Assignment 4, Design Specification

SFWRENG 2AA4

April 16, 2021

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game *2048*. At the start of each game, the board is is randomly filled with two numbers either 2 or 4. The user has four moves in total to play the game. The keyboard keys w, a, s, d are used to move the numbers in the board along the top, left, down and right respectively. The game continues until the entire board is filled up with numbers and cannot merge anymore or if the user achieves the number 2048. The only way to win this game is to achieve 2048 in the board.

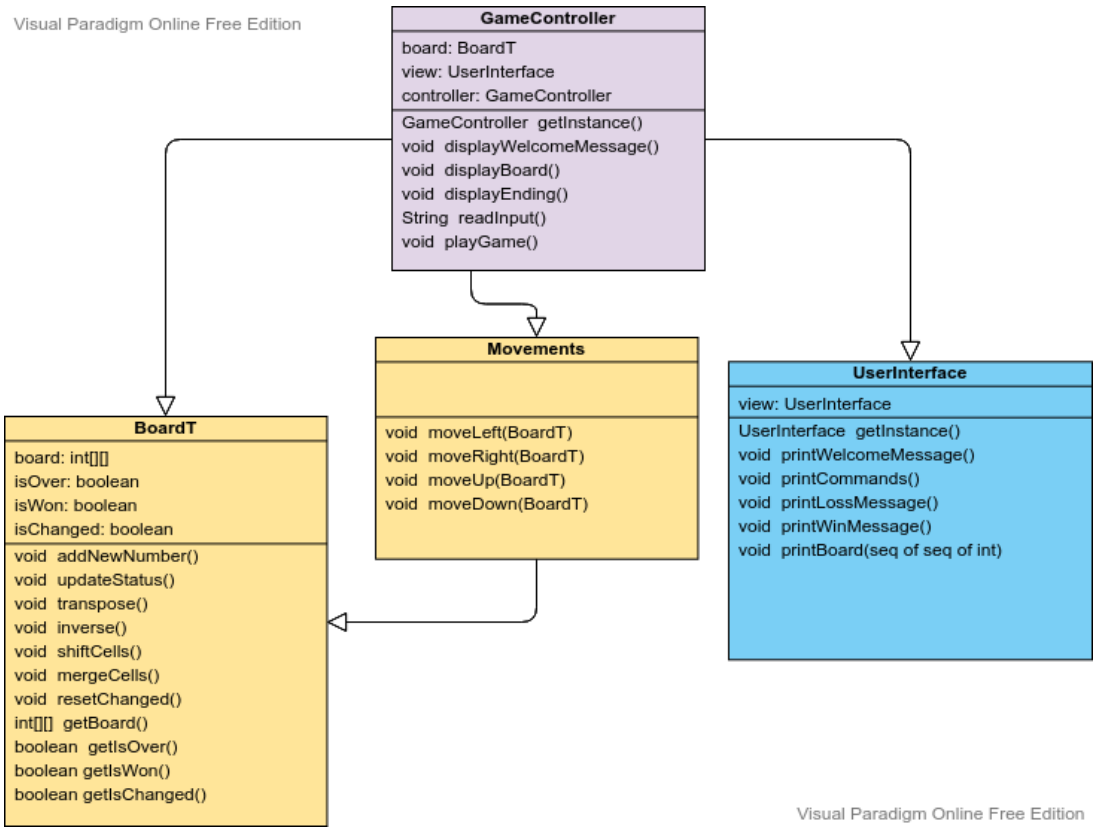
1 Overview of the design

This design applies Model-View-Controller (MVC) design pattern and Singleton design pattern. The MVC components are $BoardT$, $Movements$ (model module), $UserInterface$ (view module) and *GameController* (controller module). Singleton design pattern is specified and implemented for $UserInterface$ and *GameController*.

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. The module $Movements$ handles all the user input. A view module $UserInterface$ displays the state of the game board and game using a test-based graphics. The controller module *GameController* is responsible for handling input actions. The design allows the board size to be anything greater than 3.

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

A UML diagram is provided below for visualizing the structure of this software architecture.



