# Assignment 4, Part 1, Specification

Ethan Vince-Budan

April 12, 2021

This Module Interface Specification document contains modules, types and methods for implementing the game *2048*. 2048 is a game about merging tiles of similar values, and trying to reach the highest possible score before getting stuck. The player can do this by shuffling the tiles on the board in one of four directions: up, down, left and right. On each turn, the player decides which way they would like to move, and every tile on the board attempts to move as far as possible in that direction. If two tiles of similar value collide during their move, they can merge into a tile of higher value. This rewards the player with some points. At the end of each turn, a new low-value tile is placed somewhere random on the board. The game is over once no more tiles can be moved, and no more merges can be made. The game can be launched and played by using the command **make demo** in a terminal.

Throughout this document, a *row* is a horizontal line of tile from end to end of the board. Similary, a *column* is a vertical line. A cell or tile is an individual piece on the board, which must occupy one of the 16 spaces on the board. In this implementation, an empty cell is a cell with a value of zero.

## 1 Overview of the Design

### 1.1 Description of Classes

This implementation uses a model-view-controller (MVC) design pattern, for controlling the graphical user interface and updaing the board during each turn. The view module, labeled as $GUI$, provides the graphical interface to the user, as well as a means for them to input instructions into the game. Display of the board itself is handled by a seperate class called a BoardDisplay, which holds all the specific information such as colors and coordinates which are used to display the tiles. Input from the user is captured by the $GUI$ object and sent to the controller portion of MVC.

The Controller handles the input given by the user, decoding the user's intentions from their input and sending commands to the model portion of the MVC pattern. The Controller also tells the GUI to update whenever a move has been completed, so the user can have the correct information displayed on the screen. The controller acts as the bridge between the GUI and the model, which is where all the actual state information for the game is stored.

The Model handles the manipulation of data, and stores anything state related about the game. To store the actual board data, a seperate class BoardStorage stores the board information, while the manipulation of this board data is done throught the Movement class, which holds the algorithms necessary to modify the board data during a turn. A Score class is invoked by the Movement class to update the scores stored in the Board class, and could potentially hold many different types of scoring schemes for different modes of gameplay.

# 2 Module Interface Specification

# Directions Module

## Module

Direction

## Uses

None

## Syntax

### Exported Constants

None

### Exported Types

Directions = {
UP,
DOWN,
LEFT,
RIGHT
}

### Exported Access Programs

None

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions

None

### Considerations

This class should be implemented as an enumerated type.

# Board Storage Module

## Template Module

BoardStorage

## Uses

None

## Syntax

### Exported Constants

None

### Exported Types

BoardStorage = ?

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| new BoardStorage | $\mathbb{N}, \mathbb{N}$ | BoardStorage | IllegalArgumentException |
| setRow | $\mathbb{N}$, seq of $\mathbb{Z}$ | | IllegalArgumentException |
| setCol | $\mathbb{N}$, seq of $\mathbb{Z}$ | | IllegalArgumentException |
| getWidth | | $\mathbb{N}$ | |
| getHeight | | $\mathbb{N}$ | |
| getBoard | | seq of (seq of $\mathbb{Z}$) | |
| isFull | | $\mathbb{B}$ | |
| addRandomTiles | $\mathbb{N}$ | | ArrayStoreException |
| getTileAt | $\mathbb{N}, \mathbb{N}$ | | |

## Semantics

### State Variables

$contents$ : seq of (seq of $\mathbb{Z}$)
$width$ : $\mathbb{N}$
$height$ : $\mathbb{N}$

### State Invariant

$|contents| = height \wedge |contents[0]| = width$

### Assumptions

None

**Access Routine Semantics**

new BoardStorage($x$, $y$):

- transition: *width*, *height*, *contents* := $x, y, \langle \forall j : \mathbb{N} | j \in [0..y-1] : \langle \forall i : \mathbb{N} | i \in [0..x-1] : 0 \rangle \rangle$ # Initialize a 2D-array of zeros.
- output: *out* := *self*
- exception: exc := $((x \leq 1 \vee y \leq 1) \Rightarrow$ IllegalArgumentException)

setRow($n$, *row*):

