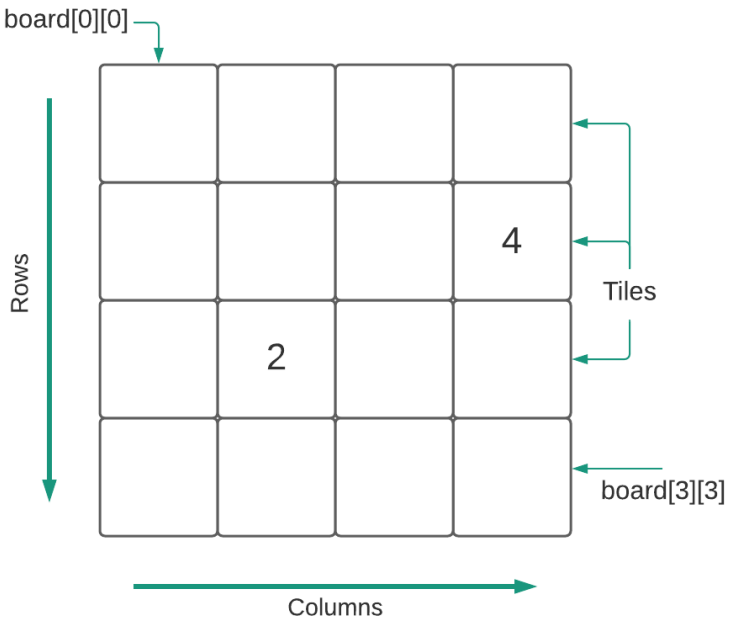
Assignment 4, Design Specification

SFWR ENG 2AA4

April 12, 2021

This Module Interface Specification (MIS) document contains modules and methods for implementing the game 2048. At the start of each game, a blank 4 by 4 board is provided to the user with 2 randomly positioned tiles of values 2 or 4. The user can move the non-zero tiles of board up, down, left, or right. The goal of the game is to move equal tiles into each other and combine them into higher valued tiles and ultimately reach the 2048 tile. In this MIS, tiles are simply denoted by natural numbers within a 2D matrix. After every valid move (a move that alters the board), a new tile can be added by replacing a zero tile. The game is lost if the board cannot be altered regardless of the direction of movement.

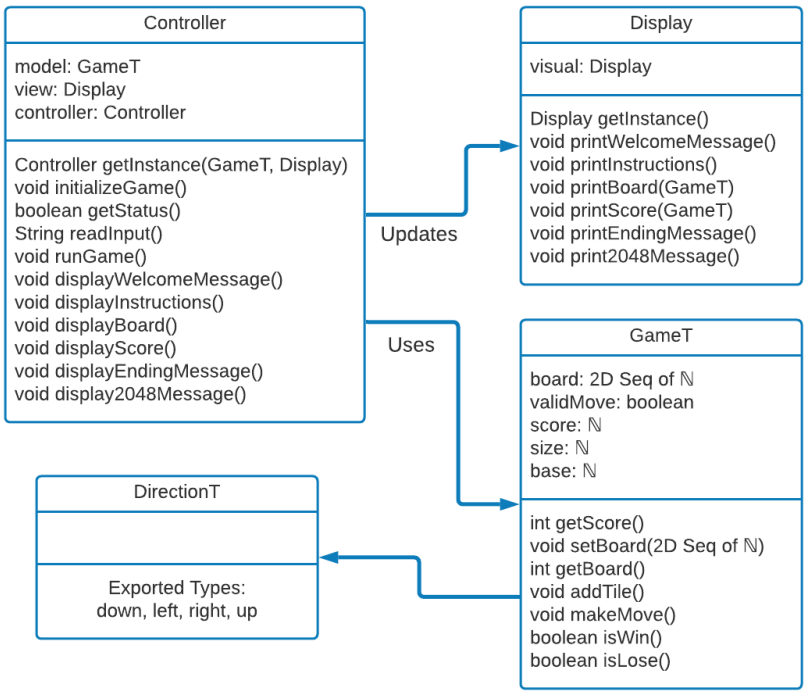


1 Overview of the design

This design applies the Module View Specification (MVC) and the Singleton design patterns. The MVC design pattern separates the computational elements from input and output elements. For this design, the MVC components are GameT (model module), Display (view module), and Controller (controller module). The model GameT module encapsulates the system's data as well as the operations on the data storing the state of the game board and the status of the game. The view Display module displays the data from the GameT component displaying the state of the game board and the status of the game using ASCII-based graphics. The controller module separately handles the input actions and calls the appropriate access routines of the model and view modules. Due to the MVC design pattern implementation, this design provides ease of modification.

