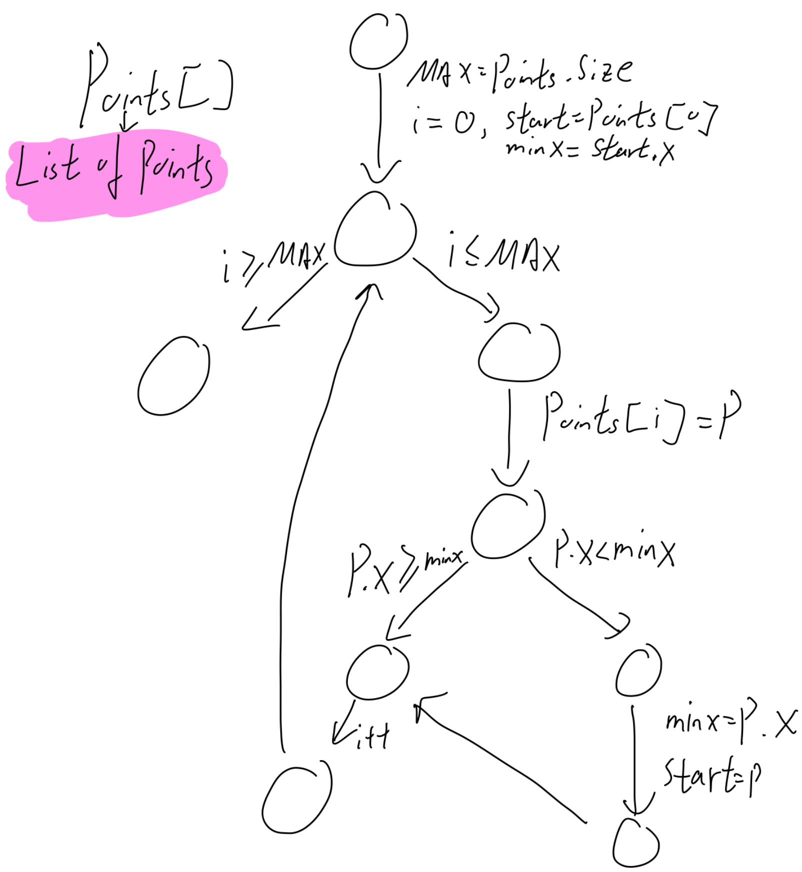
Answers to Questions:

Question1:



Question2:

First loop for finding leftmost point.



Second loop for basic algorithm