- transition: *contents*[n] := *row* # Replace row n with *row*.
- exception: exc := $(n \notin [0..height - 1] \vee |row| \neq width) \Rightarrow$ IllegalArgumentException

setCol($n$, *col*):

- transition: *contents* := $\forall y : \mathbb{N} | (y \in [0..height - 1] \Rightarrow contents[y][n] = col[y])$ # Replace every value in column n with corresponding value from *col*.
- exception: exc := $(n \notin [0..width - 1] \vee |col| \neq height) \Rightarrow$ IllegalArgumentException

getWidth():

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

getHeight():

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

getBoard():

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

isFull():

- output: *out* := $\wedge \{y : \mathbb{N} | y \in [0..height-1] : \{x : \mathbb{N} | x \in [0..width-1] : contents[y][x] \neq 0\}\}$ # Returns *True* if all cells are nonzero
- exception: None

addRandomTiles($n$):

- transition: $\forall i : \mathbb{N} | i \in [0..n] : contents[Random.next() * height][Random.next() * width] = 0 \Rightarrow contents[Random.next() * height][Random.next() * width] = (Random.next() < 0.9 \Rightarrow 2 | True \Rightarrow 4)$ # Add n tiles to the board in random empty positions, with a 10% chance of being a 4 and a 90% chance of being a 2.
- exception: $exc := (isFull() \Rightarrow$ ArrayStoreException)

getTileAt($x$, $y$)

- output: *out* := $((x \in [0..width - 1] \wedge y \in [0..height - 1]) \Rightarrow contents[y][x]) | (True \Rightarrow -1)$ # Return value of tile at position, if position out of bounds return -1
- exception: None

# Movement Algorithms Module

## Module

Movement

## Uses

Direction, Score

## Syntax

### Exported Constants

None

### Exported Types

None

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| shift | Direction, seq of $\mathbb{Z}$ | seq of $\mathbb{Z}$ | |
| moveMade | | $\mathbb{B}$ | |
| clearMoveMade | | | |

## Semantics

### State Variables

$moveMade : \mathbb{B}$

### State Invariant

None

### Assumptions

the *moveMade* state variable should be initialized to *False*.

### Access Routine Semantics

moveMade():

- output: $out := moveMade$
- exception: None

clearMoveMade():

- transition: $moveMade := false$

- exception: None

shift(*dir*, *seq*):

- output: $out := (dir = Direction.UP \vee dir = Direction.LEFT) \Rightarrow shiftLeft(seq) | (dir = Direction.DOWN \vee dir = Direction.RIGHT) \Rightarrow shiftRight(seq)$

## Local Functions

shiftLeft: seq of $\mathbb{Z} \rightarrow$ seq of $\mathbb{Z}$
shiftLeft($seq$) $\equiv (\forall i : \mathbb{N} | i \in [0..|seq| - 1] : (\forall j : \mathbb{N} | j \in [1..i] : ((seq[i-1] = seq[i] \wedge \neg merged[i-1]) \vee seq[i-1] = 0) \Rightarrow (seq[i-1] := seq[i-1] + seq[i], Score.updateScore(seq[i-1] - seq[i]), seq[i] := 0)))$

# For every entry in *seq*, if the entry to the left of it is equal in value or zero, add the cells values together, update the game score using their difference, and erase the original position. Since empty cells have a value of zero, merging a cell with an empty one results in 0 additional score. NOTE: This algorithm should NOT be greedy, that is cells can only merge once. this could be acheived in many ways, but here we are using a sequence of boolean values *merged*.

shiftRight: seq of $\mathbb{Z} \rightarrow$ seq of $\mathbb{Z}$
shiftRight($seq$) $\equiv (\forall i : \mathbb{N} | i \in [0..|seq| - 1] : (\forall j : \mathbb{N} | j \in [i..1] : ((seq[i+1] = seq[i] \wedge \neg merged[i+1]) \vee seq[i+1] = 0) \Rightarrow (seq[i+1] := seq[i+1] + seq[i], Score.updateScore(seq[i+1] - seq[i]), seq[i] := 0)))$

# This is the same algorithm as above, but cells are merged to the right instead of to the left. Again, this algorithm should not be greedy, and cells should only merge once. Note that shiftRight starts merging from the rightmost element and traverses leftwards, while shiftLeft starts merging from the leftmost and traverses rightwards.

