Assignment 4, Design Specification

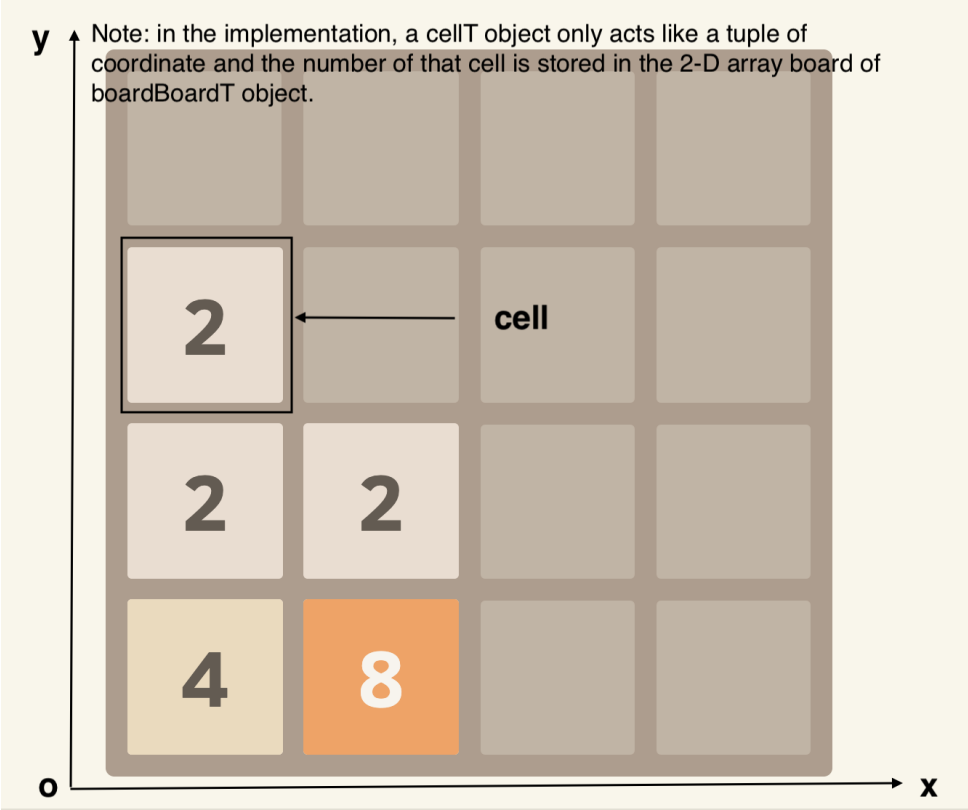
SFWRENG 2AA4

April 14, 2021

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game *2048*. At the start of each game, two random cells will be given number 2 or 4. Then the user can choose a move direction to move all the cells to that direction, if two cells have the same number, then they can be merged into a new cell with the number doubled. After that, the system will choose an empty cell (a cell with number 0 in implementation) to have a new number 2 or 4, with the probability of 90% and 10%, respectively. Then the new round starts, the user chooses a move direction... The user will win when a cell with number 2048 appears in the board; otherwise, the game ends when the board is fulfilled with cells with number other than 0 and moving to any direction will not update the board. During the game, a score is updated by adding the number doubled to the current score when a merge happens.

Some special behaviors of these game are listed below:

- Merge can only happen between two old cells (the cells that are already on the board) during one movement. For example, if a row has the value $[2, 2, 2, 2]$ and the user chooses to move left, then the result is $[4, 4, 0, 0]$, rather than $[8, 0, 0, 0]$; if a row has the value $[2, 2, 4, 4]$ and the user chooses to move left, then the result is $[4, 8, 0, 0]$, rather than $[8, 4, 0, 0]$.
- If the user choose to move to the direction that will not update the board (not change the numbers on the board), then no random 2 or 4 will be generated. This behavior is the same with the current online 2048 version, which in some way can avoid the unsatisfying cases caused by user's miss hitting.
- Currently, the game can be played in the terminal. Enter one of "w", "s", "a", "d" to indicate move up, down, left, right, respectively. When you want to quit, just enter "exit", then you will quit the game and go back to terminal.



