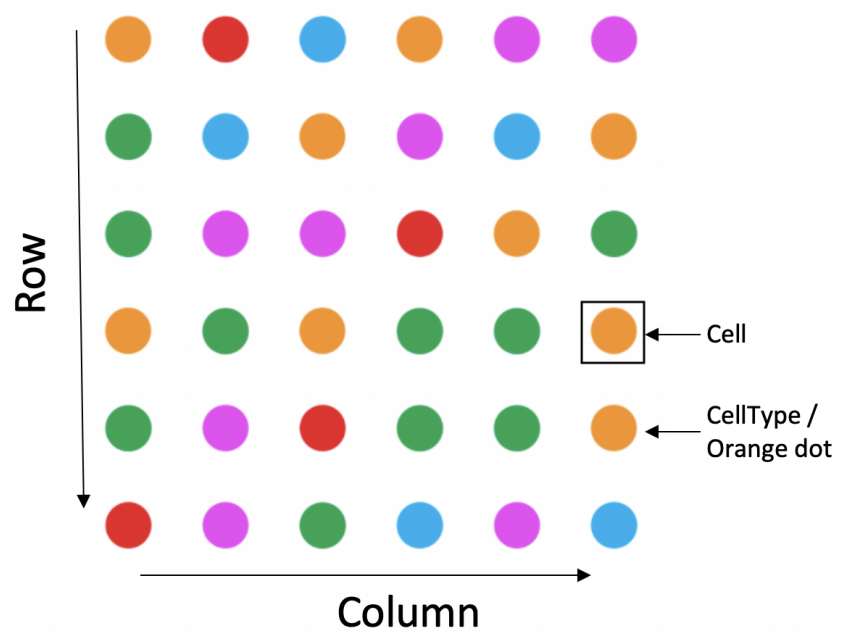
Assignment 4, Design Specification

SFWRENG 2AA4

April 14, 2021

This Module Interface Specification (MIS) document contains the modules necessary to implement the model and the view of the game *2048*. At the start of the game, the player will see a 4x4 board containing 16 cells. Two random cells will contain either two `2` tiles, or one `2` tile and one `4` tile. The player can then move the tiles either up, down, left, or right, and the tiles slide over in that direction as far as they can go, unless they are blocked by the edge of the board, or another tile. If the tiles that slide together are the same, then they combine to create a single tile whose value is two times its original value. Therefore, all cells are either empty, or contain a tile whose value is 2^n, where n is a natural number not including 0. For this implementation, the empty cells will be represented by a `0` tile. A visualization of the game board and an example of a move `right` is shown below:



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

1 Overview of the design

This design applies Model View Controller (MVC) design pattern, where *BoardT* is the model module and *View* is the view module. The module *BoardT* stores the state of the game board and the status of the game, while the view module *View* displays the current state of the game board and the overall game using text-based (ASCII) graphics. There is no controller module in this implementation.

Likely Changes my design considers:

- Data structure used for storing the game board
-
-
- Change in game ending conditions to adjust the difficulty of the game.

Board ADT Module

Template Module

BoardT

Uses

None

Syntax

Exported Types

None

Exported Constant

size = 4 // Size of the board 4 x 4

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT		BoardT	
getBoard		seq of (seq of $\mathbb{N}$)	
getScore		$\mathbb{N}$	
emptyCells		set of (seq of $\mathbb{N}$)	
isGameWon		$\mathbb{B}$	
isGameLost		$\mathbb{B}$	
setBoard	seq of (seq of $\mathbb{N}$)		IllegalArgumentException
resetBoard			
isValidMoveRight		$\mathbb{B}$	
isValidMoveLeft		$\mathbb{B}$	
isValidMoveUp		$\mathbb{B}$	
isValidMoveDown		$\mathbb{B}$	
moveRight			
moveLeft			
moveUp			
moveDown			