The Singleton design pattern was implemented for the Display and Controller modules where there is only one instance - the getInstance() method is used to obtain the abstract object for each.

An UML diagram is provided below for visualizing this software architecture.



Likely Changes my design considers:

- My design considered likely changes regarding the data structure that is used for storing the game board. You can use a nested Array, nested ArrayList, or any other data structure that can form a 2D matrix.
- My design also considered that it is likely for the board size to change. You can effortlessly change the size of the matrix storing the board by altering the "Size exported constant. The access routines all consider the "Size variable when iterating through the tiles of the board. You can change the board from the default 4 by 4 to 5 by 5, 6 by 6, etc.
  - This change is also considered when adding a new random tile into the board as the size of the current board is taken into consideration.
  - This change is also considered within the Display module. The printBoard() access routine considers the boards current size when displaying the content to the terminal. Although it may not look as good for large boards, it will still work.
- My Display module also considers different flows of the game and as such provides different messages to be displayed.
- My design also considers that the controller may take in inputs of various kinds. As such, it provides a standardized input for the makeMove() access routine within the GameT module. No matter what input is provided to the controller, the controller can call the makeMove() access routine with the respective DirectionT object.
- My design considers that the game may be played with a base of 3 rather than 2, or any other style of gameplay. As such the "Base constant can be changed to implement many types of gameplay (base of N) additionally, this constant changes what random tiles are added while the game will still be considered "won if a tile is greater than 2048. This change is also anticipated within the Display module by providing a general end-game message rather than a specific message for 2048.
- It is also considered that the gameplay may differ in regards to how many or when new tiles are added to the board. As such, the addTile() access routine is left to be manually used by a controller rather than automatically adding new tiles to the board after every move.

DirectionT Module

Module

DirectionT

Uses

None

Syntax

Exported Constants

None

Exported Types

DirectionT = {
up, *#Moving the game tiles upward.*
down, *#Moving the game tiles downward.*
left, *#Moving the game tiles to the left.*
right, *#Moving the game tiles to the right.*
}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Considerations

When implemented in Java, use enums.

GameT ADT Module

Template Module

GameT

Uses

DirectionT

Syntax

Exported Constants

Size = 4 // Size of the board in each direction Base = 2 // Base tile value

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new GameT		GameT	
makeMove	DirectionT		
addTile			
getScore		$\mathbb{N}$	
setBoard	seq [Size,Size] of $\mathbb{N}$		
getBoard		seq [Size,Size] of $\mathbb{N}$	
isWin		$\mathbb{B}$	
isLose		$\mathbb{B}$	

Semantics

State Variables

board: sequence [Size,Size] of $\mathbb{N}$
score: $\mathbb{N}$
validMove: $\mathbb{B}$ // Previous move was valid iff it altered the board.

State Invariant

None

Assumptions

It is assumed that the order of sequences within the board are be maintained. Also that when items in a sequence are traversed, they are traversed in order from start to end. It is assumed that the GameT constructor is ran before any other access routine is called. It is assumed that a random tile is added manually after every valid move - this process is not automatic within this module. setBoard() will be used only for testing purposes and will not be a part of the actual game play - as such, it is assumed that correct input will be provided by the developer.

Access Routine Semantics

new GameT():

- transition: // Initiates the board to a [size][size] matrix of zeros.

  $$score, validMove, board := 0, true, \langle \begin{matrix} \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \end{matrix} \rangle$$

  addTile(), addTile() // Initial 2 random tiles

- output: $out :=$ self
- exception: none

getScore():

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

setBoard(b):

- transition: $board := b$
- exception: It is assumed that the developer testing will provide correct input. This functionality is meant only for testing purposes.

getBoard():

- output: $out := board$
- exception: none

addTile(): // Changes a random 0 from the board to a 2 or 4.

