Assignment 4, Design Specification

Abhiraj Gogia

COMPSCI 2ME3

April 12, 2021

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game *2048*. At the start of each game, begins with a 4×4 grid of squares, sparsely filled with numbers. The player can then shift all the numbered squares in the grid to the right, left, upward, or downward. If numbers of the same value collide, they combine to form a larger number. After each move made by the player, new numbers will randomly spawn on the grid. The game ends when the player can no longer make any moves which change the position of the board (i.e. every block is stuck in place, as none of them can combine). The game will always terminate, but it's the player's job to last as long as they can and achieve the highest possible score. Throughout this specification document, the grid of squares may be referred to as a matrix or board interchangeably. Furthermore, an empty square on the board will be one that holds a value of zero. In addition to this, the row number of the grid increases when going from top to bottom, and the column number of the grid increases when going from left to right. The game uses a graphic user interface (GUI), therefore, it cannot simply be run in terminal on Mills. Thus, to run the game, run the 'Main' Module from your integrated development environment (IDE).

Overview of the design

This design applies a variation of the Model-View-Controller (MVC) design pattern where the controller is integrated into the View Module. The MVC design pattern was implemented in the following way: the module *BoardStatus* stores the state of the board and takes care of any manipulation of the board elements. An ASCII *Viewer* module displays the state of the board in terminal. A Viewer/Controller Module by the name of *InputReader*, records and processes user input and displays it on a GUI.

Likely Changes my design considers:

- Adjustable UI to allow for varying colour modes.
- Room for adding addition inputs that execute actions (like a mouse click or other key inputs).

Board State ADT Module

Template Module

BoardStatus

Uses

InputReader, Random

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getBoardValues		Integer[ ][ ]	
verticalCombine	$\mathbb{N} \times \mathbb{N} \times \mathbb{N}$		
horizontalCombine	$\mathbb{N} \times \mathbb{N} \times \mathbb{N}$		
shiftRight			
shiftLeft			
shiftUp			
shiftDown			
newElement			
increaseScore	N		

Semantics

State Variables

boardValues : sequence of sequence of $\mathbb{N}$

score : $\mathbb{N}$

State Invariant

None

Assumptions

- None

Access Routine Semantics

BoardStatus():

- transition: boardValues := $\langle i : \text{Integer} \mid i \in \text{boardValues} : 0 \rangle$

  boardValues[i][j] := 2

  boardValues[k][l] := 2

  # Where i, j, k, l are random integers between 0 and 3 using the nextInt() function in the Random module and $\neg(i \equiv k \wedge j \equiv l)$

- output: *out* := *self*

- exception: None

getBoardValues():

- output: *out* := *boardValues*

- exception: none

verticalCombine(*index1, index2, column*):

- transition: $\langle$ boardValues[index1][column] $\equiv$ boardValue*s*[index2][column] $\Rightarrow$ boardValues[index1][column] := 0 $\wedge$ boardValue*s*[index2][column] := boardValue*s*[index2][column] $\times$ 2 $\wedge$ increaseScore(boardValue*s*[index2][column] $\times$ 2) $\wedge$ inputReader.setText("2048 | Score: " + BoardStatus.getScore()) | boardValues[index1][column] $\neq$ 0 $\wedge$ boardValue*s*[index2][column] $\equiv$ 0 $\Rightarrow$ boardValues[index2][column] :=boardValues[index1][column] $\wedge$ boardValues[index1][column] := 0$\rangle$

- exception: none

horizontalCombine(*index1, index2, row*):

- transition: $\langle$ boardValues[row][index1] $\equiv$ boardValue*s*[row][index2] $\Rightarrow$ boardValues[row][index1] := 0 $\wedge$ boardValue*s*[row][index2] := boardValue*s*[row][index2] $\times$ 2 $\wedge$ increaseScore(boardValue*s*[row][index2] $\times$ 2) $\wedge$ inputReader.setText("2048 | Score: " + BoardStatus.getScore()) | boardValues[row][index1] $\neq$ 0 $\wedge$ boardValue*s*[row][index2] $\equiv$ 0 $\Rightarrow$ boardValues[row][index2] :=boardValues[row][index1] $\wedge$ boardValues[row][index1] := 0$\rangle$

- output : none

- exception: none

shiftRight():

- transition: $\langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [0..2] : \text{horizontalCombine}(j, j+1, i) \rangle \rangle \wedge$ newElement()

