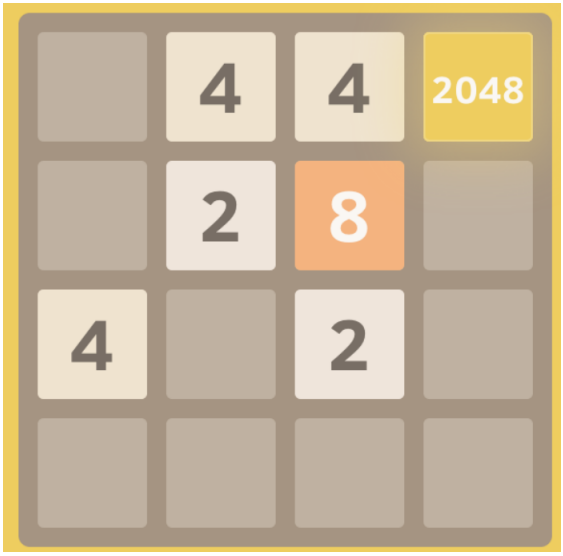
Assignment 4 Specification

SFWR ENG 2AA4

April 13, 2021

This Module Interface Specification (MIS) document contains modules, types and methods used for implementing the game 2048. At the start of the game, a board is created with two tiles in two random cells on the board containing a 2 or a 4 in each. The user must specify the direction that the board shifts in order to merge the tiles that appear. When two tiles merge, their value get added together and leave behind a single tile instead. After a direction has been specified and the tiles have been merged or moved in that direction, a new tile is added in a non-occupied spot containing a 2 or a 4. The possible directions are up ("w"), down ("s"), left ("a"), and right ("d"). This game play will continue until there are no more places to add tiles and there are no possible merges, or when the player merges the tiles to create the value 2048.

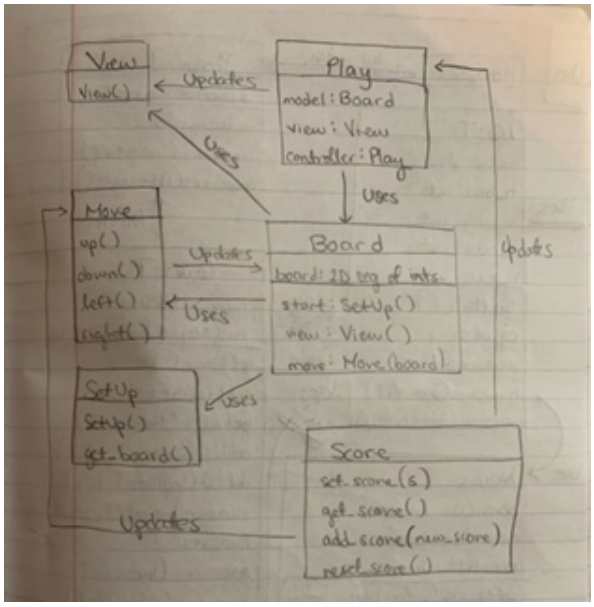


The above board visualization is from http://play2048.co/

Overview of the design

This design applies Module View Specification (MVC) design pattern and Singleton design pattern. The MVC components are *Play* (controller module), *Board* (model module) and *View* (view module). The Singleton pattern is specified and implemented by *Play* and *View*.

A UML diagram is provided below for visualizing the structure of this software architecture:



The MVC design pattern is specified and implemented in the following way: the module *Board* stores the current game board state and the status of the game. It also allows for modifications to the current board state via its methods. A view module *View* can display the state of the game board at a given time using text-based graphics printed to the user's console/screen. The controller *Play* is responsible for handling input actions from the player, as well as displaying the rules and other messages for the user to see and interact with.

For *Play*, they can get the score of the game by using the abstract method *Score.get_score*() to check if the user has gotten to the value of 2048.

Likely changes my design considers:

Data structure for storing the game board.
Changes to the user input commands (play again ("y"), try again ("t") and keep going ("k")).

Changes to the scoring (input does not have to be power of 2 so could increase/decrease amount of points for high scores)

SetUp Module

Template Module

SetUp

Uses

None

Syntax

Exported Constants

None

Exported Types

SetUp = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new SetUp			

Semantics

State Variables

game_board := $seq[4] of seq[4] of \mathbb{N}$
where game_board :=<<>, <>, <>, <>>

State Invariant

None

Assumptions

None

Access Routine Semantics

new SetUp():

- transition: game_board(random_spot() = random_num()), such that there are two unique cells filled once the transition is finished.
- output: *out* := *none*
- exception: none

get_board():

- output: *out* := game_board
- exception: none

Local Functions

random_num: none →N
random_num() ≡ ($0 <= r < 10 -> 2 | 11 <= r <= 99 -> 2$) # *Where num is a random number such that a 2 is 90 percent likely, and 4 is 10 percent likely to occur, where r is a random number between 0 and 100*

random_entry: none → seq of N
random_entry() ≡ cell # *Where cell is a random set of coordinates from the grid where each space in the grid is equally likely*

Score (Abstract Object)

Module

Score

Uses

None

Syntax

Exported Constants

None

Exported Types

Score = ?