- transition: $(validMove) \Rightarrow (board[i][j] := val)$

  such that $(val = \langle Base, Base{*}2 \rangle [random(0,1)]) \wedge (i, j := random(0, size{-}1), random(0, size{-}1))$

- exception: none

makeMove($move$):

- transition: $validMove := false$

$$(move = up \Rightarrow board := \text{transpose}(\langle row : \text{seq of } \mathbb{N} \mid row \in b : \text{combine\_seq\_left}(row) \rangle)$$
$$\text{such that } b = transpose(board))$$
$$(move = down \Rightarrow board := \text{transpose}(\langle row : \text{seq of } \mathbb{N} \mid row \in b : \text{combine\_seq\_right}(row) \rangle)$$
$$\text{such that } b = transpose(board))$$
$$(move = left \Rightarrow board := (\langle row : \text{seq of } \mathbb{N} \mid row \in b : \text{combine\_seq\_left}(row) \rangle$$
$$(move = right \Rightarrow board := (\langle row : \text{seq of } \mathbb{N} \mid row \in b : \text{combine\_seq\_right}(row) \rangle$$

- exception: none

isWin():

- out: $(\exists\, i, j : \mathbb{N} \mid i, j \in [0..\text{Size} - 1] \wedge ((b[i][j] \geq 2048)))$
- exception: none

isLose(): // Checks if any move could alter a copy of the board.

- out: $(board = result.getBoard())$

  such that result.makeMove(up), result.makeMove(down),
  result.makeMove(left), result.makeMove(right)

  such that $result : GameT \wedge result.setBoard(board)$

- exception: none

Local Functions

Note Pertaining To Local Functions: The order of sequences should be maintained. Items in a sequence should be traversed in order.

// Combines non-zero equal adjacent elements and shifts them to the left of the sequence.
combine_seq_left : seq of $\mathbb{N} \to$ seq of $\mathbb{N}$
combine_seq_left(seq) $\equiv$ shift_seq_left(adj)
such that $(k : \mathbb{N} \mid k \in [0..\text{Size} - 2] : (adj[k] = adj[k+1] \implies adj[k] = adj[k] * 2,$
$adj[k+1] = 0, score := score + adj[k] * 2)$
such that $adj :=$ shift_seq_left(seq)

$$transition : (seq \neq \text{combine\_seq\_left}(seq)) \implies validMove := true$$

// Combines non-zero equal adjacent elements and shifts them to the right of the sequence.
combine_seq_right : seq of $\mathbb{N} \to$ seq of $\mathbb{N}$
combine_seq_right(seq) $\equiv$ shift_seq_right(adj)
such that $(k : \mathbb{N} \mid k \in [1..\text{Size} - 1] : (adj[k] = adj[k-1] \implies adj[k] = adj[k] * 2,$
$adj[k-1] = 0, score := score + adj[k] * 2)$
such that $adj :=$ shift_seq_right(seq)

$$transition : (seq \neq \text{combine\_seq\_right}(seq)) \implies validMove := true$$

// Shifts non-zero elements to the left of the sequence.
shift_seq_left : seq of $\mathbb{N} \to$ seq of $\mathbb{N}$
shift_seq_left(seq) $\equiv \langle i : \mathbb{N} \mid i \in seq \wedge i \neq 0 : i \rangle || \langle j : \mathbb{N} \mid j \in seq \wedge j = 0 : j \rangle$

// Shifts non-zero elements to the right of the sequence.
shift_seq_right : seq of $\mathbb{N} \to$ seq of $\mathbb{N}$
shift_seq_right(seq) $\equiv \langle i : \mathbb{N} \mid i \in seq \wedge i = 0 : i \rangle || \langle j : \mathbb{N} \mid j \in seq \wedge j \neq 0 : j \rangle$

// Interchanges the rows and columns of the 2D array.
transpose : seq of (seq of $\mathbb{N}$) $\to$ seq of (seq of $\mathbb{N}$)
transpose(b) $\equiv \langle i : \mathbb{N} | i \in [0..\text{Size} - 1] : \langle j : \mathbb{N} | j \in [0..\text{Size} - 1] : b[j][i] \rangle \rangle$

random : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
random(a, b) $\equiv$ random natural number between a and b (inclusive)

Display Module

Display

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		Display	
printWelcomeMessage			
printInstructions			
printBoard	GameT		
printScore	GameT		
printEndingMessage			
print2048Message			

Semantics

Environment Variables

window: Portion of the computer screen allocated to display the game board and messages.