The above board visualization is from https://play2048.co/.

How to Play :)

In your terminal, cd to A4, enter `make demo`. Then a prompt will be displayed. Enter any number for the game size (enter 4 or any invalid character for a 'normal' game). To move the cells, enter one of "w"(up), "a"(left), "s"(down), "d"(right) and hit enter/return. During the game, if you want to quit, just enter "exit" and hit enter/return.

Note: the main reason I choose to use "wasd" as the game control key is that the key mapping for arrows could raise some issues on different OSs.

On the next page is a sample play trace for illustrating this process. Have fun : -)

```
(base) Mingzhes-MacBook-Pro:A4 kidsama$ make demo
javac -cp "src:.:../junit-4.5.jar" -g src/Demo.java
java -cp "src:.:../junit-4.5.jar" src/Demo
------------------------------------------------
                Welcome to 2048
------------------------------------------------
                    New game
                ----------------

               Choose a game size:
           Please enter a small number
         Note: if you input is NOT a number,
      a default game of size 4 will be started.
4
------------------------------------------------
Current Score: 0
Current Board:
[[0, 0, 2, 0]
[0, 0, 0, 0]
[0, 0, 0, 2]
[0, 0, 0, 0]]
Please enter next move direction: w, a, s, d
------------------------------------------------
d
move right
------------------------------------------------
Current Score: 0
Current Board:
[[0, 0, 0, 2]
[0, 0, 2, 0]
[0, 0, 0, 2]
[0, 0, 0, 0]]
Please enter next move direction: w, a, s, d
------------------------------------------------
w
move up
------------------------------------------------
Current Score: 4
Current Board:
[[0, 0, 2, 4]
[0, 0, 0, 0]
[2, 0, 0, 0]
[0, 0, 0, 0]]
Please enter next move direction: w, a, s, d
------------------------------------------------
```

Overview of the design

This design applies Module View Specification (MVC) design pattern and Singleton design pattern. The MVC components are *GameController* (controller module), *BoardT* (model module), and *UserInterface* (view module). Singleton pattern is specified and implemented for *GameController* and *UserInterface*.

The MVC design pattern is specified and implemented in the following way: the module *BoardT* stores the state of the game board and the status of the game. A view module *UserInterface* can display the state of the game board and game using a text-based graphics. The controller *GameController* is responsible for handling input actions.

For *GameController* and *UserInterface*, use the getInstance() method to obtain the abstract object.

Likely Changes my design considers:

- Data structure used for storing the game board
- The visual representation of the game such as UI layout.
- Change in peripheral devices for taking user input.
- Change of the board size to adjust some variant versions of 2048.

Cell ADT Module

Template Module

CellT

Uses

None

Syntax

Exported Constants

None

Exported Types

CellT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
CellT	$\mathbb{N}$, $\mathbb{N}$	CellT	
getX		$\mathbb{N}$	
getY		$\mathbb{N}$	

Semantics

State Variables

x : $\mathbb{N}$
y : $\mathbb{N}$

State Invariant

None

Access Routine Semantics

new CellT(newX, newY):

- output: out := self
- transition: x, y := newX, newY

getX():

- output: out := x

getY():

- output: out := y

LinkedList ADT Module

Template Module

LinkedList

Uses

None

Syntax

Exported Constants

None

Exported Types

LinkedList = ?

Exported Access Programs

Routine name	In	Out	Exceptions
LinkedList		LinkedList	LinkedListERROR
add	$\mathbb{N}$		addERROR
pop		$\mathbb{N}$	popERROR
peek		$\mathbb{N}$	peekERROR
addAll	LinkedList		addAllERROR
isEmpty		$\mathbb{B}$	isEmptyERROR
size		$\mathbb{N}$	sizeERROR

Consideration

The `LinkedList` class is already in the java library. The above is only an list of the methods used in this design.