# Score Calculation Module

## Module

Score

## Uses

Board

## Syntax

### Exported Constants

None

### Exported Types

None

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| updateScore | $\mathbb{Z}$ | | |

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions

None

### Access Routine Semantics

updateScore(*val*):

- transition: Board.*setScore*(*val*)
- exception: None

# Game State Module (Abstract Object)

## Module inherets BoardStorage

Board

## Uses

BoardStorage, Movement, Direction

## Syntax

### Exported Constants

None

### Exported Types

None

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| init | $\mathbb{N}, \mathbb{N}$ | | IllegalArgumentException |
| move | Direction | | |
| getScore | | $\mathbb{Z}$ | |
| getHighScore | | $\mathbb{Z}$ | |
| setScore | $\mathbb{Z}$ | | |
| movesPossible | | $\mathbb{B}$ | |
| getBoard | | seq of (seq of $\mathbb{Z}$) | |

## Semantics

### State Variables

*score* : $\mathbb{Z}$
*highScore* : $\mathbb{Z}$
*contents* : BoardStorage

### State Invariant

None

### Assumptions

- The init access program will be called before any other access program
- Any parameters passed to the access programs will be of the correct type.

**Access Routine Semantics**

init($x$, $y$):

- transition: *score*, *highScore*, *contents* := $0, 0,$ new BoardStorage($x$, $y$)
- exception: $((x \leq 1 \vee y \leq 1) \Rightarrow IllegalArgumentException)$

move($dir$):

- transition: $(dir = Direction.UP \vee dir = Direction.DOWN) \Rightarrow (\forall i : \mathbb{N} | i \in [0..contents.getWidth() - 1] : contents.setCol(i, Movement.shift(dir, contents.getCol(i))))$
  $|(dir = Direction.LEFT \vee dir = Direction.RIGHT) \Rightarrow (\forall i : \mathbb{N} | i \in [0..contents.getHeight() - 1] : contents.setRow(i, Movement.shift(dir, contents.getCol(i))))$
  # If direction is vertical, update every column to be shifted in the given direction. If direction is horizontal, update every row to be shifted in the given direction.
- exception: None

getScore():

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

getHighScore():

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

setScore($s$):

- transition: *score*, *highScore* := $s, (s > highScore \Rightarrow s) | True \Rightarrow highScore$
- Exception: none

movesPossible():

- output: *out* := $(\neg contents.isFull()) \Rightarrow True | True \Rightarrow (\forall y : \mathbb{N} | y \in [0..|contents.getBoard()| - 1] : (\forall x : \mathbb{N} | y \in [0..|contents.getBoard()[0]| - 1] : ((contents.getTileAt(x, y-1) = contents.getTileAt(x, y) \Rightarrow True | (contents.getTileAt(x, y + 1) = contents.getTileAt(x, y) \Rightarrow True | (contents.getTileAt(x - 1, y) = contents.getTileAt(x, y) \Rightarrow True | (contents.getTileAt(x+1, y) = contents.getTileAt(x, y) \Rightarrow True) | True \Rightarrow False))$
  # If board is not full, immediately return True. If it is full, check to see if any tiles have neighbors of the same value. If this is true, return True. If no tiles have same-valued neighbors, return False.
- exception: None

getBoard():

- output: *out* := $\{y : \mathbb{N} | y < height : contents.\text{getRow}(i)\}$
- exception: None

# Controller Module

## Template Module inherets KeyListener, ActionListener

Controller

## Uses

Board, KeyEvent, KeyListener, ActionListener, ActionEvent

## Syntax

### Exported Constants

None

### Exported Types

Controller = ?

