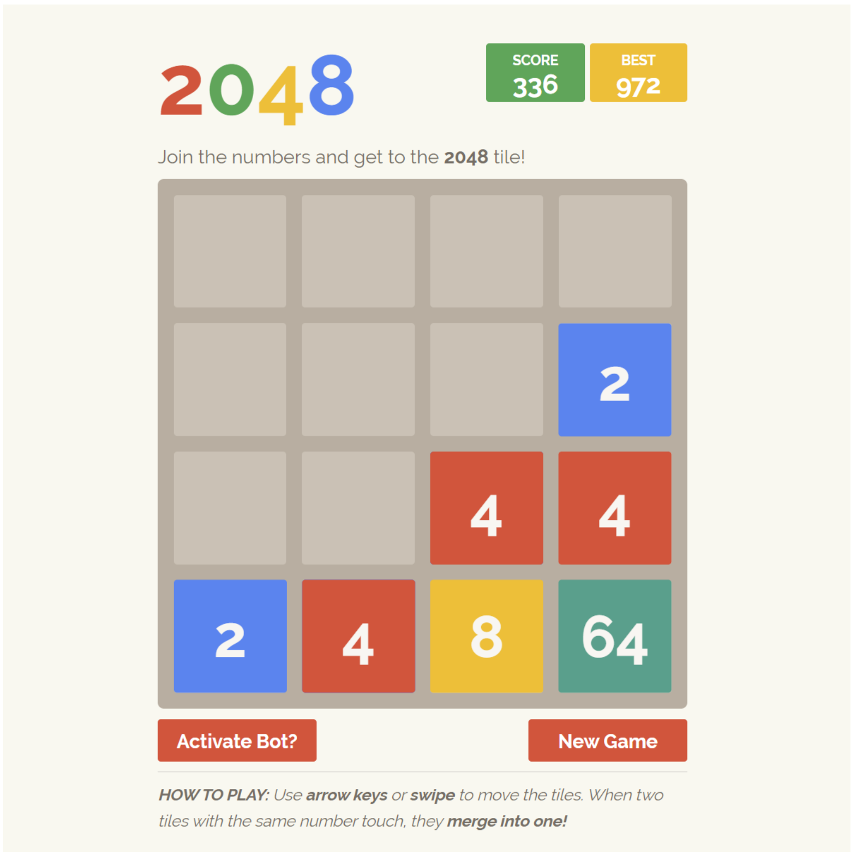
2048: Design Specification

Jishan Sharif

February 22, 2022

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game *2048*. The rules of the game are as follows. The game starts off with a $4x4$ board with two randomly generated values placed in any part of the board. The board is initially empty. The randomly generated values will either be a 2 or a 4. We can move either left, right, up or down and for each movement the numbered tile will move to the direction specified. Tiles with the same number merge into one when they touch. Add them up to reach 2048! The game can be launched and played by typing `make demo` in the terminal.


2048
SCORE
336
BEST
972
Join the numbers and get to the 2048 tile!
2
4
4
2
4
8
64
Activate Bot?
New Game
HOW TO PLAY: Use arrow keys or swipe to move the tiles. When two tiles with the same number touch, they merge into one!

The above board visualization is from https://elgoog.im/2048/

1 Overview of the design

Our first step was to initialize the board. The key thing to note about this game is that we start the game with two randomly placed tiles on a 4x4 board. The score is set to 0. The filled tiles can have values 2 or 4. It'd be best to create a new Object for a Tile in case we may want to optimize our game to work for different values.

Once we get the board initialized, The next steps was to get the movements to work. There are four movements to consider on our model, each of these movements can be trigerred using the Controller or the View module

For each movement, I've worked on it using the process of incrementality. The more I worked on each movement, the better I understood the game and the cases required. For the most part, each movement required precise understanding of board positions and when merging is possible. Furthermore, once this is complete - A point to consider is to check when the game is over. Since a game check would take a longer run time, it's best to make this check after one move is over. Hence, it was most appropriate to add this method onto our controller file.

The difference between this game I've created versus the one available to play online is that empty tiles are denoted as tiles containing the value 0. Any tile which does not contain 0 is a filled tile. In a real game, An empty tile would be a void space.

Likely Changes my design considers:

The next step would be to wire the game built to a GUI. This would mean refactoring the way the board looks by adding graphics and colours. The most important feature would be to remove the 0 tiles and add an empty tile. We can also add a points section above the board which updates the points as the game progresses. Currently, we can only view the points by entering the command p on our terminal.

The next thing we could do is make use of a Key Listener instead of asking a User for inputs. This would mean we can make use of arrow keys instead of letters such as r, l, u, d to play the game.

Although this is beyond the scope of this project, it would be ideal to have a previous score saved. This can be the $HighScore$ and anytime you beat that, a message can be displayed saying you have beaten the highest score, regardless of the fact whether you won the game or not. We can then update the value of the high score as required.

Tile Type Module

Module

TileT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

None

A Board will contain tiles, Each Tile can either be empty or it will contain and integer value.