- output: None

- exception: None

shiftLeft():

- transition: $\langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [2..0] : \text{horizontalCombine}(j, j-1, i) \rangle \rangle \wedge$ newElement()
- output: None
- exception: None

shiftDown():

- transition: $\langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [0..2] : \text{verticalCombine}(j, j+1, i) \rangle \rangle \wedge$ newElement()
- output: None
- exception: None

shiftUp():

- transition: $\langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [2..0] : \text{verticalCombine}(j, j-1, i) \rangle \rangle \wedge$ newElement()
- output: None
- exception: None

boardFull():

- output := $\langle i : \text{Integer} \mid i \in [0..3] : j : \text{Integer} \mid j \in [0..3] : \text{boardValue}[i][j] \equiv 0 \mid \Rightarrow False \mid True \rangle$
- exception: None

newElement():

- transition := $\langle boardFull() \Rightarrow \langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [0..2] : \text{boardValues}[i][j] \equiv \text{boardValues}[i][j+1] \vee$ boardValues$[j][i] \equiv$ boardValues$[j+1][i] \Rightarrow$ break | inputReader.setText("2048 | Score: " + BoardStatus.getScore() + " | Game Over")$\rangle \rangle$ | boardValues$[i][j] := k$ # where i, j are random integers between $0 and 4$ selected by Random.nextInt(), where the value of boardValues$[i][j] \equiv 0$, and is set to k, which is a random integer 2 or 4 that is also selected by Random.nextInt(). This has a 90% chance of being a 2 and a 10% chance of being a 4. $\rangle$

increaseScore(*value*):

- transition: score := score + *value*
- out: None
- exception: None

Local Function:

- setBoardValues: seq of seq of Integers → void

  setBoardValues(*values*): boardValues := values

- setCell: Integer × Integer × Integer → void

  setCell(*value, rowIndex, columnIndex*) : boardValues[*rowIndex*][*columnIndex*] := *value*

- setScore: Integer → void

  setScore(*values*): score := values

ASCII Viewer Module

Module

Viewer

Uses

BoardStatus

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
updateValues			
printBoard			

Semantics

State Variables

values : sequence of sequence of N

State Invariant

None

Assumptions

- None

Access Routine Semantics

Viewer():

- transition : values := BoardStatus.getBoardValues()
- output := self
- exception: None

updateValues():

- transition: values := BoardStatus.getBoardValues()

printBoard():

- out: $\langle i : \text{Integer} \mid i \in [0..3] : \langle j : \text{Integer} \mid j \in [0..3] : \text{print}(\text{values}[i][j] + \text{“ ”}) \rangle \text{ print}(\text{new line}) \rangle$

GUI Viewer and Controller Module

Module

InputReader

Uses

BoardStatus

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
normalize	$\mathbb{N}$	$\mathbb{N}$	
fontChoose	$\mathbb{N}$	Colour	
colourChoose	$\mathbb{N}$	Colour	
setText	String		
setVal	$\mathbb{N} \times \mathbb{N} \times \mathbb{N}$		

Semantics

Environment Variables

frame: A portion of computer screen to display the game.

table : A portion of the frame that holds and displays data from BoardStatus.

itemLabel : A label at the bottom of the frame, which is used to display score and messages.

State Variables

size : N

data : sequence of sequence of N

Assumptions

- Assumes that GUI will be run after a BoardStatus() ADT is initialized.

Access Routine Semantics

inputReader():

- Set size state variable to be used as a relative size throughout the module.
- Initialize frame, table, and itemLabel environment variables.
- Initialize sizes, font sizes, font colours, and background colours for frame, table, and itemLabel.
- Set data equal to BoardStatus.getBoardValues().
- Initialize KeyListener() to read controller inputs.

normalize(*value*):

- out : $n := \langle\ value \neq 0 \Rightarrow n = \frac{log(value)}{log(2)} \mid n = 0\rangle\ \wedge \langle$ while $n > 15 \Rightarrow n = n - 16 \wedge \langle n \equiv 0 \Rightarrow n = n + 1\rangle\rangle$

fontChoose(*value*);

- out: Colours[normalize(value)] # where Colours is a locally initialized list of RGB colours

colourChoose(value);

- out: Colours[normalize(value)] # where Colours is a locally initialized list of RGB colours

setText(*text*):

- inputLabel.text := *text*

setVal(*value, rowIndex, columnIndex*):