Likely Changes my design considers:

- Data structure used for storing the game board.
- Change in the size of the game board.
- Change in peripheral devices for taking user input.
- The way the game is visually represented.

Board ADT Module

Template Module

BoardT

Uses

None

Syntax

Exported Types

BoardT = ?

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT	$\mathbb{N}$	BoardT	IllegalArgumentException
addNewNumber			
updateStatus			
transpose			
inverse			
shiftCells			
resetChanged			
getBoard		seq of (seq of $\mathbb{N}$)	
getIsOver		$\mathbb{B}$	
getIsWon		$\mathbb{B}$	
getIsChanged		$\mathbb{B}$	

State Variables

board: seq of (seq of $\mathbb{N}$)
isOver: $\mathbb{B}$
isWon: $\mathbb{B}$
changed: $\mathbb{B}$

State Invariant

None

Assumptions

- Assume there is a random function that generates a random value between 0 and 1.
- Access routine addNewNumber is called two times after constructor is called.

Access Routine Semantics

new BoardT(size):

- transition: board, isOver, changed := sequence [size, size] of $\mathbb{N}$, false, false
- output: *out* := *self*
- exception: exc := ($size < 4 \Rightarrow IllegalArgumentException$)

addNewNumber():

- transition: board := Adds a number (either 2 or 4) on a random position in the board. The number 2 has a 90% probability of being picked whereas 4 has 10% probability.
- output: None
- exception: None

updateStatus():

- transition:
  isWon := $\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = 2048) \Rightarrow true)$

  isOver := $(\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = 2048) \Rightarrow true)|$
  $\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = board[i + 1][j]) \Rightarrow false)|$
  $\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = board[i][j + 1]) \Rightarrow false)|$
  $\exists(i : \mathbb{N} | i \in [0..|board| - 1] : board[i][|board| - 1] = board[i+1][|board| - 1]) \Rightarrow false)|$
  $\exists(i : \mathbb{N} | i \in [0..|board| - 1] : board[|board| - 1][j] = board[|board| - 1][j+1]) \Rightarrow false)|$
  $true \Rightarrow true)$
- output: None
- exception: None

transpose():

- transition: board := $\forall(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = board[j][i])$
- output: None
- exception: None

inverse():

- transition: board := $\forall(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : board[i][j] = board[i][|board| - 1 - j])$
- output: None
- exception: None

shiftCells():

- transition: Moves all the numbers present in the board towards their respective left most index that is not filled up with any other number.
- output: None
- exception: None

mergeCells():

- transition:
  board := $\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : (board[i][j] = board[i][j+1]) \wedge board[i][j] \neq 0) \Rightarrow (board[i][j] = board[i][j] * 2) \wedge (board[i][j + 1] = 0))$

  changed := $\exists(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : (board[i][j] = board[i][j + 1]) \wedge board[i][j] \neq 0) \Rightarrow true$
- output: None
- exception: None

resetChanged():

- transition: changed := false
- output: None
- exception: None

getBoard():

- output: board
- exception: None

getIsOver():

- output: isOver
- exception: None

getIsWon():

- output: isWon
- exception: None

getIsChanged():

- output: changed
- exception: None

Movements Module

Module

Movements

Uses

BoardT

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
moveLeft	BoardT		
moveRight	BoardT		
moveUp	BoardT		
moveDown	BoardT		

State Variables

None

State Invariant

None

Assumptions

- Assume there is already an instance of BoardT called before using any of the access routines.

Access Routine Semantics

moveLeft(board):

- transition: Performs actions on the game board when the left button "a" is pressed. This method calls shiftCells() to shift all the numbers in the board to the left most index and calls mergeCells() to see if any of the number merges or not. It again calls shiftCells() in case there are any empty cells in between.
- output: None
- exception: None

moveRight(board):

- transition: Performs actions on the game board when the right button "d" is pressed. This method calls inverse() to inverse the board and calls moveLeft() to perform a left moving action. It again calls inverse() to inverse the board to result the action of moving right.
- output: None
- exception: None

moveUp(board):

- transition: Performs actions on the game board when the up button "w" is pressed. This method calls the transpose() method to interchange the rows and columns of the board and performs a moveLeft() method. It again calls the transpose() to result in a moving up action.
- output: None
- exception: None

moveDown(board):