Exported Access Programs

Routine name	In	Out	Exceptions
TileT	$\mathbb{N}$		
setValue	$\mathbb{N}$		
getValue		$\mathbb{N}$	
toString		String	

Semantics

State Variables

value: $\mathbb{N}$ *This value will be initially set to 0*

State Invariant

None

Access Routine Semantics

setValue($\mathbf{N}$ $newValue$):

- $value = newValue$

getValue():

- $out := value$

toString():

- $out := intToString(value)$

Board ADT Module

Template Module

BoardT

Uses

TileT

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	
getScore		$\mathbb{N}$	
getBoard		seq of seq of TileT	
setBoard	seq of seq of TileT		
moveRight			
moveLeft			
moveUp			
moveDown			
noMoreMoves	seq of seq of TileT	$\mathbb{B}$	
Transpose			

Semantics

State Variables

board: sequence [Size, Size] of TileT
score: $\mathbb{N}$
moves: $\mathbb{N}$

State Invariant

We need to make sure the number of occupied tiles and the number of unoccupied tiles should be equal to the area of the board

This is already dealt with when we initialize a 2-D array in Java using the $[][]TileTboard = newTileT[size][size]$

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 between 0 and 1.
- The board will be initialized with two random values.

Design decision

Before we start off, we need to answer a few questions to better construct the game. What would be the best way to represent the board? We are trying to replicate a Matrix so a 2-D ArrayList would work in our example. What will each element contain in the 2-D Matrix, we know the only possible values are either Integers or the element is empty.

What functions do we need for this Board Module and what are the access programs? We need to verify if any move is considered valid. For every move, we need to update the board. The board will also need a constructor. We would also like to know the state of a certain cell at the game board. This will help us identify if a move is valid or not. A move is considered invalid when the direction you want the tile to move to cannot move further. No further merge can take place and the board cannot move in any position. For every merge, a new tile will be placed in any free space at the board. The position of the new tile will once again be randomly generated. Therefore, we know the game will be over when no new tiles cannot be placed, there are no possible merges for any position and the tiles can no longer move. Therefore, we need to know if there are any valid moves left.

Four separate movement functions needs to be made, for each of them, we need to figure out if a merge is possible, if it is, we go ahead and merge them. Mathematics of Matrices tells us that moving up in a matrix is the same as transposing a matrix, moving left and transposing again. This means that our *moveUp* and *moveDown* functions will solely depend of the correctness of our $moveLeft$ and *moveRight* functions accordingly. We will also then need to create a *Transpose* function which takes in a board and returns the transpose of that board. A transposed board is simply a board with its rows and columns swapped.

Access Routine Semantics

BoardT():

- transition:

  $$\text{board} := \left\langle \begin{array}{c} \langle TileT_0, ..., TileT_3\rangle \\ \langle TileT_0, ..., TileT_3\rangle \\ \langle TileT_0, ..., TileT_3\rangle \\ \langle TileT_0, ..., TileT_3\rangle \end{array} \right\rangle$$

  status, moves = true, 0

  generateValues(board)

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

getScore():

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

getBoard():

- transition: none
- output: *out* := (board)
- exception: None

setBoard(newBoard):

- transition: board = newBoard
- output: none
- exception: None

moveRight():

- transition: None
- exception: None

moveLeft():

- output:
- exception:

moveDown():

- output:
- exception:

moveUp():

- output:
- exception:

noMoreMoves(board):

- output:
- exception:

Transpose():

- output:
- exception:

Local Functions

generateValues(board):

- output:
- exception:

insertRandom(board):

- output:
- exception:

canMergeLeft(i):

- output:

- exception:

canMergeRight(i):

- output:
- exception:

findEmptyLeftPosition(i):

- output:
- exception:

findEmptyRightPosition(i):

- output:
- exception:

canMergeTiles(board):

- output:
- exception:

isValid(moves):

- output:
- exception:

View Module

Uses

None

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
main			

Semantics

State Variables

greetings
start
instructions

State Invariant

None

Assumptions

- We assume that the makefile is already set up and we just need to run *makedemo* for the game to work

Access Routine Semantics

main():

- print the instructions
- print the greetings message
- print the board
- print the commands available to use
- Allow the scanner to use inputs from the reader
- Print the state of the game, and whether the game has ended

Local Function:

checkWinner(score):

- If the score is below 2048, we return false, else we return true

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 the view is in the same file since it would be easier to control the state of the game during runtime.
- I returned the points of the game after I enter a specific command, this is not a good idea since the ideal situation would want to know their points after each move. They would not want to manually check their score by entering a separate command.
- There seems to be an issue with my *moveRight* print function where the Tile value is printed differently from what is actually in the board. However, there are no issues with the board or the function itself. The confirmation of this is that the score gets updated as expected.
- The test cases designed were made to not deal witht he random values inserted into a board. Instead we fake a board and move values independently and ensure the updated tiles are in the right place and show the right values.
- My design has a high cohesion and low coupling by applying MVC. We keep high cohesion since we group relation functionalities in each module. The modules TileT, BoardT and View are fairly independent. A change in one of the modules does not heavily impact the other.
- Our game is fairly straight forward to use. Once we have the Makefile set up and the files imported, the game instructions once we run the command is easy to understand and interpret. Therefore, this project is general for the most part.
- The way my files are organized and because of low coupling, my code format is consistent and clean. The code is well documented and my algorithm design is rather simple to understand and very intuitive. Therefore, my modules created are consistent
- With regards to information hiding, my modules does a good job since we are making use of the MVC Model. The user will only get to see whats on their screen and that's the state of the board. The view module is most likely to change since the model works as expected. The only changes would be from the UI flow. Therefore, since we isolate both these modules, Information Hiding is very apparent in this project.

- With respect to minimaility. Our code makes use of Java libraries. We make use of no hardware and the modules used are minimalized. We haven't added anything redundant and every method included are essential to the workings of the game. Therefore, Our project is as minimal as it can be.