Board ADT Module

Template Module

BoardT

Uses

CellT, LinkedList, MoveDirection

Syntax

Exported Types

BoardT = ?

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT	$\mathbb{N}$	BoardT	
setBoard	seq of (seq of $\mathbb{N}$)		
setScore	$\mathbb{N}$		
getBoard		seq of (seq of $\mathbb{N}$)	
getScore		$\mathbb{N}$	
getSize		$\mathbb{N}$	
getEmptyCells		seq of CellT	
isGen24Flag		$\mathbb{B}$	
moveDown		seq of (seq of $\mathbb{N}$)	
down			
up			
left			
right			
move	MoveDirection		
genRand2Or4			
isWin		$\mathbb{B}$	
isLoose		$\mathbb{B}$	

Note: `getEmptyCells` and `getEmptyCells` both return an arraylist.

Semantics

State Variables

board: seq of (seq of $\mathbb{N}$) # the length of this 2d array is specified by the constructor.
score: $\mathbb{N}$
gen24Flag: $\mathbb{B}$

State Invariant

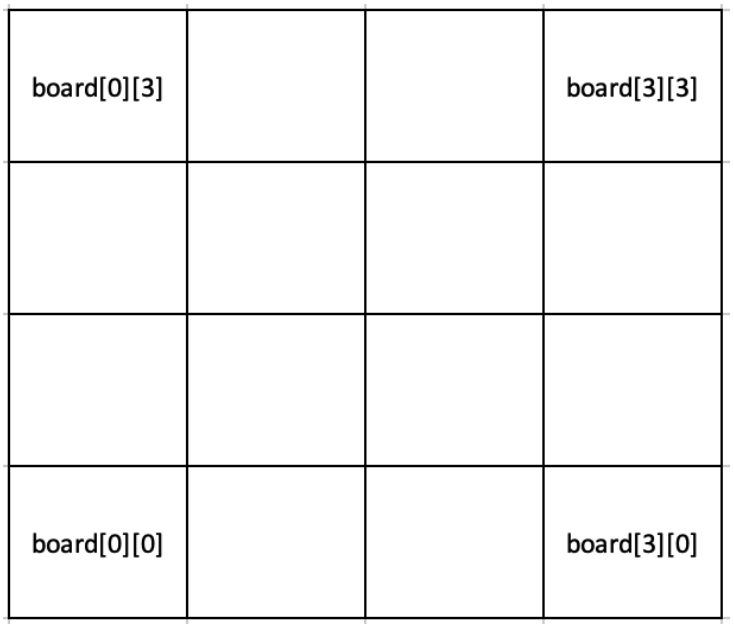
None

Assumptions

- The constructor BoardT is called for each object instance before any other access routine is called for that object.
- Assume there is a random function that generates a random value beteern 0 and 1.

Design decision

The coordinates of the board is stored in a 2D sequence. The coordinates used for retriving the cell is different from the coordinate entered by the user. When the user refers to row 0 and column 0 cell on their screen, that cell is actually stored in board[0][3]. Thus, board[0][0] is located at bottom left corner of the UI and board[3][3] is located at top right corner of the UI.



In the above illustration, the number in (0,0) of the UI is stored in board[0][3].

- The main reason I store the board in this way is for the ease of cell merge process. It is easier to perform move down operation if those cells are in the same row. Since each row of the board is represented as a column on the UI. The linkedlist operations like pop, peak and add can be naturally used in a row to simulate a drop down on the UI.