Exported Access Programs

Routine name	In	Out	Exceptions
set_score	$\mathbb{N}$		
get_score		$\mathbb{N}$	
add_score	$\mathbb{N}$		
reset_score			

Semantics

State Variables

score: $\mathbb{N}$

State Invariant

None

Assumptions

The state variable will be set to zero (to start new score for game) before it is modified.
It is also assumed that all inputs will be powers of 2.

Access Routine Semantics

set_score(s):

- transition: score := s
- exception: none

get_score( ):

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

add_score(new_score):

- transition: score := score + new_score
- out: none
- exception: none

reset_score():

- transition: score := 0
- out: none
- exception: none

Move

Module

Move

Uses

Score

Syntax

Exported Constants

None

Exported Types

Move = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Move	$seq[4]of seq[4]of \mathbb{N}$		
up			
down			
left			
right			

Semantics

State Variables

game_board := $seq[4]of seq[4]of \mathbb{N}$

State Invariant

None

Assumptions

Assumes the Board object that the Move class will be used in will be set up first before engaging in a move.

Access Routine Semantics

Move(board):

- transition: game_board := *board*
- out: none
- exception: none

up( ):

- transition: *# Loop through board starting at second highest row and work towards last row. Check if cell above current cell has same value, if so merge(current, above cell). If cell above current cell is* 0*, make above cell value the value of the current cell, current cell value now empty. Then after all cells have been shifted, add_new_block to game_board*
- exception: none

down( ):

- transition: *# Loop through board starting at second lowest row and work towards first row. Check if cell below current cell has same value, if so merge(current, below cell). If cell below current cell is* 0*, make below cell value the value of the current cell, current cell value now empty. Then after all cells have been shifted, add_new_block to game_board*
- exception: none

left( ):

- transition: *# Loop through board starting at second leftmost row and work towards rightmost row. Check if cell to left of current cell has same value, if so merge(current, left cell). If cell left of current cell is* 0*, make left cell value the value of the current cell, current cell value now empty. Then after all cells have been shifted, add_new_block to game_board*

- exception: none

right( ):

- transition: *# Loop through board starting at second rightmost row and work towards leftmost row. Check if cell to right of current cell has same value, if so merge(current, right cell). If cell right of current cell is* 0*, make right cell value the value of the current cell, current cell value now empty. Then after all cells have been shifted, add_new_block to game_board*

- exception: none

Local Functions

merge: seq of $\mathbb{N}$, seq of N $\rightarrow$ *none*
merge(cell1, cell2) $\equiv$ game_board[*cell*1[0]][*cell*[1]] = 0, game_board[*cell*2[0]][*cell*2[1]] = game_board[cell2[0]][cell2[1]] x 2
Score := *Score*+ game_board[cell2[0]][cell2[1]]
 # Where merge changes the cell1 to a value of 0*, and doubles the value of cell2,then adds the value in cell2 to the Score abstract object*

add_new_block: none $\rightarrow$ *none*
add_new_block() $\equiv$ game_board[*random_entry*()[0]][*random_entry*()[1]] = *random_num*()
 # Where another random number is added in a random empty spot on the game board, if no spot is free, do not add block

random_num: none $\rightarrow$N
random_num() $\equiv$ ($0 <= r < 10 -> 2 | 11 <= r <= 99 -> 2$) *# Where num is a random number such that a 2 is 90 percent likely, and 4 is 10 percent likely to occur, where r is a random number between 0 and 100*

random_entry: none $\rightarrow$ seq of N
random_entry() $\equiv$ cell *# Where cell is a random set of coordinates from the grid where each space in the grid is equally likely*

Board (ADT)

Module

Board

Uses

SetUp, View, Move

Syntax

Exported Constants

None

Exported Types

Board = ?

Exported Access Programs

Routine name	In	Out	Exceptions
start			
view			
move	String		
get_board		seq of (seq of) $\mathbb{N}$	
get_board_value	$\mathbb{N}, \mathbb{N}$	$\mathbb{N}$	

Semantics

State Variables

game_board: sequence of (seq of)$\mathbb{N}$

State Invariant

None

Assumptions

Move input (direction) assumed to be only "wasd" characters.

Access Routine Semantics

start( ):

- transition: game_board := SetUp.get_board()
- out: none
- exception: none

view( ):

- transition: *# New instance of View(game_board) to show player the state of the board*
- exception: none

move(direction):

- transition: *# New instance of Move(game_board)*

  ($\langle$direction = "w" $\rightarrow$ $Move.up()$|direction = "s" $\rightarrow$ $Move.down()$|direction = "a" $\rightarrow$ $Move.left()|True \rightarrow Move.right()\rangle$
- exception: none

get_board( ):

- out: game_board
- exception: none

get_board_value(i, j):

- out: game_board[i][j]
- exception: none

View Module

Module

View

Uses

Score

Syntax

Exported Types

View = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new View	grid of $\mathbb{N}$		

Semantics

Environmental Variables

window: A portion of the computer screen that displays the game and messages.