- table.data[*rowIndex*][*columnIndex*] := value

Local Type and Local Type Functions

KeyListener:

KeyListener.keyReleased: KeyEvent ⇒ void

KeyListener.keyReleased(*event*) : $\langle e$: KeyEvent | $e \equiv$ KeyEvent.Up ⇒ BoardStatus.shiftUp() ∧

setVal(BoardStatus.getBoardValues() | $e \equiv$ KeyEvent.Down ⇒ BoardStatus.shiftDown() ∧

setVal(BoardStatus.getBoardValues() | $e \equiv$ KeyEvent.Right ⇒ BoardStatus.shiftRight() ∧

setVal(BoardStatus.getBoardValues() | $e \equiv$ KeyEvent.Left ⇒ BoardStatus.shiftLeft() ∧ setVal(BoardStatus.getBoardValues()

| *void*⟩

KeyListener.setVal : sequence of sequence of $\mathbb{N} \rightarrow$ void

KeyListener.setVal(values) : $\langle i$: Integer | $i \in [0..3] : \langle j$: Integer | $j \in [0..3]$: table.setValueAt(values[i][j], i, j)⟩⟩

Random Module

Module

Random

Uses

None

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
nextInt	$\mathbb{N}$	$\mathbb{N}$	

Semantics

State Variables

None

Assumptions

- None

Access Routine Semantics

nextInt(upperBound):

- out: returns a pseudorandom uniformly distributed int value between 0 (inclusive) and upperBound (exclusive).

Critique of Design

- The Controller and View Modules are specified as a single abstract object as there is a significant amount of interweaving between the two. Their resources are also shared, as the Controller module requires a KeyListener to read inputs and the View module requires the KeyListener to be embedded into it's frame. Thus, it is most efficient to combine the two. Furthermore, this diminishes any possibilities of conflicting changes to these overlapping resources.

- The setCell() local function in the BoardState Module is non-essential, as one can set individual values using setBoard-Values. However, this function was made keeping testing in mind, as it would greatly aid with debugging specific cases.

- Many methods across several modules, such as verticalCombine()/horizontalCombine() in BoardStatus and a majority of the GUI methods violate the principle of minimality. This was purposely done for two reasons. The first was to ensure there is no delay between operations, as the smoothness of the animations of the GUI heavily depend on these functions. Simultaneously while these methods perform operations, they update the screen or components of the screen, therefore it is paramount to avoid delays. The second reason was to condense the amount of code being written. By Breaking every method into several smaller methods, it would've greatly increased the amount of code that needed to be written, as most functions have unique code that is not reused.

- The setBoardValues() and setCell() functions in BoardStatus provide flexibility for testing, as typically a new board is initialized with randomized values. However, for testing purposes, random values do not work, as generally speaking tests are supposed to validate the correctness of the program and reveal mistakes. Randomness does not allow for reproduce-able results, therefore, it is not useful for testing. Thus, having the ability to set values allows for there to be a predetermined correct results, which allows for much better testing.

- The Viewer module also increases flexibility in terms of testing. It was designed such that tests for BoardStatus could be visualized when using the GUI is unfeasible, such as on Mills.

- Given how finicky the design for the GUI is, there is a lack of generality for the game, as the board size is locked to be 4×4. Future considerations for this design would be to apply this concept of generality towards the GUI, such that games with smaller or larger boards could be played.

- Did not build any test cases for the GUI and Controller, as they would require the GUI to be running while the tests occur. Thus, different scenarios were manually tested to confirm correctness. Any additional test cases for the BoardState were located in *TestBoardState.java*.

- Using the MVC design pattern allows for the design to be easily maintainable for the most part, as the design is splits up the three components by separation of concerns. BoardStatus is encapsulates the board data and state, while the Viewer/Controller Module handle viewing the game, processing inputted commands, and executing any actions related to those commands.

- Aside from the occasional call from BoardStatus to the InputReader module to update the GUI and calls from InputReader to BoardStatus to get the Board's values, both modules have relatively low coupling as all of their functionalities are independent of each other. Most changes to one would theoretically not impact the other. An exception to this would be of course if things like the board's size in BoardStatus was changed. This would require the dimensions of the table of InputReader to be changed as well, but excluding this, very few other changes would need to be made that aren't visual.

