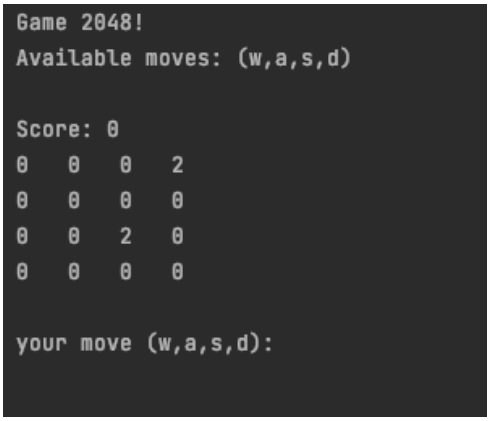
Assignment 4, Design Specification

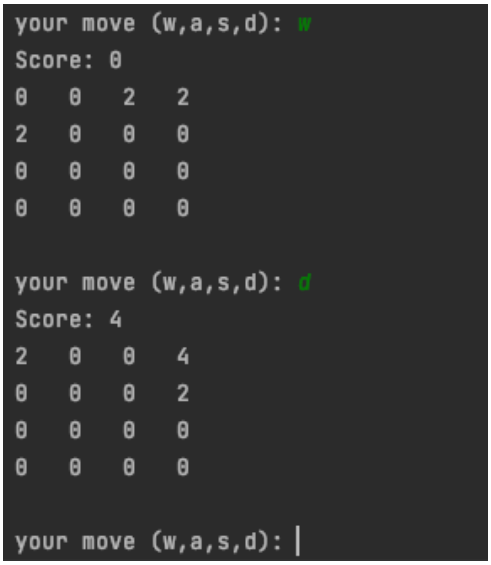
SFWRENG 2AA4

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, the user is provided with a Board of 4x4 tiles with any two random tiles with value 2 (90%) or 4 (10%), whereas all other tiles are initialized with a value of 0. The user can choose to make their move by typing "w , a, s, d", where 'w' is for moving and merging the tiles in upward direction, 'a' for left, 's' for down and 'd' for right. This will change the state of the Board and shift the tiles with non zero values towards the input direction. The Tiles with the same value merge into one when they touch and increase the total score of the user. Add them up to reach 2048! After every move that is feasible, one random tile with 0 value takes up the value of 2 or 4. Finally, the game can be launched and play by typing `make demo` in terminal.



Initial state of game



Score update when two tiles with value 2 merge

1 Overview of the design

This design applies Module View Specification (MVC) design pattern and Singleton design pattern. The MVC components are *Board* (model module), and *Game* (View and Controller module). Singleton pattern is specified and implemented for *Game*.

The MVC design pattern is specified and implemented in the following way: the module *Board* stores the state of the game board and the score of the user. A view module *Game* can display the state of the game board and game using a text-based graphics. *Game* is also responsible for handling input actions. For *Game*, use the main() method to run the game.

Likely Changes my design considers:

- Data structure used for user login to keep track of their highest score.
- The visual representation of the game such as UI layout.
- Change in game ending conditions to adjust the difficulty of the game.
- Modularization could be done with more precision.
- Implementing GUI to make it more interactive.

MoveT Module

Module

MoveT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

MoveT = {w, a, s, d}

//w is for merging in upper direction , a for left direction, s for downward direction, d for right direction

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Tile Module

Template Module

Tile

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
Tile		Tile	
Tile	$\mathbb{Z}$	Tile	IllegalArgumentException
getValue		$\mathbb{Z}$	
setValue	$\mathbb{Z}$		IllegalArgumentException
toString		String	

Semantics

State Variables

value: $\mathbb{Z}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new Tile():

- output: out := self
- transition: this.value = 0
- exception: None

new Tile(number):

- output: out := self
- transition: this.value = number
- exception: IllegalArgumentException

getValue():

- output: out := this.value
- exception: None

setValue(value):

- transition: this.value := value
- exception: IllegalArgumentException

toString():

- out: out := String.valueOf(value)
- exception: None

Board ADT Module

Template Module

Board

Uses

Tile

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
Board		Board	
Board	seq of (seq of Tile)	Board	
Board	$\mathbb{Z}$	Board	
getScore		$\mathbb{Z}$	
highestTile		$\mathbb{Z}$	
spawnRandom			
move	MoveT		IllegalArgumentException
gameWon		$\mathbb{B}$	
gameOver		$\mathbb{B}$	
up			
down			
vMerge			
left			
right			
hMerge			
toString		String	

Semantics

State Variables

board: Sequence of (Sequence of Tile)
edge: $\mathbb{Z}$
score: $\mathbb{Z}$

State Invariant

None

Assumptions

- The constructor Board is called for each object instance before any other access routine is called for that object.
- SpawnRandom needs to be called independently two time swhen initiating the game.
- Methods will be used wisely to control the flow of the game.