Access Routine Semantics

BoardT():

- transition:

  $$\text{board} := \langle \begin{matrix} \langle 0, ..., 0 \rangle \\ \langle 0, ..., 0 \rangle \\ ..... \\ \langle 0, ..., 0 \rangle \end{matrix} \rangle$$

  score, gen24Flag = 0, true
- output: *out* := *self*
- exception: None

setBoard(b):

- transition: board := b
- output: None
- exception: None

setScore(s):

- transition: score := s
- output: None
- exception: None

getBoard():

- transition: none
- output: $out := \langle \forall x, y : \mathbb{N} \mid 0 \leq x, y \leq |board| - 1 : board[x][y] \rangle$
- exception: None

getScore():

- transition: none
- output: $out :=$ score
- exception: None

getSize():

- transition: none
- output: $out :=$ board.length
- exception: None

getEmptyCells():

- transition: none
- output: $out := \langle \forall i, j : \mathbb{N} \mid board[i][j] = 0 : CellT(i, j) \rangle$
- exception: None

isGen24Flag():

- transition: none
- output: *out* := gen24Flag
- exception: None

moveDown():

- transition: # procedural specification

```
newBoard = new Seq[|board|] of (Seq[|board|] of ℕ)
for x in ⟨0, 1, ..., |board| − 1⟩
    curColum = new LinkedList()
    for y in ⟨0, 1, ..., |board| − 1⟩
        if board[x][y] > 0
            curColum.add(board[x][y])
    newColum = new LinkedList()
    while curColum.size() ≥ 2
        firstNum = curColum.pop()
        secondNum = curColum.peek()
        if firstNum = secondNum
            newColum.add(firstNum ∗ 2)
            score = score + firstNum ∗ 2
            curColum.pop()
        else
            newColum.add(firstNum)
    newColum.addAll(curColum)
    newBoard[x] = new Seq[|board|] of ℕ
    for y in ⟨0, 1, ..., |board| − 1⟩
        if newColum.isEmpty()
            newBoard[x][y] = 0
        else
            newBoard[x][y] = newColum.pop()
```

- output: *out* := newBoard
- exception: None

down():

- transition: # procedural specification

```
board = moveDown()
```

- output: None
- exception: None

moveDown will implicitly update the gen24Flag and score field as listed above.

up():

- transition: # procedural specification

  board = mirrorLeftRight(board)
  board = mirrorLeftRight(moveDown())

- output: None
- exception: None

moveDown will implicitly update the gen24Flag and score field as listed above.

left():

- transition: # procedural specification

  board = transpose(board)
  board = transpose(moveDown())

- output: None
- exception: None

moveDown will implicitly update the gen24Flag and score field as listed above.

right():

- transition: # procedural specification

  board = transpose(board)
  board = mirrorLeftRight(board)
  board = transpose(mirrorLeftRight(moveDown()))

- output: None
- exception: None

moveDown will implicitly update the gen24Flag and score field as listed above.

move(m):

- transition: # procedural specification

  (m = moveDirection.UP ⇒ up() | m = moveDirection.DOWN ⇒ down() | m = moveDirection.LEFT ⇒ left() | m = moveDirection.RIGHT ⇒ right())

- output: None
- exception: None

moveDown will implicitly update the gen24Flag and score field as listed above.

genRand2Or4():

- transition: # procedural specification

  Randomly select a cell from all cells in getEmptyCells(), then update the board[cell.getX()][cell.getY()] to 2 or 4, whose occurrence probability is 90% and 20%, respectively.

- output: None
- exception: None

isWin():

- transition: None
- output: out := $\exists(x, y : \mathbb{N} \mid 0 \leq x, y \leq |board| - 1 : board[x][y] = 2048)$
- exception: None

isLoose():

- transition: None
- output: *out* := compareTwoInt2DArr(oldBoard, newBoard1) ∧ compareTwoInt2DArr(oldBoard, newBoard2) ∧ compareTwoInt2DArr(oldBoard, newBoard3) ∧ compareTwoInt2DArr(oldBoard, newBoard4),
  where oldBoard is the current board state, newBoard1, newBoard2, newBoard3 and newBoard4 are the new board states after calling up(), down(), left(), and right(), respectively.
- exception: None