- transition: Performs actions on the game board when the down button "s" is pressed. This method calls the transpose() method to interchange the rows and columns of the board and performs a moveRight() method. It again calls the transpose() to result in a moving down action.
- output: None
- exception: None

UserInterface Module

Module

UserInterface

Uses

None

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		UserInterface	
printWelcomeMessage			
printCommands			
printLossMessage			
printWinMessage			
printBoard	seq of (seq of $\mathbb{N}$)		

State Variables

view: 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: view := (view = null ⇒ new UserInterface())
- *out* := *self*
- exception: None

printWelcomeMessage():

- transition: Displays a welcome message on the screen when user first enters the game.

printCommands():

- transition: Displays the input commands on the screen to play the game.

printLossMessage():

- transition: Displays the game lost message on the screen when the game ends.

printWinMessage():

- transition: Displays the game won message on the screen when the game ends.

printBoard(board):

- transition: Draws the game board on the screen. The board[x][y] is displayed in such a way that x is increasing from top of the screen to the bottom, and y value is increasing from left of the screen to the right. For example, board[0][0] is displayed at the top-left of the screen and board[4][4] is displayed at the bottom-right of the screen.

Local Function:

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

GameController Module

Module

GameController

Uses

BoardT, UserInterface

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance	BoardT, UserInterface	GameController	
displayWelcomeMessage			
displayBoard			
displayEnding			
readInput		String	
playGame			

Semantics

Environment Variables

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

State Variables

board: 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 board and view instances are already initialized before calling GameController constructor

Access Routine Semantics

getInstance(*board*, *view*):

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

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage() $\wedge$ view.printCommands()

displayBoard():

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

displayEnding():

- transition: view := ($board.getIsWon() \Rightarrow view.printWinMessage()|$ $true \Rightarrow view.printLossMessage()$)

readInput():

- output: *input* := String entered from the keyboard by the user

playGame():

- transition: Runs the game in an infinite loop until board.getIsOver() is true. Initially runs displayWelcomeMessage() and displayBoard() to the screen. Then takes input from the user (w, s, a, d) and moves the numbers on the board accordingly. After each user input, the board is displayed and the status of game is updated by running board.updateStatus().

	out :=
input = 'w'	Movements.moveUp(board)
input = 's'	Movements.moveDown(board)
input = 'a'	Movements.moveLeft(board)
input = 'd'	Movements.moveRight(board)

Local Function:

GameController: BoardT × UserInterface → GameController
GameController($board, view$) ≡ new GameController($board, view$)