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

View(game_board)

- transition: *window* := Displays Score using Score abstract object, displays all cells in game_board

Critique of Design

- The design was consistent in how the functions were able to be used in a consistent way in the Board class. Also the naming of game_board variables was consistent, as well as ways to access this variable across all modules that involved it.

- Some methods like get_score, get_board and get_board_value are not essential in order for the game to work, but allowed ease of use when accessing this information from other classes to help get the status of the game more easily.

- The methods up(), down(), left() and right() in the Move class are minimal as they allow movement in each specific direction as opposed to be able to move in all the different ways using one method. This allows for better usability as the programmer knows exactly which method moves which way and if they wanted to change the functionality of a certain direction only then this would allow them to do so. It also allows for better readability as any future programmers that may read it know how each direction will function in that method.

- The View method allows for some generality as it does not specify the size of the board that is being printed for the user, it just prints based on the length of the first row. This means that any square board would be able to be printed out using this method. This is also true in the Board class, however the SetUp and move methods are specified for a 4x4 grid. This could be changed to be more general design if there was an input in the SetUp constructor that allowed for the changing of the size of the grid to fit the input parameters, and in the Move method in the loops instead of specifying the loop stop when i less than 4 which is the length of the grid, that could have been changed to simply the length of the rows for i and length of columns for j to be checked instead.

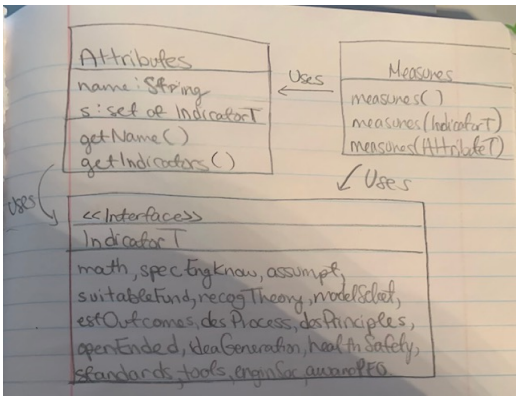
- This design was not general in its implementation of the controls as the user can only use "wasd" to play instead of also being able to use the arrow keys or "ijkl" is they wanted to. A way to fix this to make it more general is to ask the user before the game starts to input four characters that they would like to use to control the game which would then be incorporated into the controls.

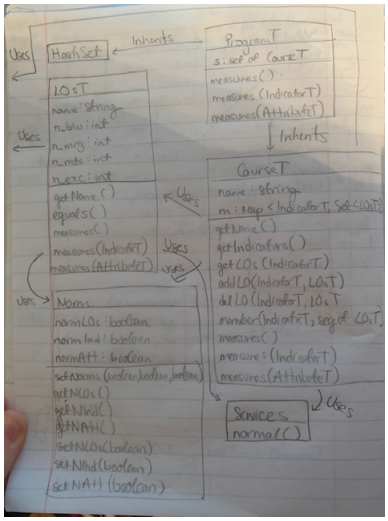
- This design has high cohesion as all the modules relate to each other as a result of MVC. For example, Score is used in Move, Board and View, and SetUp, Move and View being used in Board. All of these methods working together allow the game to function and it would not be able to function correctly if one of them was not working correctly, as they are all essential to making the Play function work and allowing the user to play the game.

- I specified Board module as an ADT as it makes it easier to create a new Board object when restarting a game.
- The score was implemented as an abstract object as it allowed the Score to be updated and used from any class when needed, which allowed for ease of use in accessing the score of the current game.
- The testing of these methods were made to try to prove the correctness of the modules and their methods. This was done by testing each method in a variety of ways to be able to catch any errors that may have arisen from that process. The testing allowed for the game to be played so that the user does not need to worry about the game crashing from an incorrect input. This was done in the Play method as if the user entered an incorrect input, the program would ask the user to try again until their input was correct.
- This design supports information hiding as there is no way to modify the SetUp or View methods, as well as the state variables for Move and Board. This allows the main board that the user is playing on to be private and keep the user from cheating. However, the Score method can be set to a different score at any time using its setter methods but that does not guarantee the win condition, only that they have a higher score for the leader boards. The win condition is also hidden from the user and cannot be modified, but a way to make this even more hidden is if it as inside a Check_Win class instead of the method in Play, as then the user would have a harder time finding the win condition.

Answers to Questions

1. UML diagram for modules in $A3$





where LOsT and CourseT use Measures (it was hard to try to line up)

2. Convex Hull Control Flow Graph

points = self.points
start = point[0]
min_x = start.x
p in points
p.x < min_x
min_x = p.x
start = p
point = start
self.hull.points.append(start)
far_point = None
far_point is not start
loop
p, loop
p is point
continue
else
p1 = p
loop
far_point = p1
p2 loop
else
p2 = point
or p2 = p1
direction