Local Functions

transpose: Seq of (Seq of $\mathbb{N}$) $\rightarrow$ Seq of (Seq of $\mathbb{N}$)
transpose(inputArr) $\equiv$

$$\left\langle \begin{array}{c} \langle a_{00}, a_{10}, ..., a_{n0} \rangle \\ \langle a_{01}, a_{11}, ..., a_{n1} \rangle \\ ..... \\ \langle a_{0n}, a_{1n}, ..., a_{nn} \rangle \end{array} \right\rangle,$$

when inputArr =

$$\left\langle \begin{array}{c} \langle a_{00}, a_{01}, ..., a_{0n} \rangle \\ \langle a_{10}, a_{11}, ..., a_{1n} \rangle \\ ..... \\ \langle a_{n0}, a_{n1}, ..., a_{nn} \rangle \end{array} \right\rangle,$$

where n = $|inputArr|$.

mirrorLeftRight: Seq of (Seq of $\mathbb{N}$) $\rightarrow$ Seq of (Seq of $\mathbb{N}$)
mirrorLeftRight(inputArr) $\equiv$

$$\left\langle \begin{array}{c} \langle a_{0n}, ..., a_{01}, a_{00} \rangle \\ \langle a_{1n}, ..., a_{11}, a_{10} \rangle \\ ..... \\ \langle a_{nn}, ..., a_{n1}, a_{n0} \rangle \end{array} \right\rangle,$$

when inputArr =

$$\left\langle \begin{array}{c} \langle a_{00}, a_{01}, ..., a_{0n} \rangle \\ \langle a_{10}, a_{11}, ..., a_{1n} \rangle \\ ..... \\ \langle a_{n0}, a_{n1}, ..., a_{nn} \rangle \end{array} \right\rangle,$$

where n = $|inputArr|$.

compareTwoInt2DArr: Seq of (Seq of $\mathbb{N}$) $\times$ Seq of (Seq of $\mathbb{N}$) $\rightarrow \mathbb{B}$
compareTwoInt2DArr(arr1, arr2) $\equiv$ ($|arr1| \neq |arr2| \Rightarrow False \mid (\forall x : \mathbb{N} \mid 0 \leq x \leq |arr1| - 1 : |arr1[x]| \neq |arr2[x]|) \Rightarrow False \mid (\forall x, y : \mathbb{N} \mid x, y \in [0, .., |arr1| - 1] : arr1[x][y] = arr2[x][y]) \Rightarrow True$).

UserInterface Module

UserInterface Module

Uses

None

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		UserInterface	
printBoard	Seq of (Seq of $\mathbb{N}$), $\mathbb{N}$		
printWelcomeMessage			
printGameSizePrompt			
printWinningMessage			
printLosingMessage	$\mathbb{N}$		
printEndingMessage			

Semantics

Environment Variables

window: A portion of computer screen to display the game and messages

State Variables

visual: UserInterface

State Invariant

None

Assumptions

- The UserInterface constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

getInstance():

- transition: visual := (visual = null ⇒ new UserInterface())
- output: *self*
- exception: None

printWelcomeMessage():

- transition: window := Displays a welcome message when user first enter the game.

printBoard(*board*, *score*):

- transition: window := Draws the game board onto the screen. Each cell of the board is accessed by traversing the 2D array board of *BoardT*. The board[x][y] is displayed in a way such that x is increasing from the left of the screen to the right, and y value is increasing from the bottom to the top of the screen. For example, board[0][0] is displayed at the bottom left corner and board[4][4] is displayed at top-right corner.

printGameSizePrompt():

- transition: window := Window appends a prompt message asking the user to enter a number as the game board size.

printWinningMessage():

- transition: window := Prints a congratulation message after the the user wins.

printLosingMessage(*score*):

- transition: window := Prints a message showing the final score after the the user looses.

printEndingMessage():

- transition: Prints a ending message after the the user either wins or looses.