State Variables

visual: Display

State Invariant

none

Assumptions

The Display constructor is called for each object instance before any other access routines are called for the object. The constructor can only be called once.

Access Routine Semantics

getInstance():

- transition: visual := (visual = null ⇒ new Display())
- output: self
- exception: none

printWelcomeMessage():

- transition: window := Displays a welcome message when user starts the game.
- exception: none

printInstructions():

- transition: window := Displays the instructions on how to play the game. Possible moves, how to win, when you lose.
- exception: none

printBoard(model):

- transition: window := Draws the game board onto the screen. The board is retrieved using the GameT functionality *game.getBoard*() and the tiles are individually printed to form a 4 by 4 board. Zeros are replaced by "O" to improve the look of the display. The board is displayed in such a way that the top left tile is board[0][0] and the bottom right tile is board[3][3]. Space is left between the tiles to make the board easy to visually scan.
- exception: none

printScore(model):

- transition: window := Displays the score of the GameT object using the *game.getScore*() functionality.
- exception: none

printEndingMessage():

- transition: window := Displays an ending "Game Over" message.
- exception: none

print2048Message():

- transition: window := Displays a message congratulating the player on reaching the 2048 tile.
- exception: none

Local Functions

Display: void $\rightarrow$ Display
Display() $\equiv$ new Display()

Controller Module

Controller

Uses

GameT, Display

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance	GameT, Display	Controller	
initializeGame			
getStatus		$\mathbb{B}$	
readInput		String	
runGame			
displayWelcomeMessage			
displayInstructions			
displayBoard			
displayScore			
displayEndingMessage			
display2048Message			

Semantics

Environment Variables

keyboard: Scanner(System.in) // Reading input from keyboard

State Variables

model: GameT
view: Display
controller: Controller

State Invariant

none

Assumptions

It is assumed that the Controller constructor is called for the object instance before any other access routines are called for the object. The constructor can only be called once. Also assuming that model and view instances are already initialized before calling the Controller constructor.

Access Routine Semantics

getInstance(model, view):

- transition: controller := (controller = null ⇒ new Controller(model, view))
- output: self
- exception: none

initializeGame():

- transition: model := new GameT()
- exception: none

getStatus():

- transition: None
- output: out := (model.isLose())
- exception: none

readInput():

- output: input : A String entered fromthe keyboard.
- exception: none

runGame():

- transition: Operational method for running the game. The game will start with the welcome message and provide the instructions on how the game can be played. The board will be displayed and displayed again after every move. If a valid move is made, a tile is added to the board. The game will automatically end and display the ending message if no moves can be made, or it will display a congratulatory message for reaching 2048 and continue.
- exception: none

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage()
- exception: none

displayInstructions():

- transition: view := view.printInstructions()
- exception: none

displayBoard():

- transition: view := view.printBoard()
- exception: none

displayScore():

- transition: view := view.printScore()
- exception: none

display2048Message():

- transition: view := view.print2048Message()
- exception: none

Local Functions

Controller : GameT $\times$ Display $\rightarrow$ Controller
Controller($model, view$) $\equiv$ new Controller($model, view$)