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| new Controller | | Controller | |
| keyPressed | KeyEvent | | |
| isGameOver | | | |
| actionPerformed | ActionEvent | | |

## Semantics

### State Variables

*parent* : GUI
*controlls* : MapCharToDir # Mapping of input character to Direction output
*gameOver* : $\mathbb{B}$
*boardWidth* : $\mathbb{N}$
*boardHeight* : $\mathbb{N}$

### State Invariant

None

### Assumptions

None

**Access Routine Semantics**

new Controller($g$):

- transition: $parent, controlls, gameOver, boardWidth, boardHeight := g, newMapCharToDir, false, 4, 4$
- output: $out := self$
- exception: None

keyPressed($e$):

- transition: ($controlls$.get($e$) $\neq null \Rightarrow$ Board.$move$($controlls$.get($e$)))
- transition: $gameOver = \not Board.movesPossible()$
- exception: None
- considerations: This method inhereted from the KeyListener class, as implemented in Java.

isGameOver():

- output: $out := gameOver$
- exception: None

actionPerformed($e$):

- transition: $gameOver := false$
- transition: $Board.init(boardWidth, boardHeight)$
- exception: None

## Local Types

MapCharToDir: tuple of (char, Direction)

# Board Display Module

## Template Module

BoardDisplay

## Uses

Board, JPanel, Graphics

## Syntax

### Exported Constants

None

### Exported Types

BoardDisplay = ?

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| new BoardDisplay | | BoardDisplay | |
| paintBoard | Graphics, $\mathbb{N}, \mathbb{N}, \mathbb{N}, \mathbb{N}$ | | |

## Semantics

### State Variables

*colours* : MapInt2Color # Mapping of tile values to colours

### State Invariant

None

### Assumptions

The displayBoard() program will only be called once the Board module has been initialized.

### Access Routine Semantics

new BoardDisplay():

- transition: *colours* := new MapInt2Colour()
- output: *out* := *self*
- exception: None

paintBoard($g, x, y, w, h$):

- transition: Display the board on the GUI using Board information gathered from Board.getBoard(). The parameter $g$ holds the Graphics object to draw to, $x$ and $y$ hold the coordinates of the top-left corner of the board display area, and $w$ and $h$ hold the total width and height of the parent window.
- exception: None

# View Module

## Template Module inherets JPanel

GUI

## Uses

Controller, BoardDisplay, JPanel, JLabel KeyEvent, Graphics, Color

## Syntax

### Exported Constants

None

### Exported Types

GUI = ?

### Exported Access Programs

| Routine Name | In | Out | Exceptions |
|---|---|---|---|
| new GUI | | | |
| paintComponent | Graphics | | |
| sendKeyEvent | char | KeyEvent | |
| sendActionEvent | | ActionEvent | |
| main | seq of String | | |

## Semantics

### State Variables

*controller* : Controller
*scoreLabel* : JLabel
*highScoreLabel* : JLabel
*gameOver* : JLabel
*boardDisp* : BoardDisplay

### State Invariant

None

### Assumptions

None

**Access Routine Semantics**

new GUI():

- transition: $controller, scoreLabel, highScoreLabel, gameOver, boardDisp :=$ $newController(), newJLabel(\text{“Score: 0”}), newJLabel(\text{“High Score: 0”}), newJLabel(\text{“Some moves remain”}),$ $newBoardDisplay()$
- output: $out := self$
- exception: None

paintComponent($g$):

- transition: Display the user interface on the screen, as well as the BoardDisplay. Also update *scoreLabel*, *highScoreLabel* and *gameOver* to reflect the actual state of the game using the Board.getScore, Board.getHighScore and Controller.isGameOver() methods.
- exception: None
- considerations: This method inhereted from the JPanel class, as implemented in Java.

sendKeyEvent($c$):

- output: Creates a KeyEvent object and sends it to any registered KeyListeners. In this case, that will be a Controller instance.
- exception: None
- considerations: This method is inhereted from the JPanel class, as implemented in Java.

sendActionEvent

- output: Creates an ActionEvent object and sends it to any registered ActionListeners. In this case, that will be a Controller instance.
- exception: None
- considerations: This method is inhereted from the JPanel class, as implemented in Java.

main(*args*):

- The entrypoint of the program. In this method, a new JFrame object must be created to hold an instance of a GUI, as well as an instance of the BoardDisplay class. Additionally, an instance of the Controller class must be registered with this JFrame as a KeyListener and ActionListener for any keyboard inputs/button presses to be detected by the game.