Access Routine Semantics

new Board():

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j : Integer | j < 4 ⇒ board[i][j] := new Tile())))
- output: *out* := *self*
- exception: None

new Board(*tt*):

- transition: ∀ (i: Integer | i < 4 ⇒ board[i] := tt[i])
- output: *out* := *self*
- exception: None

new Board(factor):

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j : Integer | j < 4 ⇒ board[i][j] := new Tile((*factor* + *i* + *j* + 1) ∗ (*i* + *j* + 1))))))
- output: *out* := *self*

- exception: None

getScore():

- transition: none
- output: *out* := this.score
- exception: None

highestTile():

- transition: none
- output: *out* := high, where
  $\forall$(i : Integer | i < 4 $\Rightarrow$ ($\forall$(j : Integer | j < 4 $\Rightarrow$ (board[i][j].getValue() > high $\Rightarrow$ high := board[i][j].getValue()))))
- exception: None

spawnRandom():

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

move(direction):

- transition: (direction == MoveT.w $\Rightarrow$ up() | (direction == MoveT.a $\Rightarrow$ left() | (direction == MoveT.s $\Rightarrow$ down() | (direction == MoveT.d $\Rightarrow$ right() | IllegalArgumentException))))
- output: None
- exception: IllegalArgumentException

gameWon():

- transition: None
- output: out := (highestTile() == 2048 $\Rightarrow$ true | false)
- exception: None

gameOver():

- transition: None

- output: out := (count == 16 ⇒ true | false), where
  count is the number of tiles which can't be combined with it's adjacent tiles.
- exception: None

up():

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j: Integer | j < 4 ⇒ (board[j][i].getValue() != 0 ⇒ (edge <= j ⇒ vMerge(j, i, "up")))))
- output: None
- exception: None

down():

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j = 3 : Integer | j >= 0 ⇒ (board[j][i].getValue() != 0 ⇒ (edge >= j ⇒ vMerge(j, i, "down")))))
- output: None
- exception: None

vMerge():

- transition: score is updated as any two tile merges in vertical direction
- output: None
- exception: None

left():

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j: Integer | j < 4 ⇒ (board[i][j].getValue() != 0 ⇒ (edge <= j ⇒ hMerge(i, j, "left")))))
- output: None
- exception: None

right():

- transition: ∀(i : Integer | i < 4 ⇒ (∀(j = 3 : Integer | j >= 0 ⇒ (board[i][j].getValue() != 0 ⇒ (edge >= j ⇒ hMerge(i, j, "right")))))
- output: None

- exception: None

hMerge():

- transition: score is updated as any two tile merges in horizontal direction
- output: None
- exception: None

toString():

- transition: None
- output: out := s, where
  $\forall$(i : Integer | i < 4 $\Rightarrow$ ($\forall$(j: Integer | j < 4 $\Rightarrow$ (s += board[i][j].toString() + " "))))
- exception: None

Critique of Design

- I've specified Board 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.
- All the methods added have high usability for the view module to display the status of the game. Therefore, the design pattern used is highly essential.
- The *move* method in *Board* preserves the principle of minimality as it combines the use of up, down, left and right methods. I design this module in this way is to ensure there is no delay or friction with the model module since the same method could serve the purpose of having 4 different methods.
- The three constructors in Board 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 *spawnRandom* method in Board because the *spawnRandom* method adds a random Tile to the board, there are no efficient ways to test the correctness of adding a randomly generated Tile.
- In *Board*, the result of *up*, *down*, $left$ and $right$ were tested using the total score generated from the merges in that move.
- 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 $TestBoard.java$
- The use of MVC also keeps my design safe and reduces the chance of transition. MVC decomposes into three separate components where the model component encapsulates the game's internal data and status, where the view shows the game status, and where the controller deals with the input behaviour to execute the relevant measures to react to the events.
- I architecture with the application of MVC achieve high cohesion and low coupling. The MVC maintains strong cohesion since within - module it groups similar features. The architecture is also low because of its mostly separate modules (model, display,

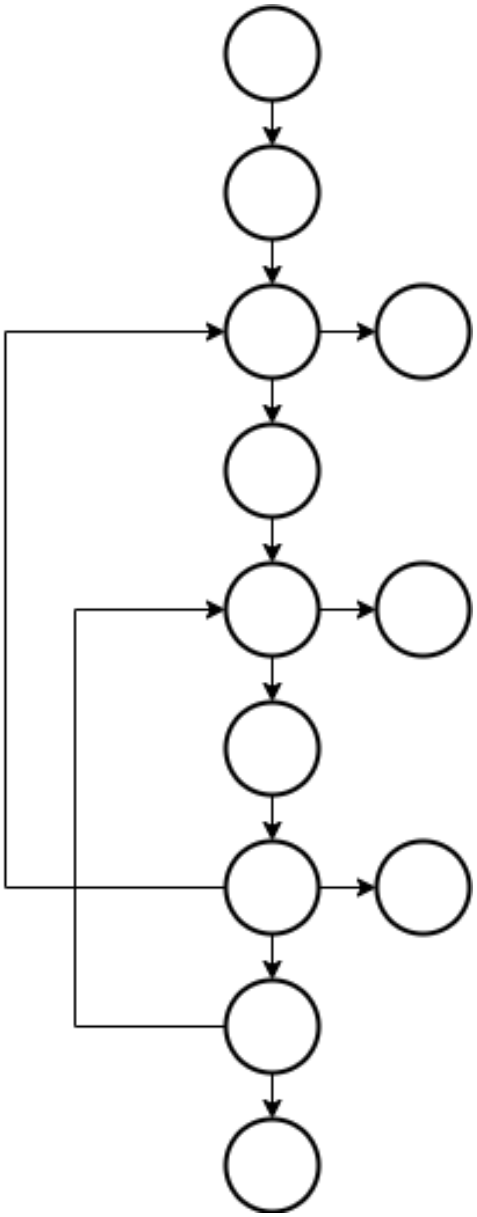
controller). A improvement in one of the modules thus has little significant influence on the other.

- I've found it easier to use Singleton designs than to use abstract object static approaches. During the development process certain alerts about the method or variables need to be accessed statically for the application of static methods and variables. For the singleton model, all these challenges are eliminated and a simpler development is achieved.

Answers to Questions:

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

Q2: Draw a control flow graph for convex hull algorithm.



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