Semantics

State Variables

board: sequence [size, size] of $\mathbb{N}$
score: $\mathbb{N}$
empty: set of (sequence of $\mathbb{N}$)
win: $\mathbb{B}$
lose: $\mathbb{B}$

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.
- The seq of (seq of $\mathbb{N}$) provided as input for the *setBoard* method will consist of correct numbers. This means that the numbers will be either 0 to represent an empty cell, or 2^n, where $n \neq 0$.
- The set of (sequence of $\mathbb{N}$) used to represent the state variable *empty* will contain the coordinates for the empty cells of the game board. The sequence of $\mathbb{N}$ will only contain two numbers to represent the row and column of an empty cell. For example, *empty* will be a set of $[row, column]$.

Access Routine Semantics

BoardT():

- transition:

$$\text{board} := \left\langle \begin{array}{l} \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \\ \langle \text{randomCellType()}_0, ..., \text{randomCellType()}_7 \rangle \end{array} \right\rangle$$

status, moves = true, 0

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

BoardT(b):

- transition: board, status, moves = b, true, 0
- output: *out* := *self*
- exception: None

getStatus():

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

getMessage():

- transition: none
- output: *out* := (condtion.message(moves))
- exception: None

getCell(x, y):

- output: *out* := board[y][7 - x]
- exception: exc := ($\neg$ validateCell(x, y) $\Rightarrow$ IndexOutOfBoundsException)

setCondition(c):

- transition: condition := c
- output: None
- exception: None

remove(*input*):

- transition: board := markCell(*input*) $\Rightarrow$ ($\forall i : \mathbb{N} \mid i < size \wedge$ board[i].removeAll($CellType.E$))

- output: None
- exception: exc := (($\neg$ validateConnectivity($input$) $\Rightarrow$ IllegalArgumentExceptio) | ()$|input| = 1 \Rightarrow$ IllegalArgumentException))

replaceEliminated($input$):

- transition: board :=
  ($|board[i]| <$ size $\Rightarrow (\forall\, i : \mathbb{N} \mid |input| <= i < size$. board[i] || ¡randomCellType()¿))
  // Add randomized colored cell to a row if that row is missing some cells

Local Functions

validateCell: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

validateCell$(x, y) \equiv x <$ Size $\wedge\ y <$ Size $\wedge\ x >= 0 \wedge\ y >= 0$

validateConnectivity: sequence of String $\rightarrow \mathbb{B}$

validateConnectivity(input) $\equiv \forall i : \mathbb{N} \mid i > 0 \wedge i < 8$. $x = input[i][0], y = input[i][1]$ $x_1 = input[i][0], y_1 = input[i-1][1]$.	$getCell(x, y) == getCell(x_1, y_1)$ $(x + 1 == x1 \vee x - 1 == x1) \wedge (y == y1)$ $(y + 1 == y1 \vee y - 1 == y1) \wedge (x == x1)$

// Checks the current cell with the previous one. It is valid if they both have the same color and adjacent meaning either horizontally or vertically connected

markCell: sequence of String

markCell: $\forall\, i : \mathbb{N} \mid i < |input| \wedge$ board$[input[i].charAt[0]][input[i].charAt[1]] = CellType.E$
// mark each coordinate in the input as CellType.E (empty cell)

UserInterface Module

UserInterface Module

Uses

None

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		UserInterface	
printBoard	BoardT		
printWelcomeMessage			
printGameModePrompt			
printCoordPrompt			
printCondition	String		
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*):

- transition: window := Draws the game board onto the screen. Each cell of the board is accessed using the *getCell* method from *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[7][7] is displayed at top-right corner.

printGameModePrompt():

- transition: window := Window appends a prompt message asking the user to select a game mode he/she wants to play.

printCoordPrompt():

- transition: window := Appends a prompt message asking the user to enter a sequence of coordinates where the dots will be eliminated.

printCondition(*message*):

- transition: Appends the *message* onto the screen, the *message* shows the objective of current game. Also, the amount of moves of number of specific dots remanin to complete the game.

printEndingMessage():

- transition: Prints a ending message after the user exit the game (entered "e").