## KeyEvent Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## KeyListener Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## JPanel Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## JLabel Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## Graphics Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## ActionEvent Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## ActionListener Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## Font Module

### Considerations

Implemented as part of Java, as described in the Oracle Documentation

## Color Module

### Considerations

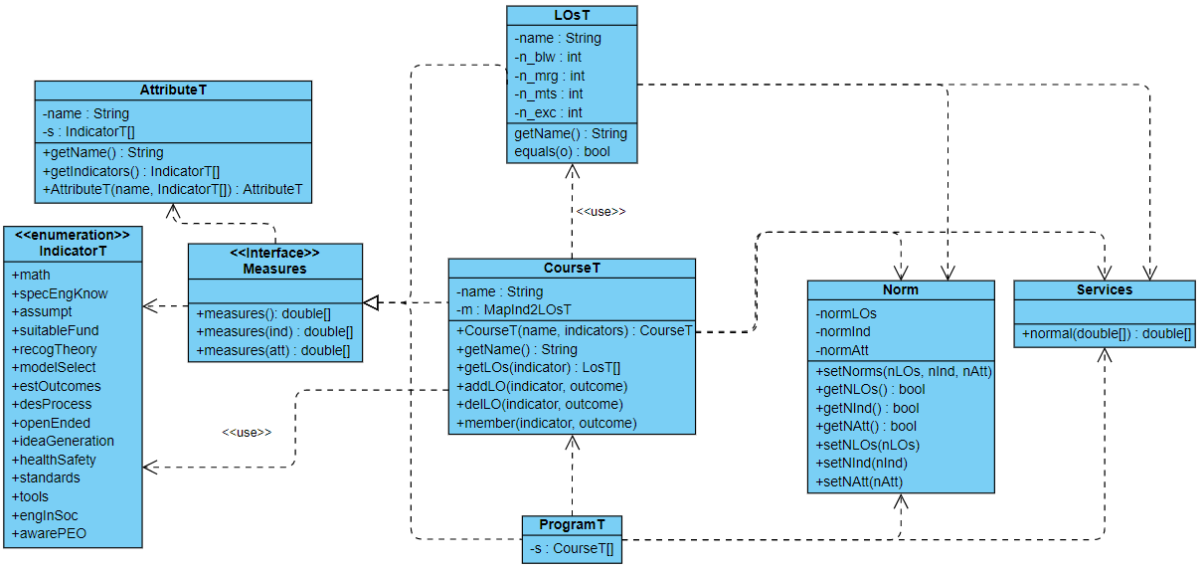
Implemented as part of Java, as described in the Oracle Documentation

# 3 Critique of Design

- Choosing to specify the Board class as an abstract object causes issues with information hiding, as the getBoard() method could potentially provide means through which to modify the board in unexpected ways. To mitigate this, the getBoard() method returns only a copy of the board data, which has no potential to cause modifications to the original board. This helps maintain a good level of information hiding.
- Moving all score information from the Board class into the Score class could provide a better level of cohesion within the modularization of this program, and also increase information hiding
- Implementing a Tile class instead of using integer values for board information storage could greatly increase the generality of the BoardStorage and Movement modules, and allow for potential changes in tile values to be more easily implemented.
- The interface for the BoardStorage class could potentially be improved by removing the getBoard() method and replacing it with getRow() and getCol() methods. This would improve the consistency and essentiality of the design, along with increasing information hiding.
- This design exhibits low minimality, as there are many methods which perform multiple operations. The main issue with creating a minimal design is that many game state variables must be updated each turn, as a result of the turn itself. For example, the number of points you score is dependent on the number of tiles you merge. Thus when performing a move, the method must not only mutate the game board, but also indicate the number of points to be given. The design decision in this case was to have method calls from within methods, to increase the cohesion of the modules themselves. Alternatively, the method could return the number of points, and the caller could then use that value in a different method call. However, this still does not get around the issue of minimality, as the method must both mutate data and return a value.

# 4 Answers to Questions

### 1. UML Diagram for A3




## 2. Control Flow Graph