Critique of Design

- I chose to specify the GameT module an as ADT rather than an abstract object. This is because it is more convenient to create a new instance of the board after a user chooses to start a new game.
- The Controller and Display modules are specified as single abstract objects as these modules are shared resources and only one instance is required to control the actions during runtime. This way, conflicts and unexpected state changes can be avoided.
- The setBoard() method was included strictly for testing purposes it is therefore largely assumed that the developer using it will provide correct inputs and would use it to test other modules rather than test the method itself.
- Consistency can be expressed by maintaining a consistent pattern to how access routines and variables are named. It can also be expressed by how such things are ordered including the parameters to the access routines. My design puts a strong emphasis on consistency in order to make the MIS easy to follow:
  - Whenever the directions or direction-based conditions are listed, I made sure to always keep the order the same: up, down, left, right. This is also true for the DirectionT module.
  - The simple setters and getters of a module are grouped together and outlined first.
  - A 2D matrix input is always referred to as b to provide intuition about what the input is since the game matrix itself is called board.
  - Whenever a GameT input is needed or referred to, it is called model. Whenever a Display object is referred to, it is called view. Furthermore, the inputs for Controller are taken in the same order every time model, view.
  - The Display and Controller access routines share very similar names the only difference between most is that within Controller, they have prefix display and in Display the prefix is print.
- Essentiality describes how unnecessary access programs are omitted. My MIS implements essentiality as every access program is used to produce the flow of the 2048 game. Every access program is needed to make a unique move, print a message according to the state of the game, or check the status of the current game. The access routines that perform simple services are used in multiple stages and that is why they are separated such as the shift_seq_left and right routines that are essential to the movement of the board. The custom congratulatory message for reaching a 2048 tile may not be essential as it can be displayed just once per game, but because it is a rare occurrence, I separated it from the

rest of the design to maintain readability. It is still essential as it signifies reaching the main objective of the game its simply rarely used.

- The goal of generality is to solve more general problems. While designing my MIS, I decided to try and make the movement of the board as general as I can. As such, I created the shift_seq_left and shift_seq_right routines. These are fairly general and do not immediately associate with the 2048 game. They simply move the non-zero elements of a sequence to the left or right. Expanding from this, I created the combine_seq_left and combine_seq_right routines. These routines combine non-zero equal adjacent elements and shift them to the left or right. The Display and Controller routines are also kept semi-general as the messages do not rely on the game state and can simply be called if needed. The messages are not strictly bounded to the GameT objects state. Lastly, the DirectionT module was kept general using enumerated types to represent the directions that the board can move in.

- A minimal interface avoids access routines that offer two different services that might be requested separately by the user. I tried to keep my access routines minimal throughout the MIS getters, setters, and printing routines can easily be kept minimal. However, certain access routines are required to provide multiple services in order to keep the MIS easy to read and implement. The following explain instances where the principle of minimality is violated.

  - My GameT module violates minimality as it must initialize the board, validMove state variable, score, and must also add two random tiles to the board.
  - Furthermore, within the GameT combine_seq routines, I return a sequence output and also update the validMove state variable. This makes the MIS far easier rather than creating a new routine to update the validMove state variable. We can simply check if the routines actions altered the input sequence and that would tell if the move was valid.
  - Also within the combine_seq routines, I must update the score respectively to what elements are being combined I would not be able to do this outside of the routine that is doing the combining.
  - The Controller module cannot be minimal because playing a game consists of many different things going on at once making a move, displaying messages, and initializing new games.

- This design implements high cohesion and low coupling. This means that the components of the module are closely related this is expressed by the MVC design pattern that was incorporated. Low coupling is seen as the modules do not strongly depend on other modules while they are all needed to play the game, they do not crash or break without another.

- Information hiding is respected within my MIS design as things that are likely to change are hidden. The exported constants that determine the gameplay (such as size and base), the state variables, and the local functions are all kept private hidden. The implementation secrets are hidden from the user how the movements are actually conducted. All the user has access to is what they need to play.

- The print access routines within the Display (view) module are not completely necessary. The Controller module has access routines that simply call for the print access routines - therefore, instead of calling them we could simply put them within the Controller module. We do not necessarily need the functionality within the Display module - it would be enough within the Controller. However, this was done to maintain organization and also reduce the dependency of the game on just the controller. This way, we can leave running the game to the Controller and modify the messages within the Display module.

- Comprehensive unit testing was done for the GameT module while the other modules were indirectly yet thoroughly manually tested using the controller. Unit tests were conducted on carefully selected cases to cover many complex cases as well as base cases. For example, some boards used were the empty board, one with only corner tiles, one with all filled except one spot, one completely filled, alternating empty rows and columns, etc. I tested every public GameT method for most of these cases to see how the program performs in intricate situations.

Answers to Questions

Q1. Draw a UML diagram for the modules in A3