Local Function:

UserInterface: void → UserInterface
UserInterface() ≡ new UserInterface()

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	$\mathbb{N}$		
readGameSizeInput		$\mathbb{N}$	
displayWelcomeMessage			
displayBoard			
displayWinningMessage			
displayLosingMessage			
displayEnding			
displayGameSizePrompt			
readUserMove			InputMismatchException
runGame			

Semantics

Environment Variables

keyboard: Scanner(System.in) *// reading inputs from keyboard*

State Variables

model: BoardT
view: UserInterface
controller: GameController

State Invariant

None

Assumptions

- The GameController constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Assume that model and view instances are already initialized before calling GameController constructor

Access Routine Semantics

getInstance(m, v):

- transition: controller := (controller = null $\Rightarrow$ new GameController (m, v))
- output: *self*
- exception: None

initializeGame($size$):

- transition: model := $newBoardT(size)$
- output: None
- exception: None

readGameSizeInput():

- output: *out* := (input is a integer, entered from the keyboard by the User $\Rightarrow$ input | 4)
- exception: None

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage()

displayBoard():

- transition: view := view.printBoard($model.getBoard()$, $model.getScore()$)

displayWinningMessage():

- transition: view := view.printWinningMessage()

displayLosingMessage():

- transition: view := view.printWinningMessage($model.getScore()$)

displayEndingMessage():

- transition: view := view.printEndingMessage()

displayGameSizePrompt():

- transition: view := view.printGameSizePrompt()

readUserMove():

- transition: None
- output: out := (($input$ =‘w’⇒ $MoveDirection.UP$) | ($input$ =‘a’⇒ $MoveDirection.LEFT$) | ($input$ =‘s’⇒ $MoveDirection.DOWN$) | ($input$ =‘d’⇒ $MoveDirection.RIGHT$)) | ($input$ =‘exit’⇒ $System.exit(0)$)), where input : $char$, entered from the keyboard by the User.
- exception: exc := ((input $\neq$ ‘w' $\wedge$ input $\neq$ ‘a' $\wedge$ input $\neq$ ‘s' $\wedge$ input $\neq$ ‘d') $\Rightarrow$ InputMismatchException).

runGame():

- transition: operational method for running the game. The game will start with a welcome message, next asking the user to enter a game board size (the default size is 4), then display the board and let the user to play the game. Eventually, when the game ends, prompt a winning or loosing message.
- output: None

Local Function:

GameController: BoardT $\times$ UserInterface $\rightarrow$ GameController
GameController($model, view$) $\equiv$ new GameController($model, view$)

Critique of Design

- For all of the modules, the `BoardT` is designed to be an ADT, because a board should be updated during the game; while `GameController` and `UserInterface` are designed to be an abstract object using an singleton design strategy, which I think can help avoid odd behaviors in the game running time and save necessary memory space.

- The `CellT` module is only like a typo class that stores the coordinate information, while the number of each cell is still stored in the board field of `BoardT`. This design may seem no **essential**, but it can be used to support other game variants or the later GUI work.

- By designing an enum class `MoveDirection`, the system can deal with all user's movement during the game, instead of deal with Strings that can easily raise exceptions.

- The 2D array state variable of BoardT is also named board, whose naming convention may seem not **consistent**. But as many online resources and previous assignment use this naming, I think follow this convention can make my modules more understandable.

- This MIS is **general** because instead of only supporting the classical 2048, user can also specify a new board size, which can make the game more fun. In addition, the `CellT` is here to deal with future changes like a GUI requirement.

- The `moveDown` method in `BoardT` violates the **minimality** principle, because it uses an algorithm to compute the entire board after move down and in the meantime updates the score. I think its unminimal cannot be avoided in the case that we want the game running more efficiently.

- This design is **high cohesion and low coupling** mostly because the using of MVC design strategy. The `GameController` updates the `BoardT` and uses `UserInterface` to display current state.