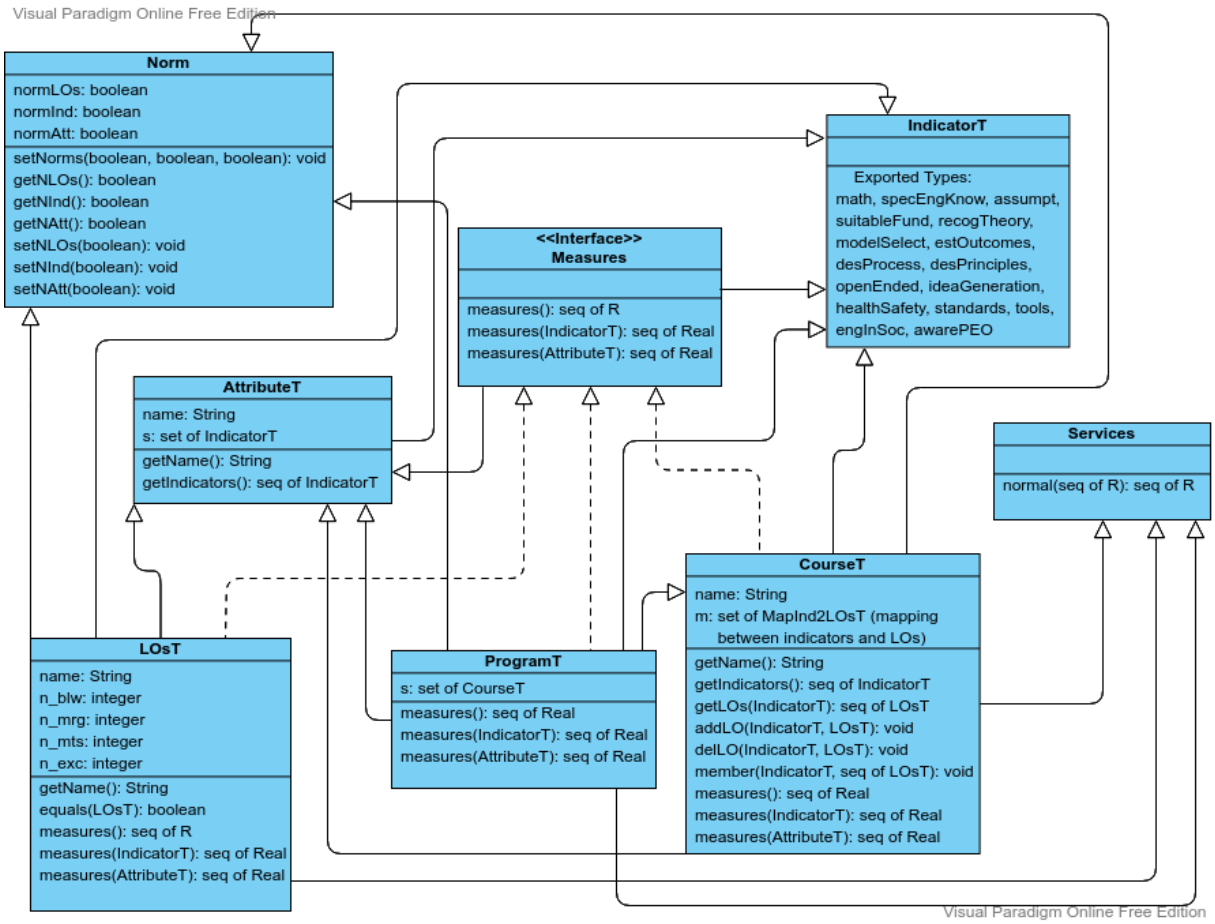
Critique of Design

- The controller and view modules are specified as a single abstract object because these modules are shared resources and only one instance is required during the runtime of the game. This allows any unexpected or conflicting errors to be avoided.
- The design is consistent for many factors. All the functions that return a boolean value is starts with a "is", this makes it easier to identify that the function returns a boolean value. Throughout the entirety of the design we follow the camelCase naming convention.
- The design is essential because we do not have any unnecessary features. There are key methods that makes the handling of user input easier - transpose(), inverse(), shiftCells() and mergeCells(). These methods are reused in all user input methods.
- The design implementation is as independent as possible. The modules $BoardT$, $Movements$ and $UserInterface$ do not interact with each other at all. Only the module $GameController$ interacts with the rest of the module and controls the game.
- The design is minimal because each method has their own independent service. All methods have their own specific role and they do that exact thing, nothing else.
- This design achieves high cohesion and low coupling due to the fact that it follows the MVC design pattern. This design achieves low coupling because it has 3 separate components - model ($BoardT$, $Movements$), view ($UserInterface$) and controller ($GameController$) that are independent of each other. It also achieves high cohesion because all the related components are separated into their own respective modules.
- Our design is opaque because a lot of information hiding is going on. During the runtime, the board is being manipulated in various ways to achieve the functionality changes caused to the board due to user inputs.
- A lot of checks were given in the design so that the programmer can avoid exceptions. In the playGame() method, while taking user input, if anything other than w, s, a or d is pressed the game still continues to ask the input again rather than breaking down.
- The test cases are designed to validate the correctness of the program based on the requirements and reveal errors or unusual behaviour during the execution of the program. Every access routine has at least one test case.

- The test cases for model are in *TestBoardT.java* and *TestMovements.java*. The controller module doesn't have any test cases because the implementation of controller's access routines uses methods from model and view.

Answers to Questions:

Q1: Draw a UML diagram for the modules in A3.



Q2: Draw a control flow graph for the convex hull algorithm. The graph should follow the approach used by the Ghezzi et al. textbook.