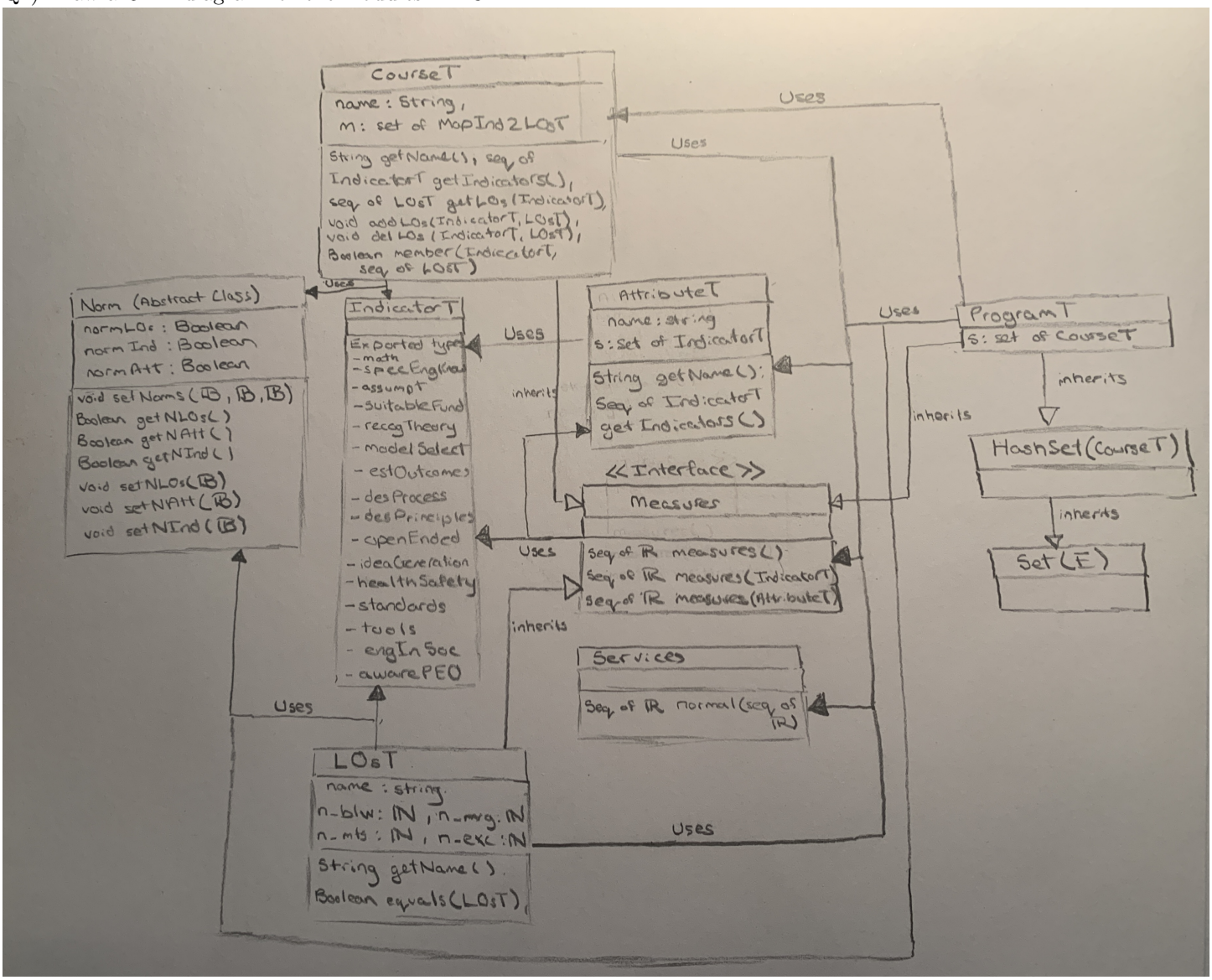
- Within BoardStatus in particular there is a high degree of cohesion, especially between the board manipulating methods, as they rely on using other methods such as the horizontal and vertical combiners as a basis for their algorithms.

- The design specification was made with consistency in mind. Thus, consistent parameter ordering and naming schemes were used.

- A weak point of this design is Information Hiding. Given how intertwined the View/Controller Module and the Board State Module are with one another, in order to increase the ease of design, the modules can essentially see every aspect of each other. Nevertheless, when accessing information from each other, proper getter functions were used.

Answers to Questions:

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



Q2) Draw a control flow graph for the convex hull algorithm.