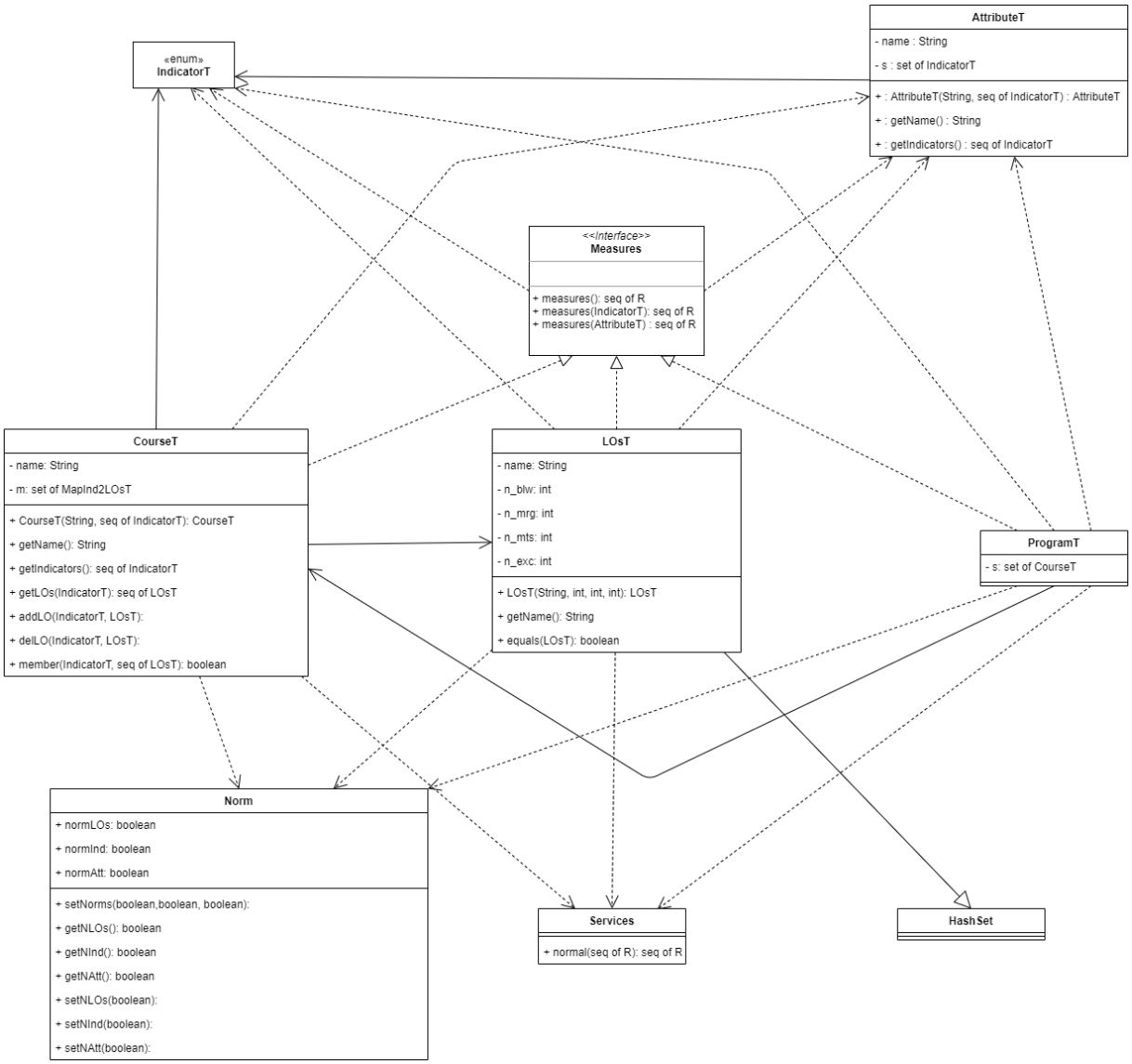
- The **information hiding** is good. By using `GameController` in the Demo, we can ignore all other unnecessary implementation details in other methods, which make the software easy to use.

- Did not build any test cases for testing the controller module since the implementation of the controller's access methods uses methods from the model and view. The test cases for the model are in `TestBoardT.java`.

- In this design, I also use a singleton design pattern in both `GameController` and `UserInterface`, for which we can only generate one instance. During the developing process, using static methods and variables usually cause some warnings about the method or variables to need to be accessed statically. Using singleton pattern eliminates all these problems, making a smoother development.

Answers to Questions:

Q1



Q2