Local Function:

UserInterface: void $\rightarrow$ UserInterface
UserInterface() $\equiv$ new UserInterface()

Critique of Design

- I choose to specify BoardT module as ADT over abstract object, because It is more convenient to create a new instance of the board after the user choose to restart a game.
- The controller and view modules are specified as a single abstract object because these modules are shared resources and only one instance is required to control the action during runtime. Thus, any conflicting or unexpected state changes can be avoided.
- The *getCell*() method in *BoardT* module is not essential. I added this method is mainly for the high usability for the view module to display the status of the game.
- EndCondition interface provides some to generality to this design when comes to solving the problem of switching objective. I choose it to be an interface instead of a generic module because, in Java, you can only implement multiple interfaces but only one inheritance. Also, it is easier to apply the strategy design pattern.
- The *gameStatus* method in *EndByMoves* and *EndByTime* violates the principle of minimality. I design this module in this way is to ensure there is no delay or friction with the model module since the method updates the status of the game after each move made by the user during runtime.
- The two constructors in BoardT improve the flexibilty of the module. The user can choose to play with a board initialized with randomly generated dots or a board that is customized or pre-defined. Also, from a testing perspective, methods can be easily tested if the board is pre-defined compared to a randomly generated board.
- The test cases are designed to validate the correctness of the program based on the requirement and reveal errors or unusual behavior during program execution, every access routine has at least one test case. One exception is made for *replaceEliminated* method in BoardT because the *replaceEliminated* method adds a random dot to the board, there are no efficient ways to test the correctness of adding a randomly generated cell.
- In *BoardT*, the result of *remove* and *replaceEliminated* method is tested using *getMessage* method. These methods do not output anything and *getMessage* method stores the information of the scoreboard, it updates after each successful dot removal.

- 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 $TestBoardT.java$

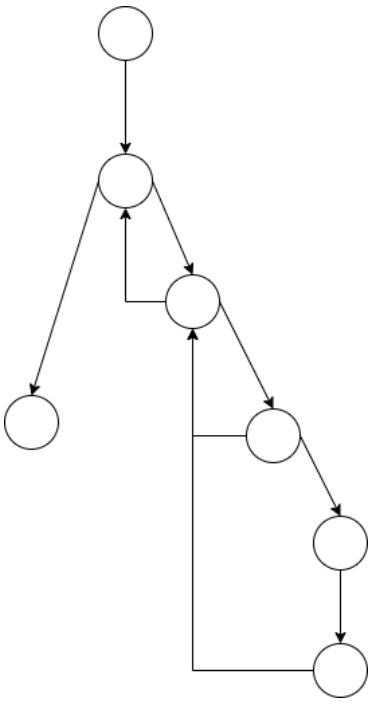
- Using MVC also makes my design maintainable and reduces risk when making changes. MVC decomposes into three component based on the separation of concerns where the model component encapsulates the internal data and status of the game, the view displays the state of the game, and the controller handles the input actions to execute related actions to respond to events.

- My design achieves high cohesion and low coupling by applying MVC. MVC keeps high cohesion since it groups related functionalities within each module. The deisgn is also low coupling because the modules (model, view, controller) are mostly independent of each other. So a change in one of the modules does not heavily impact the other.

- A strategy design pattern provides convenience and flexibility when comes to design for change. It is easier to switch between different algorithms during runtime through polymorphism with the Strategy pattern. In addition, it increases maintainability and readability in a way that the concerns are separated into classes instead of using conditional statements to switch strategy in runtime.

- I found using the Singleton design pattern is better than using static methods for abstract objects. 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.

Q2: Draw a control flow graph for bubble sort algorithm



The control flow graph is constructed using https://app.diagrams.net/